100 MAP MYŚLI ZE SPRING BOOTA

E-BOOK

Ucz się
Buduj
Rozwijaj
Osiągaj więcej!

Lepsze projekty
Lepszy kod
Lepsza kariera
Lepsza przyszłość!

OD FUNDAMENTÓW DO PRODUKCJI

Spring Boot 4.x • Java 17+ • Web • Data • Security • Testy • Docker • Cloud

1. **Fundamenty** — Spring Boot, projekt, uruchomienie
2. **Konfiguracja** — Właściwości, profile, środowiska
3. **REST API** — Kontrolery, walidacja, dokumentacja
4. **Dane i JPA** — Bazy danych, Spring Data, transakcje
5. **Security** — Uwierzytelnianie, Autoryzacja, OAuth2
6. **Testy** — Testy jednostkowe, integracyjne, E2E
7. **Integracje** — HTTP, komunikaty, zewnętrzne usługi
8. **Observability** — Logi, metryki, tracing, Actuator
9. **Reaktywność** — WebFlux, programowanie reaktywne
10. **Produkcja** — Docker, chmura, wdrożenia, skalowanie



Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

NAUKA | PRAKTYKA | ROZWÓJ

Spring Boot
Więcej niż framework
To możliwości!

SPIS TREŚCI

CZĘŚĆ 1/4 • MAPY 01–25

Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

Ucz się
Twórz
Rozwijaj
z Spring Boot!

01 MODUŁ 01 — FUNDAMENTY I ARCHITEKTURA SPRING BOOT

01 Ekosystem Spring — gdzie znajduje się Spring Boot? 01
02 Co naprawdę robi Spring Boot? 02
03 Anatomia projektu Spring Boot 03
04 Jak uruchamia się aplikacja Spring Boot? 04
05 @SpringBootApplication pod lupą 05
06 IoC, DI i ApplicationContext 06
07 Cykl życia beanów i zakresy 07
08 Component Scanning i stereotypy Springa 08
09 @Configuration i @Bean 09
10 Startery, Maven, Gradle i Dependency Management 10



Poznaj solidne fundamenty i zrozum, jak działa Spring Boot od środka.

02 MODUŁ 02 — AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

11 Jak działa Auto-Configuration? 11
12 Rodzina @Conditional 12
13 Back-off, wykluczanie i diagnostyka Auto-Configuration 13
14 Externalized Configuration — kolejność i priorytety 14
15 application.properties, YAML i Config Data 15
16 @ConfigurationProperties vs @Value 16
17 Profile, Profile Groups i konfiguracja środowisk 17
18 Zmienne środowiskowe, sekrety i kontenery 18
19 Logging w Spring Boot 19
20 Application Events, Runner i hooki startowe 20



Skonfiguruj aplikację jak profesjonalista i wykorzystaj pełen potencjał auto-konfiguracji.

03 MODUŁ 03 — SPRING MVC I REST API

21 Servlet Stack i wbudowany serwer 21
22 Droga żądania przez Spring MVC 22
23 @Controller, @RestController i Request Mapping 23
24 Path, Query, Header i Body — binding danych HTTP 24
25 Walidacja danych wejściowych 25



Twórz nowoczesne REST API i obsługuj żądania HTTP w Spring Boot.



Spring Boot
Więcej niż framework
To możliwości!

SPIS TREŚCI

CZĘŚĆ 2/4 • MAPY 26–50

Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

Ucz się
Twórz
Rozwijaj
z Spring Boot!

03 MODUŁ 03 — SPRING MVC I REST API

26 DTO, Entity i Record — rozdzielanie warstw 26
27 JSON i Jackson 3 w Spring Boot 4 27
28 Obsługa błędów REST 28
29 Filter vs Interceptor vs ControllerAdvice 29
30 Dobrze zaprojektowane REST API 30

Buduj czyste, nowoczesne API, które są zrozumiałe, łatwe w utrzymaniu i gotowe na rozwój.

04 MODUŁ 04 — BAZY DANYCH, JPA I TRANSAKCJE

31 DataSource, connection pool i JDBC 31
32 Spring Data Repository 32
33 JPA Entity i Persistence Context 33
34 Relacje JPA 34
35 Query Methods, JPQL i Native Query 35
36 Projection, Specification i Query by Example 36
37 @Transactional bez magii 37
38 LAZY, EAGER i problem N+1 38
39 Qptimistic vs Pessimistic Locking 39
40 Flyway i Liquibase 40

Stabilne i wydajne warstwy danych to solidny fundament każdej aplikacji Spring Boot.

DOBRY KOD LEPSZE JUTRO

05 MODUŁ 05 — SPRING SECURITY

41 Security Filter Chain 41
42 Authentication vs Authorization vs SecurityContext 42
43 Użytkownik, hasło i PasswordEncoder 43
44 HttpSecurity i reguły dostępu do endpointów 44
45 Method Security 45
46 Session vs Stateless Authentication 46
47 CSRF 47
48 CORS 48
49 OAuth2 Resource Server 49
50 OAuth2 Client, Login i Authorization Server 50

Twórz bezpieczne aplikacje, chroń dane i użytkowników oraz korzystaj z nowoczesnych mechanizmów autoryzacji.



Spring Boot
Więcej niż framework
To możliwości!

SPIS TREŚCI

CZĘŚĆ 3/4 • MAPY 51–75

Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

Ucz się
Twórz
Rozwijaj
z Spring Boot!

06 MODUŁ 06 — TESTOWANIE APLIKACJI

51 Strategia testów w Spring Boot 51
52 Test jednostkowy bez Spring Context 52
53 @SpringBootTest — co naprawdę uruchamia? 53
54 Test Slices: @WebMvcTest i @DataJpaTest 54
55 Testowanie REST z MockMvc 55
56 Testowanie aplikacji przez prawdziwe HTTP 56
57 Testowanie repozytoriów i bazy danych 57
58 Testcontainers 58
59 @ServiceConnection i integracja z Testcontainers 59
60 Spring REST Docs 60

Lepsze testy.
Wyższa jakość.
Pewniejszy kod
w Spring Boot!

07 MODUŁ 07 — INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

61 RestClient — synchroniczny klient HTTP 61
62 WebClient — klient HTTP i model reactive 62
63 Application Events i @EventListener 63
64 @Async, Executor i pule wątków 64
65 @Scheduled i zadania cykliczne 65
66 Spring Cache 66
67 Redis 67
68 Kafka w Spring Boot 68
69 RabbitMQ i AMQP 69
70 Niezawodna komunikacja: idempotencja, retry, DLQ i outbox 70

Integracje,
asynchroniczność
i skalowalność.
Buduj systemy
na większą skalę
z Spring Boot!

08 MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

71 Spring Boot Actuator — mapa możliwości 71
72 Health, Liveness i Readiness 72
73 Micrometer Metrics 73
74 Własne metryki i cardinality 74
75 Trzy filary observability 75

Monitoruj.
Analizuj.
Miej pełną
widoczność
w Spring Boot!



SPIS TREŚCI

CZĘŚĆ 4/4 • MAPY 76–100

Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

Spring Boot
Więcej niż framework.
To możliwości!

Ucz się
Twórz
Rozwijaj
z Spring Boot!

08 MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

76 Distributed Tracing i OTLP 76
77 Log correlation i Structured Logging 77
78 Graceful Shutdown i Application Availability 78
79 Bezpieczny Actuator w produkcji 79
80 Diagnostyka JVM i aplikacji 80

Obserwability i diagnostyka dają pewność, stabilność i kontrolę w produkcji.

09 MODUŁ 09 — REACTIVE, WYDAJNOŚĆ I ARCHITEKTURA

81 Spring MVC vs WebFlux — jak dokonać wyboru? 81
82 Reactor: Mono, Flux i Backpressure 82
83 Jak przepływa żądanie w WebFlux? 83
84 R2DBC i reaktywny dostęp do danych 84
85 Virtual Threads w Spring Boot 85
86 Blocking vs Async vs Virtual Threads vs Reactive 86
87 WebSocket i Server-Sent Events 87
88 GraphQL w Spring Boot 88
89 Architektura modułowa aplikacji Spring Boot 89
90 Granice transakcji, domeny i integracji 90

Systemy reaktywne, wyższa wydajność i skalowalna architektura otwierają nowe możliwości w Spring Boot.

DOBRY KOD LEPSZE JUTRO

10 MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

91 Od kodu do executable JAR 91
92 Jak zbudowany jest Spring Boot JAR? 92
93 Dockerfile vs Cloud Native Buildpacks 93
94 Docker Compose + Spring Boot Service Connections 94
95 Spring Boot na Kubernetes 95
96 HTTPS, TLS i SSL Bundles 96
97 GraalVM Native Image i AOT 97
98 Optymalizacja aplikacji Spring Boot 98
99 Migracja Spring Boot 3.5 → 4.x 99
100 Spring Boot Production Checklist 100

Od buildu po chmurę — praktyczna wiedza, która przygotuje Twoje aplikacje na prawdziwą produkcję.



MAPA 01/100

EKOSYSTEM SPRING — GDZIE ZNAJDUJE SIĘ SPRING BOOT?

Jak elementy rodziny Spring łączą się w jedną platformę do budowy aplikacji Java

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 SPRING FRAMEWORK — RDZEŃ

FUNDAMENT CAŁEGO EKOSYSTEMU

- **IoC / DI** — Zarządzanie obiektami i zależnościami (wstrzykiwanie zależności).
- **Spring MVC** — Aplikacje webowe w modelu servlet.
- **WebFlux** — Programowanie reaktywne.
- **Obsługa zdarzeń, walidacja, transakcje i integracja.**
- To fundament, na którym działa **Spring Boot**.

2 SPRING BOOT — WARSTWA UPRASZCZAJĄCA

SZYBCIEJ • PROŚCIEJ • PRODUKCYJNIE

- **Przyspiesza start projektu** i ogranicza ręczną konfigurację.
- **Auto-configuration** dobiera domyślne ustawienia na podstawie classpathu i właściwości.
- **Startery** grupują typowe zależności.
- **Wbudowany serwer** (np. Tomcat) ułatwia uruchamianie aplikacji.
- **Actuator** wspiera monitoring i diagnostykę.

Spring Boot nie zastępuje Spring Framework — on go wygodnie konfiguruje.

3 NAJWAŻNIEJSZE OBSZARY EKOSYSTEMU

SPECJALISTYCZNE MODUŁY DO RÓŻNYCH ZADAŃ

- **Spring Data** — Repozytoria, JPA, JDBC, R2DBC.
- **Spring Security** — Uwierzytelnianie, autoryzacja, filtry, OAuth2.
- **Spring Integration / Messaging** — Komunikacja i integracje.
- **Spring Batch** — Przetwarzanie wsadowe.
- **Spring GraphQL** — API oparte na GraphQL.
- **Spring for Apache Kafka / AMQP** — Komunikacja zdarzeniowa.
- **Spring Cloud** — Narzędzia dla systemów rozproszonych i chmury (konfiguracja, rejestracja usług, itp.).

4 JAK TO SIĘ ŁĄCZY?

OD POMYSŁU DO DZIAŁAJĄCEJ APLIKACJI

Pomysł biznesowy → **Spring Framework** (mechanizmy) → **Spring Boot** (szybka konfiguracja) → **Moduły specjalistyczne** (Data, Security, Messaging...) → **Gotowa aplikacja**

Te same klocki. Większe możliwości!

5 KIEDY WYBIERASZ SPRING BOOT?

IDEALNY, GDY POTRZEBUJESZ:

- Gdy chcesz szybko uruchomić aplikację produkcyjną.
- Gdy potrzebujesz rozsądnych domyślnych ustawień.
- Gdy zależy Ci na łatwej konfiguracji środowisk.
- Gdy chcesz wygodnie wdrażać REST API, aplikacje webowe, usługi i integracje.

6 SPRING FRAMEWORK VS SPRING BOOT

NAJWAŻNIEJSZE RÓŻNICE

Aspekt	Spring Framework	Spring Boot
Zakres	Zestaw mechanizmów	Wygodna warstwa startowa
Konfiguracja	Więcej ręcznych decyzji	Więcej gotowych ustawień
Start projektu	Wolniejszy	Szybszy
Monitoring	Osobna konfiguracja	Actuator
Serwer	Częściej konfiguracja ręczna	Często serwer wbudowany (np. Tomcat)

7 W PRAKTYCE

TYPOWY PROJEKT WYKORZYSTUJE:

- Spring Boot
- Spring Web
- Spring Data
- Spring Security

Dzięki temu zyskujesz spójny stos technologiczny, ale nadal pracujesz na mechanizmach Spring Framework.

8 UWAGA

- Nie każdy moduł Springa jest częścią samego Spring Boot.
- Boot porządkuje i upraszcza pracę, ale architekturę aplikacji nadal projektuje programista.

9 SZYBKIE PODSUMOWANIE

- **Spring Framework** = fundament.
- **Spring Boot** = wygodny start i konfiguracja.
- **Inne projekty Spring** = konkretne możliwości.
- Razem tworzą ekosystem do budowy nowoczesnych aplikacji Java.

Lepsze aplikacje tworzy się z dobrych narzędzi i jeszcze lepszych pomysłów!



MAPA 02/100

CO NAPRAWDĘ ROBI SPRING BOOT?

Najważniejsze mechanizmy, które przyspieszają budowę aplikacji

Ten sam ekosystem. Większa produktywność. Realne możliwości.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST SPRING BOOT?

Rozszerzenie ekosystemu Spring, które upraszcza konfigurację i uruchamianie aplikacji.

Nie zastępuje Spring Framework — wykorzystuje jego mechanizmy.

2 AUTO-CONFIGURATION

Analizuje classpath, właściwości i istniejące beany.

Tworzy sensowne ustawienia domyślne.

Działa warunkowo dzięki adnotacjom z rodziny `@Conditional`.

3 STARTERY

Grupują typowe zależności pod jednym artefaktem.

Przykłady:

- `spring-boot-starter-web`
- `spring-boot-starter-data-jpa`
- `spring-boot-starter-security`
- `spring-boot-starter-test`

4 WBUDOWANY SERWER

- Aplikacja może działać jako samodzielny JAR.
- Najczęściej używany jest **Tomcat**; możliwe są też **Jetty** lub **Undertow**.
- Ułatwia uruchamianie i wdrażanie.

5 ZEWNĘTRZNA KONFIGURACJA

- application.properties lub application.yml.
- Profile: `dev` , `test` , `prod`
- Możliwość nadpisywania przez zmienne środowiskowe i argumenty startowe.

6 ACTUATOR

- Dostarcza endpointy diagnostyczne i metryki.
- Pomaga sprawdzać health, info, metrics, env i wiele innych aspektów działania aplikacji.

7 JAK TO DZIAŁA W TLE?

`main()` → SpringApplication .run(...) → przygotowanie Environment → tworzenie ApplicationContext → component scanning + auto-configuration → uruchomienie serwera → aplikacja gotowa

8 CZEGO SPRING BOOT NIE ROBI?

- Nie projektuje architektury za programistę.
- Nie zastępuje znajomości HTTP, baz danych czy bezpieczeństwa.
- Nie eliminuje potrzeby testów i świadomych decyzji technicznych.

9 W PRAKTYCE

Dla prostego REST API często wystarczy:

`Web` + `Validation` + `Actuator`

Dla aplikacji biznesowej często dochodzą:

`Data JPA` + `Security` + `Flyway` + `Testcontainers`

10 UWAGA

Zbyt duża liczba starterów „na zapas" zwiększa złożoność i czas startu.

Domyślne ustawienia pomagają, ale warto rozumieć, co dzieje się pod spodem.

11 SZYBKIE PODSUMOWANIE

- Spring Boot przyspiesza start.
- Automatyzuje konfigurację.
- Upraszcza budowę aplikacji produkcyjnych.
- Nadal opiera się na fundamentach Spring Framework.

```java
@SpringBootApplication
public class App {
  public static void main(String[] args) {
    SpringApplication.run(App.class, args);
  }
}
```

Mniej konfiguracji, więcej tworzenia wartości!



MAPA 03/100

ANATOMIA PROJEKTU SPRING BOOT

Jak uporządkować kod, zasoby i konfigurację od pierwszego dnia

Dobrze zorganizowany projekt to szybszy rozwój i mniej problemów!

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 PLIKI BUDOWANIA

ZARZĄDZAJĄ ZALEŻNOŚCIAMI I KONFIGURACJĄ

Maven

`pom.xml`

Gradle

`build.gradle`

lub `build.gradle.kts`

Tu znajdują się zależności, pluginy i konfiguracja builda.

2 STRUKTURA KATALOGÓW

PODSTAWOWY UKŁAD PROJEKTU

```
src
├── main
│   ├── java          → kod aplikacji
│   └── resources     → konfiguracja i zasoby
└── test
    ├── java          → testy
    └── resources     → dane testowe i konfiguracja testów
```

3 KLASA STARTOWA

PUNKT WEJŚCIA DO APLIKACJI

```
@SpringBootApplication
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(
      DemoApplication.class, args);
  }
}
```

- Zawiera metodę main() i zwykle adnotację @SpringBootApplication.
- Najlepiej umieścić ją wysoko w strukturze pakietów, aby component scanning obejmował resztę aplikacji.

4 KOD W src/main/java

TYPOWE PAKIETY I ICH ODPOWIEDZIALNOŚĆ

controller / api	Warstwa HTTP (endpointy, request/response).
service	Logika biznesowa.
repository	Dostęp do danych (np. Spring Data JPA).
domain / model	Model domenowy (encje, DTO).
config	Konfiguracja techniczna (np. beany, ustawienia).

5 ZASOBY W src/main/resources

KONFIGURACJA, PLIKI I WIDOKI

application.yml lub application.properties	Główna konfiguracja aplikacji.
static/	Pliki statyczne (np. CSS, JS, obrazy).
templates/	Widoki serwerowe (np. Thymeleaf).
Pliki SQL	Skrypty bazy danych (np. schema.sql, data.sql).
Komunikaty i inne	Pliki z komunikatami, konfiguracje pomocnicze.

6 TESTY

JAK PISAĆ I ORGANIZOWAĆ TESTY

Testy jednostkowe	W src/test/java, testują pojedyncze klasy i komponenty.
Testy integracyjne	Również w src/test/java, ale dobrze oznaczone (np. @SpringBootTest).
Organizacja testów	Warto wydzielać klasy testowe zgodnie z warstwami aplikacji (np. controller, service).

7 DWA STYLE ORGANIZACJI

WYBIERZ STYL PASUJĄCY DO PROJEKTU

Package by layer
Organizacja według warstw.

```
com.example.app
├── controller
├── service
├── repository
├── domain
└── config
```

Package by feature
Organizacja według funkcjonalności.

```
com.example.app
├── users
│   ├── controller
│   ├── service
│   └── repository
├── orders
└── payments
```

W większych projektach często lepiej skaluje się podejście package by feature.

8 CO POWINNO TRAFIĆ DO KONFIGURACJI?

NAJWAŻNIEJSZE ELEMENTY

- **Właściwości środowisk** np. profile (dev, test, prod).
- **Dane połączeń** np. baza danych, Redis, zewnętrzne API.
- **Flagi funkcjonalne** np. włączanie/wyłączanie funkcji.
- **Ustawienia bezpieczeństwa** np. klucze, tokeny, konfiguracja OAuth2.
- **Konfiguracje klientów zewnętrznych** np. URL-e, timeouts, dane dostępowe.

9 W PRAKTYCE

DOBRE NAWYKI OD PIERWSZEGO DNIA

- Trzymaj kod biznesowy z dala od warstwy kontrolerów.
- Nie mieszaj konfiguracji technicznej z logiką domenową.
- Dbaj o spójne nazewnictwo pakietów.

Przykładowa struktura projektu

```
demo-app/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/example/demo/
│   │   │       └── DemoApplication.java
│   │   └── resources/
│   │       ├── application.yml
│   │       ├── static/
│   │       └── templates/
│   └── test/
│       ├── java/
│       └── resources/
└── pom.xml (lub build.gradle)
```

Przejrzysta struktura to lepszy kod!



10 UWAGA

- Zbyt płytka lub chaotyczna struktura szybko utrudnia rozwój projektu.
- Umieszczenie klasy startowej w złym pakiecie może spowodować, że część beanów nie zostanie wykryta.

11 SZYBKIE PODSUMOWANIE

Dobra struktura projektu = szybszy rozwój. Łatwiejsze testy i mniej chaosu.

Poświęć chwilę na przemyślaną strukturę — to się opłaca!

MAPA 04/100

JAK URUCHAMIA SIĘ APLIKACJA SPRING BOOT?

Od `main()` do gotowej aplikacji — krok po kroku

Każda wielka aplikacja zaczyna się od prostego kroku!

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 PUNKT STARTOWY

OD MAIN() DO CAŁEGO PROCESU

Aplikacja zwykle zaczyna się w metodzie `main()`.

Wywołanie `SpringApplication.run(...)` uruchamia cały proces startu.

```
1  import org.springframework.
   boot.autoconfigurre.
   SpringBootApplication;
2  @SpringBootApplication
3  public class DemoApplication {
4    public static void main(String[] args) {
       SpringApplication.run(
         DemoApplication.class, args);
     }
5  }
```

2 PRZYGOTOWANIE ŚRODOWISKA

TWORZENIE ENVIRONMENT

Tworzony jest `Environment`.

Ładowane są właściwości z różnych źródeł:

- plików `application.properties` i `application.yml`
- zmiennych środowiskowych
- argumentów startowych
- innych źródeł (np. chmura, system)

Dzięki temu aplikacja może być łatwo konfigurowana w różnych środowiskach (dev, test, prod).

3 TWORZENIE APPLICATIONCONTEXT

KONTENER SPRINGA

Powstaje kontener Springa (`ApplicationContext`).

To on przechowuje beany i zarządza ich cyklem życia.

Spring Boot domyślnie tworzy `AnnotationConfigServletWebServer-ApplicationContext` dla aplikacji webowej.

ApplicationContext to serce aplikacji — łączy konfigurację, beany i mechanizmy Springa w jedną całość.

4 COMPONENT SCANNING

WYSZUKIWANIE KOMPONENTÓW

Spring wyszukuje klasy oznaczone adnotacjami, takimi jak:

`@Component` `@Service` `@Repository` `@Controller`

Dzięki temu klasy są automatycznie rejestrowane w kontenerze jako beany.

5 AUTO-CONFIGURATION

INTELIGENTNA KONFIGURACJA

Na podstawie classpathu i warunków Boot tworzy domyślną konfigurację.

Przykład: jeśli wykryje Spring MVC, może skonfigurować obsługę webową (DispatcherServlet, mapowanie, konwertery itp.).

Auto-konfiguracja znacznie upraszcza start aplikacji — nie musisz konfigurować wszystkiego ręcznie.

6 TWORZENIE BEANÓW

INICJALIZACJA I ZALEŻNOŚCI

Rozwiązywane są zależności między beanami (Dependency Injection).

Beany są inicjalizowane i przygotowywane do pracy (np. wywoływane są metody @PostConstruct).

Kontener zarządza cyklem życia beanów — od utworzenia, przez inicjalizację, aż po zniszczenie.

7 START WARSTWY WEB

URUCHOMIENIE WBUDOWANEGO SERWERA

Gdy projekt jest aplikacją webową, uruchamiany jest wbudowany serwer (np. Tomcat, Jetty, Undertow).

Dopiero wtedy aplikacja zaczyna odbierać żądania HTTP.

W tym momencie aplikacja jest gotowa do przyjmowania ruchu zewnętrznego.

8 EVENTY STARTOWE

KOLEJNE ETAPY ROZRUCHU

- **ApplicationStartingEvent** — Najwcześniejszy event — początek startu.
- **ApplicationEnvironmentPreparedEvent** — Środowisko zostało przygotowane.
- **ApplicationPreparedEvent** — Kontekst jest już przygotowany, ale beany jeszcze nie zostały utworzone.
- **ApplicationStartedEvent** — Kontekst został odświeżony, aplikacja wystartowała.
- **ApplicationReadyEvent** — Aplikacja jest w pełni gotowa do pracy.

Te eventy pozwalają reagować na kolejne etapy rozruchu aplikacji (np. do własnej logiki startowej).

9 RUNNERY

WYKONANIE PO STARCIE

`CommandLineRunner` i `ApplicationRunner` wykonują się po starcie kontekstu.

Służą np. do inicjalizacji danych, wykonywania prostych zadań startowych lub integracji z innymi systemami.

Runnery są dobrym miejscem na prostą logikę startową, która powinna wykonać się tylko raz po uruchomieniu aplikacji.

10 W PRAKTYCE

CO WPŁYWA NA CZAS STARTU

Im więcej zależności i auto-konfiguracji, tym dłuższy może być czas startu.

W logach startowych często widać, co zostało skonfigurowane (np. auto-konfiguracje, beany, serwery).

11 UWAGA

URUCHOMIENIE ≠ GOTOWOŚĆ

To, że aplikacja się uruchamia, nie oznacza jeszcze, że jest gotowa do ruchu produkcyjnego.

Kluczowe są też:

- health checks
- migracje bazy danych
- konfiguracja
- testy

12 SZYBKIE PODSUMOWANIE

CAŁY PROCES W JEDNYM CIĄGU



Od `main()` do gotowej aplikacji — Spring Boot wykonuje wiele kroków za Ciebie, aby uruchomić nowoczesną aplikację Javy.

MAPA 05/100

@SPRINGBOOTAPPLICATION BEZ TAJEMNIC

Jedna adnotacja, trzy ważne mechanizmy

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST?

GŁÓWNA ADNOTACJA KLASY STARTOWEJ

- Główna adnotacja klasy startowej w aplikacjach Spring Boot.
- Łączy w sobie kilka kluczowych mechanizmów konfiguracji.
- Upraszcza start projektu i eliminuje potrzebę wielu ręcznych ustawień.

2 @SPRINGBOOT-CONFIGURATION

SPECJALIZACJA @CONFIGURATION

- Specjalizacja @Configuration.
- Pozwala definiować beany metodami @Bean.
- Oznacza, że klasa bierze udział w konfiguracji aplikacji.

3 @ENABLEAUTO-CONFIGURATION

AKTYWUJE AUTO-KONFIGURACJĘ

- Aktywuje auto-konfigurację Spring Boot.
- Boot dobiera konfigurację na podstawie classpathu, właściwości i istniejących beanów.
- Pozwala szybko uruchomić aplikację z gotową konfiguracją wielu modułów (np. baza danych, web, security, itd.).

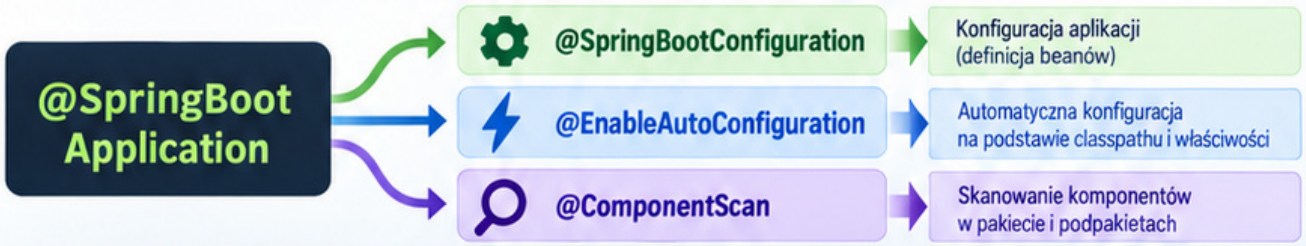
4 @COMPONENTSCAN

WYSZUKUJE KOMPONENTY

- Wyszukuje komponenty w pakiecie klasy startowej i w jego podpakietach.
- Dzięki temu wykrywane są @Service, @Repository, @Controller i inne komponenty.
- Umożliwia automatyczne rejestrowanie beanów i budowanie całego kontekstu aplikacji.

5 JAK TO DZIAŁA? – JEDNA ADNOTACJA, TRZY MECHANIZMY

ADNOTACJA @SPRINGBOOTAPPLICATION ŁĄCZY W SOBIE TRZY KLUCZOWE ELEMENTY:



Trzy mechanizmy. Jedna adnotacja. Wielkie możliwości!

6 DLACZEGO LOKALIZACJA KLASY MA ZNACZENIE?

- Jeśli klasa startowa znajduje się zbyt nisko w strukturze pakietów, część komponentów może nie zostać zeskanowana.
- Najbezpieczniej umieścić ją wysoko w głównym pakiecie aplikacji.
- Dzięki temu wszystkie podpakiety są objęte skanowaniem.

7 KIEDY DODAĆ KONFIGURACJĘ RĘCZNIE?

- Gdy potrzebujesz własnych beanów.
- Gdy konfigurujesz zewnętrznych klientów lub biblioteki.
- Gdy chcesz nadpisać zachowanie domyślne.
- Najczęściej robisz to w osobnych klasach z adnotacją @Configuration.

8 WYKLUCZANIE AUTO-KONFIGURACJI

- Możliwe przez parametr **exclude**.
- Przydaje się, gdy jakaś auto-konfiguracja jest zbędna lub koliduje z architekturą projektu.
- Możesz wykluczyć jeden lub wiele modułów.

```
@SpringBootApplication(
  exclude = { DataSourceAutoConfiguration.class }
)
```

9 W PRAKTYCE

- To najczęściej jedyna adnotacja potrzebna w klasie startowej.
- Większość codziennej konfiguracji odbywa się w innych klasach lub plikach właściwości.
- Dzięki temu możesz szybko uruchomić działającą aplikację i skupić się na logice biznesowej.

Mniej konfiguracji. Więcej działania!

10 PRZYKŁAD

MINIMALNA KLASA STARTOWA APLIKACJI

```
1  @SpringBootApplication
2  public class App {
3    public static void main(String[] args) {
4      SpringApplication.run(App.class, args);
5    }
6  }
```

11 SZYBKIE PODSUMOWANIE

- **@SpringBootApplication** = konfiguracja + auto-konfiguracja + skanowanie komponentów.
- Łączy trzy potężne mechanizmy w jednej, prostej adnotacji.
- Umożliwia szybki start i wygodną konfigurację aplikacji Spring Boot.
- W większości projektów to jedyny wymagany element w klasie startowej.

! UWAGA

- @SpringBootApplication nie jest „magiczną naprawą wszystkiego".
- Jeśli komponenty nie są skanowane albo beany się konfliktują, nadal trzeba rozumieć mechanizmy Springa.
- W przypadku problemów warto sprawdzić logi, konfigurację i strukturę pakietów.

MAPA 06/100

IOC, DI I APPLICATIONCONTEXT

Jak Spring zarządza obiektami i ich zależnościami

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 IOC — INVERSION OF CONTROL

Zamiast samodzielnie tworzyć obiekty, oddajesz kontrolę kontenerowi Springa.

To **framework** zarządza cyklem życia komponentów.

Spring przejmuje odpowiedzialność za tworzenie, konfigurowanie i zarządzanie obiektami.

2 DI — DEPENDENCY INJECTION

Zależności są dostarczane z zewnątrz.

Dzięki temu klasy są luźniej powiązane i łatwiejsze do testowania.

Komponent nie tworzy swoich zależności – otrzymuje je od kontenera Springa.

3 APPLICATIONCONTEXT

Główny kontener Springa.

Przechowuje beany, rozwiązuje zależności i udostępnia konfigurację.

Umożliwia dostęp do beanów, obsługuje profile, właściwości i zdarzenia aplikacji.

Najczęściej używamy AnnotationConfigApplicationContext lub, w aplikacjach webowych, kontenera uruchamianego przez Spring Boot.

4 JAK POWSTAJE BEAN?

Spring wykrywa komponent lub metodę @Bean. → Tworzy instancję obiektu. → Wstrzykuje zależności. → Inicjalizuje obiekt i udostępnia go innym komponentom

5 RODZAJE WSTRZYKIWANIA

Typ wstrzykiwania	Opis
Constructor injection	Najczęściej najlepszy wybór; ułatwia testy i immutabiilty.
Setter injection	Przydatne dla zależności opcjonalnych.
Field injection	Wygodne, ale zwykle niezalecane.

6 PRZYKŁAD PRZEPŁYWU

Controller (np. UserController) → **Service** (np. UserService) → **Repository** (np. UserRepository)

Każda warstwa otrzymuje zależność przez konstruktor, co sprawia, że kod jest jasny, testowalny i łatwy w utrzymaniu.

7 DLACZEGO TO WAŻNE?

- Lepsza testowalność.
- Mniejsza liczba zależności ukrytych w kodzie.
- Łatwiejsza wymiana implementacji (np. mock w testach, inna implementacja).

8 NAJCZĘSTSZE PUŁAPKI

- Cykliczne zależności.
- Zbyt wiele odpowiedzialności w jednej klasie.
- Nadużywanie @Autowired w polach.

9 W PRAKTYCE

- Preferuj wstrzykiwanie przez konstruktor.
- Zadbaj, aby klasy miały pojedynczą odpowiedzialność (SRP).
- Traktuj kontener jako narzędzie organizujące współpracę klas.

10 UWAGA

- IoC i DI nie oznaczają, że każda klasa musi być beanem Springa.
- Pomocnicze obiekty domenowe często nie powinny trafiać do kontenera (np. proste klasy wartości, DTO, encje).

11 SZYBKIE PODSUMOWANIE

- IoC oddaje kontrolę frameworkowi.
- DI dostarcza zależności.
- **ApplicationContext** spina to wszystko w całość.

</> PRZYKŁAD: WSTRZYKIWANIE PRZEZ KONSTRUKTOR

```java
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getUsers() {
        return userService.findAll();
    }
}
```

```java
@Service
public class UserService {

    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public List<User> findAll() {
        return userRepository.findAll();
    }
}
```



MAPA 07/100

CYKL ŻYCIA BEANÓW I SCOPE

Od utworzenia obiektu do jego zniszczenia

Poznaj, jak Spring tworzy, zarządza i niszczy beany oraz jakie są dostępne zakresy.

Ten sam ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST BEAN?

OBIEKT ZARZĄDZANY PRZEZ SPRINGA

Bean to obiekt zarządzany przez kontener Springa (ApplicationContext).

Może pochodzić z component scan (np. adnotacje @Component, @Service, @Repository) albo z metody @Bean w klasie konfiguracyjnej @Configuration.

2 GŁÓWNE ETAPY CYKLU ŻYCIA

OD UTWORZENIA DO ZNISZCZENIA

Instancja	Wstrzyknięcie zależności	Inicjalizacja	Użycie	Zniszczenie
Utworzenie obiektu beana.	Spring wstrzykuje wymagane zależności.	Wywołanie @PostConstruct i innych callbacków.	Bean jest dostępny i wykorzystywany przez aplikację.	Wywołanie @PreDestroy (dla odpowiednich beanów).

3 INICJALIZACJA

@POSTCONSTRUCT I CALLBACKI

Po utworzeniu obiektu Spring może wykonać metodę @PostConstruct.

Możliwe są też inne callbacki inicjalizacyjne z konfiguracji, np. poprzez interfejs afterPropertiesSet() (InitializingBean) lub metodę initMethod w @Bean.

4 ZAKOŃCZENIE

@PREDESTROY I SPRZĄTANIE

Dla odpowiednich beanów może zostać wywołana metoda @PreDestroy.

Jest to ważne np. przy zamykanie zasobów, takich jak połączenia z bazą danych, pliki czy wątki.

5 SCOPE: SINGLETON

DOMYŚLNY ZAKRES

- Domyślny zakres w Springu.
- Jedna współdzielona instancja w ApplicationContext.
- Najczęstszy wybór dla serwisów, repozytoriów i większości komponentów aplikacji.

6 SCOPE: PROTOTYPE

NOWA INSTANCJA ZA KAŻDYM RAZEM

- Nowa instancja przy każdym pobraniu z kontekstu.
- Przydatne rzadziej, gdy obiekt ma być tworzony wielokrotnie, np. obiekty stanu, generatory, obiekty pomocnicze.

7 SCOPE WEBOWE

ZAKRESY W APLIKACJACH WEBOWYCH

request	Jedna instancja na żądanie HTTP.
session	Jedna instancja na sesję użytkownika.
application	Jedna instancja na poziomie aplikacji webowej (ServletContext).

8 LAZY I EAGER

MOMENT TWORZENIA BEANÓW

Beany singleton zwykle są tworzone przy starcie kontekstu (eager initialization).

@Lazy pozwala odroczyć inicjalizację beana do momentu jego pierwszego użycia.

9 PUŁAPKI

NA CO UWAŻAĆ?

- Wstrzykiwanie beana prototype do singletona może prowadzić do nieoczekiwanych efektów (np. współdzielenie tej samej instancji).
- Stan współdzielony w singletonach bywa źródłem błędów współbieżności (np. problemy z wielowątkowością).

10 W PRAKTYCE

SPRAWDZONE WSKAZÓWKI

- Większość komponentów aplikacji to singletons.
- Scope request przydaje się dla danych zależnych od konkretnego żądania.
- @PostConstruct stosuj oszczędnie — tylko tam, gdzie naprawdę trzeba.

</> PRZYKŁAD: BEAN Z @POSTCONSTRUCT i @PREDESTROY

```
@Component
public class DemoService {
    private static final Logger log =
            LoggerFactory.getLogger(DemoService.class);

    @PostConstruct
    public void init() {
        log.info("DemoService - inicjalizacja beana");
    }

    @PreDestroy
    public void destroy() {
        log.info("DemoService - niszczenie beana");
    }
}
```

11 SZYBKIE PODSUMOWANIE

- Spring zarządza nie tylko tworzeniem beanów, ale też ich życiem i zakresem.

Lepsze aplikacje tworzy się z dobrych narzędzi!



MAPA 08/100

COMPONENT SCANNING I STEREOTYPY

Jak Spring odnajduje klasy, które mają stać się beanami

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 COMPONENT SCANNING

MECHANIZM WYSZUKIWANIA KLAS

Mechanizm wyszukiwania klas oznaczonych adnotacjami komponentów.

Uruchamia się zwykle od pakietu klasy startowej (np. z **@SpringBootApplication**).

Spring skanuje wszystkie klasy w tym pakiecie i jego podpakietach.

Przekształca znalezione klasy w **beany** i rejestruje je w **ApplicationContext**.

2 NAJWAŻNIEJSZE STEREOTYPY

GOTOWE ADNOTACJE DLA RÓŻNYCH RÓL

@Component
Ogólny komponent.
Podstawowy stereotyp.

@Service
Logika biznesowa.
Najczęściej dla warstwy usług.

@Repository
Dostęp do danych.
Może wspierać tłumaczenie wyjątków (np. na wyjątki Springa).

@Controller / @RestController
Warstwa web.
Obsługa żądań HTTP i zwracanie odpowiedzi.

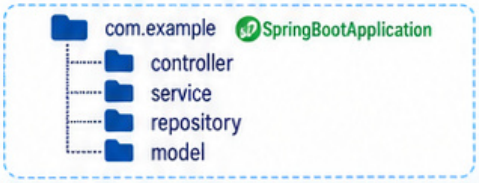
3 GDZIE SPRING SZUKA KLAS?

LOKALIZACJA MA ZNACZENIE

W **pakiecie klasy startowej i jego podpakietach.**

Dlatego lokalizacja klasy z **@SpringBootApplication** ma znaczenie.

Najlepiej umieścić klasę startową w głównym, nadrzędnym pakiecie aplikacji, aby objąć skanowaniem wszystkie istotne klasy.



4 KIEDY UŻYĆ @COMPONENT?

DLA KLAS POMOCNICZYCH LUB TECHNICZNYCH

Dla klas pomocniczych lub technicznych (np. narzędzia, konwertery, walidatory, schedulery).

Gdy żadna bardziej szczegółowa adnotacja (np. @Service, @Repository, @Controller) nie opisuje lepiej roli klasy.

To nadal pełnoprawny komponent zarządzany przez Springa.

5 @RESTCONTROLLER VS @CONTROLLER

DWA SPOSOBY OBSŁUGI ŻĄDAŃ HTTP

@RestController

- Zwraca zwykle dane (np. JSON).
- Łączy **@Controller** i **@ResponseBody**.
- Stosowany w REST API.
- Nie wymaga dodatkowych adnotacji @ResponseBody.

@Controller

- Częściej współpracuje z widokami serwerowymi (np. Thymeleaf, JSP).
- Zwraca nazwę widoku.
- Wymaga **@ResponseBody**, jeśli ma zwracać dane (JSON, XML).

6 CO, JEŚLI KLASA JEST POZA SKANOWANIEM?

SPOSOBY NA DODANIE BEANA

Użyj @Import
Zaimportuj konfigurację lub klasę z adnotacją komponentu.

Zadeklaruj obiekt metodą @Bean
Utwórz beana w klasie konfiguracyjnej (@Configuration).

Rozszerz zakres skanowania
Użyj opcji scanBasePackages w @SpringBootApplication lub @ComponentScan.

7 NAJCZĘSTSZE PROBLEMY

NA CO UWAŻAĆ?

Klasa poza odpowiednim pakietem.

Duplikaty beanów tego samego typu (np. dwie implementacje interfejsu).

Niewłaściwa odpowiedzialność klasy względem zastosowanego stereotypu.

8 W PRAKTYCE

DOBRE NAWYKI

Stereotyp powinien komunikować rolę klasy.

To poprawia czytelność projektu i ułatwia orientację w kodzie.

Konsekwentne stosowanie stereotypów pomaga zespołowi lepiej zrozumieć strukturę aplikacji.

</> PRZYKŁAD W KODZIE

Prosty przykład z @Service i @RestController.

```java
@Service
public class GreetingService {
    public String powiedzCzesc(String imie) {
        return "Cześć, " + imie + "!";
    }
}
@RestController
@RequestMapping("/api")
public class GreetingController {
    private final GreetingService service;
    public GreetingController(GreetingService service) {
        this.service = service;
    }
    @GetMapping("/powitanie/{imie}")
    public String powitanie(@PathVariable String imie) {
        return service.powiedzCzesc(imie);
    }
}
```

9 UWAGA

WARTO PAMIĘTAĆ

- Nie każde oznaczenie klas @Component jest dobrym pomysłem.
- Dla obiektów z bibliotek zewnętrznych często lepiej użyć konfiguracji @Bean.

10 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- **Component scanning** odnajduje komponenty.
- **Stereotypy** mówią, jaką rolę pełni klasa.
- Razem pomagają budować spójny **ApplicationContext**.

Lepsze aplikacje tworzy się z dobrych narzędzi i jeszcze lepszych pomysłów!



MAPA 09/100

@CONFIGURATION I @BEAN

Kiedy warto jawnie definiować obiekty w kontenerze Springa

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 @CONFIGURATION

OZNACZA KLASĘ KONFIGURACYJNĄ

@Configuration
Oznacza klasę konfiguracyjną.

Zawiera metody tworzące i konfigurujące beany.

Najczęściej umieszczana w osobnym pakiecie np. config.

2 @BEAN

OZNACZA METODĘ TWORZĄCĄ BEAN

@Bean
Oznacza metodę, której wynik ma trafić do ApplicationContext.

Spring zarządza dalej tym obiektem jak każdym innym beanem.

Domyślnie bean ma nazwę metody, ale możesz ją zmienić w adnotacji: @Bean("mojBean")

3 KIEDY TO SIĘ PRZYDAJE?

TYPOWE SYTUACJE

Gdy konfigurujesz biblioteki zewnętrzne.

Gdy chcesz jawnie zbudować klienta HTTP, zegar, mapper lub strategię.

Gdy obiekt nie powinien być tworzony przez component scan.

4 METODA @BEAN JAKO FABRYKA

SPRING DBA O ZALEŻNOŚCI

Metoda może przyjmować argumenty.

Spring sam wstrzykuje do niej potrzebne zależności.

Dzięki temu możesz łatwo składać bardziej złożone obiekty z innych beanów.

5 PORÓWNANIE

DWA SPOSOBY REJESTRACJI BEANÓW

Component scan	@Bean
Wygodny dla własnych klas aplikacji.	Wygodny dla jawnej konfiguracji i klas zewnętrznych.
Wykrywa adnotacje np. @Component, @Service, @Repository.	Daje pełną kontrolę nad tworzeniem i konfiguracją obiektu.

6 DODATKOWE MOŻLIWOŚCI

PRZYDATNE ADNOTACJE

@Profile
Różne beany dla różnych środowisk (np. dev, test, prod).

@ConditionalOnProperty
Bean aktywny tylko przy określonej właściwości w pliku konfiguracyjnym.

@Primary
Domyślny wybór przy wielu implementacjach tego samego typu.

7 PRZYKŁAD ZASTOSOWANIA

NAJCZĘSTSZE SCENARIUSZE

Konfiguracja klienta REST
Np. RestTemplate, WebClient.

Konfiguracja Clock
Np. Clock.systemUTC() w aplikacji.

Konfiguracja ObjectMapper
Dostosowanie mapowania JSON (np. moduły, formaty).

Konfiguracja klienta do systemu zewnętrznego
Np. klient do płatności, mailingu, API partnera.

8 W PRAKTYCE

DOBRE ZASADY

Używaj @Bean, gdy chcesz wyraźnie pokazać, jak obiekt powstaje.

To dobre miejsce na konfigurację techniczną, a nie logikę biznesową.

9 UWAGA

NA CO UWAŻAĆ

Zbyt duże klasy konfiguracyjne szybko stają się nieczytelne.

Warto dzielić konfigurację tematycznie: web, security, clients, data.

10 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

@Configuration porządkuje konfigurację.

@Bean pozwala jawnie tworzyć obiekty.

To ważne uzupełnienie component scanu.

```
@Configuration
class AppConfig {
  @Bean
  Clock systemClock() {
    return Clock.systemUTC();
  }
}
```

Prosta konfiguracja. Wielkie możliwości!



MAPA 10/100

STARTERY, MAVEN, GRADLE I ZARZĄDZANIE ZALEŻNOŚCIAMI

Jak mądrze dobierać zależności i utrzymywać porządek w projekcie

Jeden ekosystem. Wiele możliwości. Większe projekty.

MODUŁ 01 • FUNDAMENTY I ARCHITEKTURA

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST STARTER?

GOTOWY ZESTAW ZALEŻNOŚCI

- Gotowy zestaw zależności dla konkretnego typu aplikacji.
- Upraszcza konfigurację i ogranicza ręczne dobieranie bibliotek.
- Startery zawierają sprawdzone, kompatybilne biblioteki.

2 NAJCZĘSTSZE STARTERY

POPULARNE STARTERY SPRING BOOT

- **spring-boot-starter-web** Tworzenie aplikacji webowych i REST API.
- **spring-boot-starter-data-jpa** Dostęp do danych, JPA, Hibernate.
- **spring-boot-starter-validation** Walidacja danych (Bean Validation).
- **spring-boot-starter-security** Zabezpieczenia i uwierzytelnianie.
- **spring-boot-starter-actuator** Monitoring i metryki aplikacji.
- **spring-boot-starter-test** Testy jednostkowe i integracyjne.

3 MAVEN I GRADLE

DWA POPULARNE NARZĘDZIA BUILDOWE

Maven

- Deklaratywny.
- Oparty na XML.
- Prosty i dojrzały.
- Duża społeczność.

Gradle

- Elastyczny.
- Oparty na DSL (Groovy lub Kotlin).
- Łatwy do rozbudowy.
- Świetny do większych projektów.

Oba narzędzia są dobrze wspierane przez Spring Boot i mają pełną integrację z ekosystemem.

4 DEPENDENCY MANAGEMENT

AUTOMATYCZNE ZARZĄDZANIE WERSJAMI

- Spring Boot zarządza zgodnymi wersjami wielu bibliotek.
- Dzięki temu zwykle nie trzeba ręcznie wpisywać wersji dla wielu zależności Springowych.
- Zmniejsza to ryzyko konfliktów wersji.

5 DLACZEGO TO WAŻNE?

KORZYŚCI W PRAKTYCE

- Mniej przypadkowych niezgodności.
- Szybsze tworzenie projektu.
- Lepsza przewidywalność działania aplikacji.

6 JAK WYBIERAĆ ZALEŻNOŚCI?

DOBRE PRAKTYKI

- Dodawaj tylko to, czego naprawdę potrzebujesz.
- Rozumiej, co wnosi każdy starter.
- Unikaj instalowania dużej liczby bibliotek „na zapas".
- Regularnie przeglądaj i usuwaj nieużywane zależności.

7 W PRAKTYCE

PRZYKŁADY KONFIGURACJI PROJEKTÓW

REST API	Web + Validation + Actuator.
Aplikacja z bazą danych	Data JPA + driver + Flyway.
Aplikacja zabezpieczona	Security.
Testy	Starter Test, a często także Testcontainers.

8 PUŁAPKI

CZEGO UNIKAĆ?

- Nadmiar zależności spowalnia analizę projektu i może wydłużać start aplikacji.
- Ręczne nadpisywanie wersji bez potrzeby zwiększa ryzyko problemów.

9 MAVEN VS GRADLE — WYBÓR

CO JEST NAJWAŻNIEJSZE?

- Wybierz narzędzie, które zespół zna najlepiej.
- Spójność w projekcie bywa ważniejsza niż sama technologia builda.
- Oba narzędzia świetnie współpracują ze Spring Boot.

MAVEN — PRZYKŁADOWA ZALEŻNOŚĆ — pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

GRADLE — PRZYKŁADOWA ZALEŻNOŚĆ — build.gradle

```
// Kotlin DSL (build.gradle.kts) lub Groovy DSL
implementation("org.springframework.boot:spring-boot-starter-web")
```

10 SZYBKIE PODSUMOWANIE

- Startery upraszczają dobór bibliotek.
- Boot pomaga utrzymać kompatybilne wersje.
- Dobre zarządzanie zależnościami to mniej niespodzianek w produkcji.

Prosty projekt dziś — stabilna produkcja jutro!



MAPA 11/100

JAK DZIAŁA AUTO-KONFIGURATION?

Jak Spring Boot sam składa sensowną konfigurację aplikacji

Mądra automatyzacja. Mniej konfiguracji. Większe możliwości!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST AUTO-KONFIGURACJA?

INTELIGENTNA KONFIGURACJA SPRING BOOTA

Spring Boot automatycznie konfiguruje aplikację na podstawie dostępnych bibliotek, ustawień i istniejących beanów.

Celem jest uruchomienie sensownej konfiguracji „od razu", bez ręcznego definiowania beanów.

Działa zgodnie z zasadą **„Convention over Configuration"** – rozsidne domyślne ustawienia.

Ty skupiasz się na logice biznesowej, a nie na konfiguracji infrastruktury.

2 SKĄD SIĘ BIERZE?

BIBLIOTEKI I STARTERY

spring-boot-autoconfigure
Zawiera setki klas auto-konfiguracyjnych dla różnych technologii (web, dane, cache, itp.).

Startery Spring Boot
Np. spring-boot-starter-web, spring-boot-starter-data-jpa. Wciągają odpowiednie zależności i aktywują auto-konfigurację.

Plik META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.im
Lista klas auto-konfiguracyjnych ładowanych przez Spring Boot.

3 TRZY SYGNAŁY DO AKTYWACJI

KIEDY AUTO-KONFIGURACJA SIĘ WŁĄCZA?

1. CLASSPATH
Czy na classpathie są potrzebne biblioteki? (np. spring-web, H2, Redis)

2. PROPERTIES
Czy w konfiguracji są odpowiednie właściwości? (np. spring.datasource.*)

3. ISTNIEJĄCE BEANY
Czy nie zdefiniowano już własnych beanów? (własna konfiguracja ma pierwszeństwo)

Auto-konfiguracja uruchamia się tylko gdy warunki są spełnione.

4 JAK TO DZIAŁA? – PRZEPŁYW

OD STARTERA DO GOTOWYCH BEANÓW



Dodajesz zależność (np. starter-web)

Spring Boot znajduje odpowiednie klasy

Sprawdzane są warunki aktywacji

Rejestrowane są beany w kontekście

5 WARUNKI URUCHOMIENIA

PRZYKŁADY NAJCZĘŚCIEJ UŻYWANYCH WARUNKÓW

@ConditionalOnClass	Uruchom, gdy dana klasa jest na classpathie.
@ConditionalOnMissingBean	Uruchom, gdy nie ma jeszcze beana tego typu.
@ConditionalOnProperty	Uruchom, gdy ustawiono określoną właściwość.
@ConditionalOnWebApplication	Uruchom, gdy to aplikacja webowa (servlet lub reactive).

6 PRZYKŁAD: APLIKACJA WEB

CO KONFIGURUJE SPRING BOOT?

Spring MVC
Konfiguracja DispatcherServlet, obsługa requestów i odpowiedzi.

Konwertery
Automatycznie konfiguruje konwertery JSON (Jackson), obsługę błędów, static resources.

Gotowe ustawienia
Wystarczy dodać spring-boot-starter-web, a aplikacja webowa działa od razu!

7 PRZYKŁAD: DANE (JDBC)

AUTOMATYCZNA KONFIGURACJA DOSTĘPU DO DANYCH

DataSource
Tworzy i konfiguruje DataSource (np. HikariCP) na podstawie właściwości spring.datasource.*

JdbcTemplate
Automatycznie dostępny, gotowy do użycia.

Obsługa baz danych
Działa dla H2, MySQL, PostgreSQL i innych – wystarczy odpowiedni driver na classpathie.

8 KORZYŚCI

DLACZEGO WARTO?

- Szybszy start projektu.
- Mniej kodu konfiguracyjnego.
- Rozsądne, sprawdzone domyślne ustawienia.
- Łatwa rozbudowa i nadpisanie konfiguracji.
- Działa dla wielu technologii (web, dane, cache, messaging, itp.).
- Pozwala skupić się na logice biznesowej.

9 CZEGO TO NIE ROBI?

WAŻNE OGRANICZENIA

- Nie zgaduje twojej logiki biznesowej.
- Nie tworzy wszystkiego – tylko to, co ma sens w danym kontekście.
- Nie zastępuje ręcznej konfiguracji – możesz (i czasem powinieneś) zdefiniować własne beany.

10 W PRAKTYCE

PROSTY PRZYKŁAD

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```



Jeden starter, a ile konfiguracji dzieje się automatycznie!

11 SZYBKIE PODSUMOWANIE

AUTO-KONFIGURACJA TO:

- Analiza classpath, properties i beanów.
- Setki klas w spring-boot-autoconfigure.
- Warunki decydują, co się uruchomi.
- Startery aktywują odpowiednią konfigurację.
- Daje gotowe beany (web, dane, cache, ...).
- Możesz łatwo nadpisać domyślne ustawienia.
- Mniej konfiguracji, więcej możliwości!

MAPA 12/100

RODZINA @CONDITIONAL

Jak Spring Boot decyduje, kiedy utworzyć beana lub włączyć konfigurację

Inteligentne warunki. Mądrzejsze aplikacje.

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO ISTNIEJE @CONDITIONAL?

KONTROLA I ELASTYCZNOŚĆ

- Pozwala włączać lub wyłączać konfiguracje i beany w zależności od warunków.
- Umożliwia tworzenie uniwersalnych, modułowych i wielokontekstowych aplikacji.
- Jest podstawą mechanizmu auto-konfiguracji w Spring Boot.
- Pozwala dostosować aplikację do środowiska, klas na classpath czy ustawień właściwości.

2 JAK TO DZIAŁA?

SPRING OCENIA WARUNKI

- Podczas uruchamiania aplikacji Spring analizuje warunki z adnotacji @Conditional.
- Jeśli warunek jest spełniony, tworzony jest bean lub wczytywana jest konfiguracja.
- Jeśli warunek nie jest spełniony, bean lub konfiguracja są pomijane.
- Dzięki temu do kontekstu trafiają tylko te komponenty, które są potrzebne w danym środowisku.

To sprawia, że aplikacje Spring Boot są lekkie, elastyczne i łatwe do konfiguracji.

3 @CONDITIONALONCLASS

WARUNEK NA PODSTAWIE KLASY

Tworzy beana lub wczytuje konfigurację, jeśli na classpath znajduje się określona klasa.

```
@ConditionalOnClass(
   name = "com.zaxxer.hikari.HikariDataSource"
)
```

Przydatne do integracji opcjonalnych bibliotek (np. bazy danych, klienta HTTP).

4 @CONDITIONALONMISSINGBEAN

WARUNEK BRAKU BEANA

Tworzy beana tylko wtedy, gdy w kontekście nie istnieje bean danego typu lub o danej nazwie.

```
@ConditionalOnMissingBean
  (DataSource.class)
```

Pozwala dostarczyć domyślną konfigurację, którą użytkownik może łatwo nadpisać własnym beanem.

5 @CONDITIONALONBEAN

WARUNEK ISTNIENIA BEANA

Tworzy beana tylko wtedy, gdy w kontekście istnieje bean danego typu lub o danej nazwie.

```
@ConditionalOnBean
  (CacheManager.class)
```

Umożliwia tworzenie komponentów zależnych od innych beanów (np. konfiguracja cache).

6 @CONDITIONALONPROPERTY

WARUNEK NA PODSTAWIE WŁAŚCIWOŚCI

Tworzy beana, jeśli określona właściwość w pliku konfiguracyjnym ma zadaną wartość.

```
@ConditionalOnProperty(
   prefix = "app.mail",
   name = "enabled",
   havingValue = "true"
)
```

Pozwala łatwo włączać/wyłączać funkcjonalności za pomocą konfiguracji.

7 @CONDITIONALONWEBAPPLICATION / @CONDITIONALONNOTWEBAPPLICATION

WARUNEK TYPU APLIKACJI

- **@ConditionalOnWebApplication** – tworzy beana tylko w aplikacji web.
- **@ConditionalOnNotWebApplication** – tworzy beana tylko w aplikacji niebędącej web (np. CLI, batch).

Pozwala dostosować konfigurację do typu uruchamianej aplikacji.

8 @CONDITIONALONSINGLECANDIDATE

WARUNEK JEDNOZNACZNEGO KANDYDATA

Tworzy beana tylko wtedy, gdy w kontekście istnieje dokładnie jeden kandydat danego typu.

```
@ConditionalOnSingleCandidate
  (RestTemplate.class)
```

Przydatne, gdy integrujesz się z komponentami, które mogą mieć wiele implementacji.

9 PRAKTYCZNY PRZYKŁAD

WAJKUNKOWA KONFIGURACJA MAILA

Tworzymy konfigurację, która włącza beana MailService tylko, gdy włączona jest właściwość app.mail.enabled.

```
@Configuration
@ConditionalOnProperty(
   prefix = "app.mail",
   name = "enabled",
   havingValue = "true"
)
public class MailConfig {

   @Bean
   public MailService mailService() {
      return new SmtpMailService();
   }
```

10 PUŁAPKI

NA CO UWAŻAĆ?

- Warunki są sprawdzane tylko podczas uruchamiania aplikacji.
- Błędne nazwy klas lub właściwości powodują, że konfiguracja nie zadziała.
- Łatwo stworzyć nieczytelne, złożone warunki.
- Pamiętaj o domyślnych wartościach właściwości.
- Zbyt wiele warunków może utrudnić debugowanie.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- @Conditional pozwala tworzyć elastyczne aplikacje.
- Istnieje wiele wyspecjalizowanych adnotacji warunkowych.
- Warunki opierają się na klasach, beanach, właściwościach i typie aplikacji.
- To kluczowy element auto-konfiguracji w Spring Boot.
- Dzięki warunkom aplikacja jest lżejsza, modułowa i łatwiejsza w utrzymaniu.

</> Mniej zbędnych komponentów. Większe możliwości!



MAPA 13/100

BACK-OFF, WYKLUCZANIE I DIAGNOSTYKA

Jak przejąć kontrolę nad auto-konfiguracją i sprawdzić, co naprawdę się włączyło

Przejmij kontrolę nad swoją aplikacją!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST BACK-OFF?

SPRING BOOT JEST "GRZECZNY"

Back-off to mechanizm, dzięki któremu auto-konfiguracja wycofuje się, gdy znajdzie własną konfigurację.

Jeśli dostarczysz własny bean lub ustawienie, Spring Boot nie tworzy domyślnej wersji.

Dzięki temu możesz łatwo dostosować aplikację, zachowując wygodę auto-konfiguracji.

2 @ConditionalOnMissingBean

KLUCZ DO BACK-OFF

Wiele auto-konfiguracji używa adnotacji `@ConditionalOnMissingBean`.

Oznacza to: "utwórz ten bean, jeśli nie ma już innego takiego beana w kontekście".

Wystarczy dodać własny bean, aby Spring Boot automatycznie się wycofał z domyślnej konfiguracji.

Przykład warunku:

```
@ConditionalOnMissingBean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .build();
}
```

3 OVERRIDE WŁASNYM BEANEM

PODAJ WŁASNĄ IMPLEMENTACJĘ

Chcesz inną implementację? Zdefiniuj własny bean w swojej konfiguracji.

Spring Boot wykryje, że taki bean już istnieje i nie utworzy domyślnego.

To najprostszy i najbezpieczniejszy sposób dostosowania działania auto-konfiguracji.

Przykład:

```
@Configuration
public class MyConfig {
    @Bean
    public DataSource dataSource() {
        return new HikariDataSource();
    }
}
```

4 WYKLUCZANIE PRZEZ ANOTACJĘ

@SPRINGBOOTAPPLICATION

Możesz całkowicie wyłączyć wybraną auto-konfigurację za pomocą parametru `exclude` w adnotacji `@SpringBootApplication`.

Przykład:

```
@SpringBootApplication(
    exclude = {
        DataSourceAutoConfiguration.class
    }
)
public class MyApplication {
  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

5 SPRING.AUTOCONFIGURE.EXCLUDE

WYKLUCZANIE PRZEZ WŁAŚCIWOŚCI

To samo wykluczenie możesz ustawić w pliku konfiguracyjnym (applications.properties lub YAML).

Przydatne, gdy nie chcesz modyfikować kodu aplikacji.

Możesz wykluczyć wiele klas (auto-konfiguracji) oddzielonych przecinkami.

Przykład (application.properties):

```
spring.autoconfigure.exclude=
org.springframework.boot.autoconnfigure.
jdbc.DataSourceAutoConfiguration
```

6 DEBUG=true LUB --DEBUG

WIĘCEJ INFORMACJI NA STARCIE

Uruchom aplikację z właściwością `debug=true` lub parametrem `--debug`.

W logach zobaczysz szczegółowy raport warunków auto-konfiguracji (Condition Evaluation Report).

Łatwo sprawdzisz, które konfiguracje zostały włączone, a które pominięte.

Przykłady uruchomienia:

```
# w pliku application.properties
debug=true

# lub z linii poleceń
java -jar app.jar  --debug
```

7 CONDITION EVALUATION REPORT

SZCZEGÓŁOWA DIAGNOSTYKA

Raport pokazuje, dlaczego dana auto-konfiguracja została włączona lub wyłączona.

Zawiera spełnione i niespełnione warunki z adnotacji takich jak `@ConditionalOnClass`, `@ConditionalOnBean` itp.

To najlepsze źródło wiedzy, dlaczego coś się (nie) dzieje w Twojej aplikacji.

8 ACTUATOR /CONDITIONS

DIAGNOSTYKA W RUNTIME

Endpunkt `/actuator/conditions` pokazuje ten sam raport warunków w działającej aplikacji.

Wymaga włączenia Spring Actuator i odpowiedniej konfiguracji dostępu (np. tylko lokalnie).

Przydatne w środowiskach testowych i produkcyjnych – bez konieczności włączania trybu debug.

Przykład zapytania:

```
GET http://localhost:8080/actuator/conditions
```

9 KIEDY WARTO WYKLUCZAĆ?

PRAKTYCZNE SYTUACJE

- Gdy nie używasz danego modułu (np. JDBC, DataSource, Security).
- Gdy chcesz zmniejszyć rozmiar aplikacji i liczbę zależności.
- Gdy auto-konfiguracja powoduje konflikty lub niepożądane bean'y.
- Gdy masz własną, niestandardową konfigurację danego obszaru.
- Gdy chcesz szybciej uruchamiać aplikację (np. w testach).

10 PUŁAPKI

NA CO UWAŻAĆ?

- Wykluczenie może wyłączyć więcej niż oczekujesz (zależności pomiędzy konfiguracjami).
- Trudniej wtedy skorzystać z domyślnych usprawnień Spring Boota.
- Niewłaściwe wykluczenie może powodować błędy uruchomienia aplikacji.
- Zawsze sprawdzaj Condition Evaluation Report, aby zrozumieć skutki zmiany.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Back-off pozwala łatwo dostosować auto-konfigurację.
- Własny bean automatycznie wyłącza domyślny.
- Możesz wykluczać auto-konfiguracje przez adnotację lub właściwości.
- Używaj debug i Actuatora do diagnostyki.
- Wykluczaj świadomie i tylko to, czego nie potrzebujesz.
- Dobra diagnostyka = pełna kontrola nad aplikacją.

Sprawdzaj, rozumiej, dostosowuj, buduj lepsze aplikacje!



MAPA 14/100

EXTERNALIZED CONFIGURATION

Kolejność i priorytety źródeł konfiguracji w Spring Boot

Źródło o najwyższym priorytecie zawsze wygrywa!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO EXTERNALIZED CONFIGURATION?

KONFIGURACJA POZA KODEM

- Oddziela konfigurację od kodu aplikacji.
- Umożliwia różne ustawienia dla różnych środowisk (dev, test, prod).
- Ułatwia zmianę ustawień bez rekompilacji.
- Wspiera zasadę dwunastu czynników (12-factor app).

2 NAJCZĘSTSZE ŹRÓDŁA KONFIGURACJI

SKĄD SPRING BOOT CZYTA USTAWIENIA?

- **Pliki konfiguracyjne** (application.properties / .yml)
- **Profile specyficzne** (application-{profile}.yml)
- **Zmienne środowiskowe** (Environment Variables)
- **Właściwości systemowe** (Java System Properties)
- **Argumenty linii poleceń** (Command Line Arguments)
- **Zewnętrzne systemy** (np. Spring Cloud Config, Vault)

3 ZASADA NADPISYWANIA

TEN SAM KLUCZ, WIELE ŹRÓDEŁ

- Jeśli ten sam klucz występuje w wielu źródłach, używana jest wartość z wyższym priorytetem.
- Niższe źródła są nadpisywane przez wyższe.
- Dotyczy to wszystkich właściwości (np. server.port, spring.datasource.url).
- Pozwala łatwo dostosować aplikację do środowiska bez zmiany plików konfiguracyjnych.

4 UPROSZCZONA KOLEJNOŚĆ

OD NAJNIŻSZEGO DO NAJWYŻSZEGO PRIORYTETU

5	Argumenty linii poleceń
4	Właściwości systemowe
3	Zmienne środowiskowe
2	Pliki konfiguracyjne (Config Data)
1	Domyślne wartości (w kodzie aplikacji)

Wyższy priorytet ↑

↓ Niższy priorytet

5 CONFIG DATA – PLIKI KONFIGURACYJNE

application.properties / application.yml

- Podstawowe źródło konfiguracji.
- Obsługuje profile (np. application-dev.yml).
- Może być ładowany z różnych lokalizacji (classpath:, file: etc.).
- Obsługuje import dodatkowych plików (spring.config.import).

```yaml
# application.yml
server:
  port: 8080
spring:
  application:
    name: moja-aplikacja
```

6 ZMIENNE ŚRODOWISKOWE

ENVIRONMENT VARIABLES

- Używane często w środowiskach chmurowych i kontenerach.
- Nazwy w formacie UPPER_SNAKE_CASE (np. SERVER_PORT).
- Kropki zamieniamy na podkreślenia.
- Bardzo przydatne w Docker, Kubernetes, CI/CD.

```
# Zmienna środowiskowa
SERVER_PORT=9090
SPRING_PROFILES_ACTIVE=prod
```

7 WŁAŚCIWOŚCI SYSTEMOWE

JAVA SYSTEM PROPERTIES

- Przekazywane do JVM przez -D.
- Mają wysoki priorytet.
- Często używane w skryptach startowych i środowiskach serwerowych.

```
# Uruchomienie aplikacji
java -Dserver.port=7070 \
     -jar app.jar
```

8 ARGUMENTY STARTOWE

COMMAND LINE ARGUMENTS

- Najwyższy priorytet.
- Przekazywane po -- przy uruchamianiu aplikacji.
- Wygodne do szybkich zmian bez edycji plików.
- Nadpisują wszystkie inne źródła.

```
# Uruchomienie aplikacji
java -jar app.jar --server.port=6060 \
     --spring.profiles.active=prod
```

9 PRZYKŁAD KONFLIKTU

SERVER.PORT W RÓŻNYCH ŹRÓDŁACH

```
# 1. application.yml
server.port=8080
# 2. Zmienna środowiskowa
SERVER_PORT=9090
# 3. Właściwość systemowa
-Dserver.port=7070
# 4. Argument startowy
--server.port=6060
```

Wynik: aplikacja uruchomi się na porcie **6060**

10 W PRAKTYCE

SPRAWDZONE WSKAZÓWKI

- Używaj plików konfiguracyjnych do wartości domyślnych.
- Nadpisuj ustawienia przez zmienne środowiskowe w kontenerach.
- Korzystaj z profili (dev, test, prod).
- Unikaj twardo zakodowanych wartości w kodzie.
- Sprawdzaj efektywną konfigurację w logach aplikacji.
- Dokumentuj ważne właściwości.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Externalized configuration oddziela kod od ustawień.
- Spring Boot czyta konfigurację z wielu źródeł.
- Źródła mają określoną kolejność i nadpisują się.
- Najwyższy priorytet mają argumenty startowe.
- Umożliwia łatwe zarządzanie konfiguracją w różnych środowiskach.
- Wspiera budowę nowoczesnych aplikacji (chmura, kontenery, CI/CD).



MAPA 15/100

PROPERTIES, YAML i CONFIG DATA

Jak organizować pliki konfiguracyjne w Spring Boot

Dobrze zorganizowana konfiguracja to spokojniejsza praca i lepsze projekty!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 FORMAT .PROPERTIES

PROSTY I UNIWERSALNY FORMAT

- Plik **application.properties** to klasyczny format konfiguracji w Spring Boot.
- Składnia oparta na parach klucz = **wartość**.
- Dobrze sprawdza się w prostych konfiguracjach.
- Łatwy do edycji i szeroko wspierany przez narzędzia.

2 FORMAT YAML

CZYTELNA I HIERARCHICZNA STRUKTURA

- Plik **application.yml** (lub **.yaml**) pozwala na bardziej czytelną i hierarchiczną konfigurację.
- Wykorzystuje wcięcia i zagnieżdżone struktury.
- Idealny dla większych i bardziej złożonych konfiguracji.
- Mniej powtórzeń, lepsza czytelność i uporządkowanie.

3 PROFILE KONFIGURACYJNE

RÓŻNE USTAWIENIA DLA RÓŻNYCH ŚRODOWISK

- Pliki specyficzne dla profili:
  application-dev.yml
  application-test.yml
  application-prod.yml
- Aktywny profil ustawiasz przez **spring.profiles.active=dev** (lub zmienną środowiskową).
- Każdy profil może nadpisywać lub rozszerzać konfigurację bazową.
- Ułatwia zarządzanie różnymi środowiskami (dev, test, prod).

4 CONFIG DATA (OD SPRING BOOT 2.4+)

NOWY, POTĘŻNY MECHANIZM

- Spring Boot używa Config Data do ładowania plików konfiguracyjnych.
- Obsługuje wiele źródeł konfiguracji i elastyczne ich łączenie.
- Umożliwia importowanie konfiguracji z plików lokalnych i zewnętrznych.
- Zastąpił stary mechanizm ładowania plików konfiguracyjnych.

5 SPRING.CONFIG.IMPORT

IMPORTOWANIE DODATKOWYCH ŹRÓDEŁ

- Właściwość **spring.config.import** pozwala dołączać dodatkowe pliki i źródła konfiguracji.
- Możesz importować pliki lokalne, zewnętrzne (np. z katalogu), a nawet z serwerów (np. Config Server).
- Przykład:
  `spring.config.import=optional:file:./config/`
  Działa także z classpath: i http:.
- Ułatwia modularizację i rozdzielenie wspólnej konfiguracji od konfiguracji środowiskowej.

6 PLIKI ZEWNĘTRZNE

KONFIGURACJA POZA APLIKACJĄ

- Możesz umieścić pliki konfiguracyjne poza plikiem JAR (np. w katalogu /config).
- Przydatne w środowiskach produkcyjnych (Docker, Kubernetes).
- Spring Boot automatycznie szuka plików w lokalizacjach zewnętrznych (np. /config, aktualny katalog).
- Pozwala na zmianę konfiguracji bez przebudowy aplikacji.

7 KIEDY PROPERTIES VS YAML?

PORÓWNANIE W PRAKTYCE

Użyj .properties, gdy:

- Masz prostą konfigurację.
- Wolisz płaską strukturę.
- Pracujesz w małym zespole.
- Potrzebujesz szybko zmienić jedną wartość.

Użyj .yml, gdy:

- Masz złożoną konfigurację.
- Potrzebujesz hierarchii i lepszej czytelności.
- Pracujesz z wieloma profilami.
- Chcesz uniknąć powtórzeń.

8 TEN SAM PRZYKŁAD – DWA FORMATY

TA SAMA KONFIGURACJA W PROPERTIES I YAML

application.properties

```
# Serwer
server.port=8080

# Baza danych
spring.datasource.url=
jdbc:mysql://localhost:3306/
mydb

# Własne ustawienie
app.mail.enabled=true
```

application.yml

```
# Serwer
server:
  port: 8080
# Baza danych
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/
         mydb
# Własne ustawienie
app:
  mail:
    enabled: true
```

9 PUŁAPKI I NAJCZĘSTSZE BŁĘDY

NA CO UWAŻAĆ?

- Błędne wcięcia w pliku YAML (aplikacja się nie uruchomi).
- Nazwa pliku niezgodna z profilem (np. **application-dev.yml** zamiast application-dev.yml).
- Nadpisanie wartości w nieoczekiwany sposób.
- Niepoprawna ścieżka w **spring.config.import**.
- Trzymanie wrażliwych danych (np. haseł) w repozytorium.

10 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Spring Boot wspiera **.properties** i **.yml**.
- Profile pozwalają na różne konfiguracje dla środowisk.
- Config Data umożliwia elastyczne ładowanie konfiguracji.
- **spring.config.import** pozwala dołączać dodatkowe pliki.
- Pliki zewnętrzne ułatwiają zarządzanie konfiguracją.
- Wybierz format dopasowany do potrzeb projektu.
- Dobra organizacja konfiguracji to stabilniejsze i łatwiejsze w utrzymaniu aplikacje.



MAPA 16/100

@CONFIGURATION PROPERTIES VS @VALUE

Dwa sposoby pobierania konfiguracji i kiedy użyć każdego z nich

Wybierz właściwe narzędzie do właściwego zadania!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO WIĄZAĆ KONFIGURACJĘ Z KODEM?

KORZYŚCI Z KONFIGIGURACJI

- Oddzielenie konfiguracji od kodu aplikacji.
- Łatwe dostosowanie aplikacji do różnych środowisk (dev, test, prod).
- Większa elastyczność i łatwiejsze zarządzanie.
- Konfiguracja może być zmieniana bez rekompilacji aplikacji.

2 @Value

PROSTE POBIERANIE POJEDYNCZYCH WARTOŚCI

- Używa wyrażenia SpEL (np. ${...}).
- Najlepszy do pojedynczych wartości (String, liczby, itp.).
- Może być używany w polach, setterach lub konstruktorze.
- Szybki i prosty w użyciu.

PRZYKŁAD

```java
@Component
public class ServerInfo {

    @Value("${server.port}")
    private int port;

    public int getPort() {
        return port;
    }
}
```

3 @ConfigurationProperties

MAPOWANIE CAŁYCH GRUP WŁAŚCIWOŚCI

- Mapuje grupę powiązanych właściwości do obiektu.
- Wymaga klasy z prefiksem i adnotacji @ConfigurationProperties.
- Idealny do bardziej złożonej konfiguracji.
- Obsługuje zagnieżdżone struktury, walidację i konwersję typów.

PRZYKŁAD

```
@ConfigurationProperties(prefix = "app.mail")
@Validated
public class MailProperties {
    private String host;
    private int port = 587;
    private String username;
    private String password;
} // getters i setters
```

4 TYPE-SAFE BINDING

BEZPIECZNE WIĄZANIE TYPÓW

- Automatyczna konwersja typów (String → int, boolean, Duration, itd.).
- Wczesne wykrywanie błędów konfiguracji.
- Obsługa typów z biblioteki Spring Boot (np. Duration, DataSize).
- Mniej błędów w czasie działania.

5 ZAGNIEŻDŻONE STRUKTURY

OBSŁUGA HIERARCHICZNYCH DANYCH

- Naturalne mapowanie zagnieżdżonych właściwości (np. app.mail, app.security).
- Możliwość tworzenia zagnieżdżonych klas w @ConfigurationProperties.
- Czytelna struktura odpowiadająca plikowi application.properties/yml.
- Dobrze sprawdza się przy większych konfiguracjach.

6 WALIDACJA KONFIGURACJI

WIĘKSZA NIEZAWODNOŚĆ

- Wsparcie dla Bean Validation (np. @NotBlank, @Min, @Max).
- Błędy wykrywane podczas startu aplikacji.
- Możliwość definiowania własnych komunikatów błędów.
- Zapewnia poprawność i kompletność konfiguracji.

7 CZYTELNOŚĆ I UTRZYMANIE

LEPSZA ORGANIZACJA KODU

- @ConfigurationProperties grupuje powiązane ustawienia.
- Łatwiejsze utrzymanie i rozwój aplikacji.
- Nowi członkowie zespołu szybciej rozumieją konfigurację.
- Dobrze opisane właściwości to lepsza dokumentacja.

8 KIEDY UŻYĆ @Value?

SPRAWDZA SIĘ, GDY:

- Potrzebujesz pojedynczej wartości.
- Konfiguracja jest prosta.
- Używasz wartości w jednej klasie.
- Szybkie rozwiązanie w małych fragmentach kodu.

PRZYKŁADY ZASTOSOWAŃ:

- server.port
- feature.enabled
- prosty adres URL
- pojedyncze hasło lub klucz API

9 KIEDY UŻYĆ @ConfigurationProperties?

NAJLEPSZY WYBÓR, GDY:

- Masz wiele powiązanych ustawień.
- Konfiguracja jest złożona.
- Potrzebujesz walidacji.
- Chcesz zachować porządek i czytelność.
- Tworzysz bibliotekę lub większą aplikację.

PRZYKŁADY ZASTOSOWAŃ:

- konfiguracja bazy danych
- ustawienia maila
- integracje zewnętrzne (API)
- własne moduły konfiguracyjne

10 PUŁAPKI

NA CO UWAŻAĆ?

- Literówki w nazwach właściwości.
- Brak prefixu w @ConfigurationProperties.
- Brak getterów/setterów (lub użycie final bez konstruktora).
- Niepoprawna konwersja typów.
- Brak walidacji.
- Używanie @Value do dużych, złożonych konfiguracji (to trudne w utrzymaniu).

11 SZYBKIE PODSUMOWANIE

Ta sama konfiguracja – różne podejścia, różne możliwości!

@Value	@ConfigurationProperties
Pojedyncze wartości	Grupy powiązanych ustawień
Prosty i szybki w użyciu	Type-safe binding
Sprawdza się w małych przypadkach	Obsługa zagnieżdżonych struktur
Mniej kodu	Walidacja
Brak zaawansowanych funkcji	Lepsza czytelność i utrzymanie
	Rekomendowane dla większych projektów

Dobrze skonfigurowana aplikacja to stabilna aplikacja!



MAPA 17/100

PROFILE I PROFILE GROUPS

Jak uruchamiać tę samą aplikację w różnych środowiskach

Zmieniaj środowiska bez zmiany kodu!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO PROFILE?

JEDNA APLIKACJA, WIELE ŚRODOWISK

- Umożliwiają uruchamianie tej samej aplikacji w różnych środowiskach (dev, test, prod).
- Oddzielają konfigurację dla różnych celów.
- Zwiększają elastyczność i bezpieczeństwo.
- Pozwalają łatwo dostosować zachowanie aplikacji do kontekstu.

2 AKTYWACJA PROFILU

WIELE SPOSOBÓW

Przez parametr uruchomieniowy:

```
--spring.profiles.active=dev
```

Przez zmienną środowiskową:

```
SPRING_PROFILES_ACTIVE=dev
```

W pliku application.yml:

```
spring:
  profiles:
    active: dev
```

Domyślnie można ustawić profil w kodzie lub w konfiguracji.

3 PLIK application-dev.yml

KONFIGURACJA DLA KONKRETNEGO PROFILU

application-dev.yml

```
spring:
  datasource:
    url: jdbc:h2:mem:devdb
    username: sa
    password:
  logging:
    level:
      root: DEBUG
```

Pliki nazwane **application-{profil}.yml** są automatycznie ładowane, gdy dany profil jest aktywny.

4 @Profile

WARUNKOWE ŁADOWANIE BEANÓW

Przykład:

```
@Profile("dev")
@Component
public class DevService {
   // logika tylko dla profilu dev
}
```

Bean zostanie utworzony tylko wtedy, gdy aktywny jest profil „dev".

5 PROFILE GROUPS

ŁĄCZENIE PROFILI

- Profile groups pozwalają aktywować kilka profili jednocześnie.
- Ułatwiają zarządzanie złożonymi środowiskami.
- Definiuje się je w pliku application.yml za pomocą właściwości spring.profiles.group.
- Idealne do środowisk produkcyjnych lub testowych z wieloma ustawieniami.

6 PRZYKŁAD GRUPY: PRODUCTION

production = mysql + security + monitoring

application.yml

```
spring:
  profiles:
    group:
      production:
        - mysql
        - security
        - monitoring
```

Aktywując profil „production", automatycznie włączysz profile mysql, security i monitoring.

7 DOMYŚLNY PROFIL

GDY ŻADEN PROFIL NIE JEST AKTYWNY

Spring Boot używa profilu **„default"**, jeśli nie wskazano żadnego innego profilu.

Można go skonfigurować:

```
spring:
  profiles:
    default: dev
```

8 ZASTOSOWANIA

GDZIE PROFILE SIĘ SPRAWDZAJĄ

- Różne bazy danych (H2, MySQL, PostgreSQL)
- Różne ustawienia logowania
- Włączanie/wyłączanie zabezpieczeń
- Konfiguracja zewnętrznych usług (np. SMTP, API)
- Ustawienia specyficzne dla środowiska (test, staging, production)

9 PUŁAPKI

NA CO UWAŻAĆ

- Zapomniany aktywny profil może prowadzić do błędnej konfiguracji.
- Nie wszystkie właściwości można łatwo nadpisać między profilami.
- Zbyt wiele profili może wprowadzić chaos – warto używać profile groups.

10 W PRAKTYCE

TYPOWY SCENARIUSZ

1. Utwórz pliki **application-dev.yml**, **application-prod.yml** itp.
2. Skonfiguruj profile groups, jeśli potrzeba.
3. Aktywuj profil odpowiednią metodą.
4. Uruchom aplikację i sprawdź, która konfiguracja została załadowana.

11 SZYBKIE PODSUMOWANIE

- **Profile** pozwalają uruchamiać tę samą **aplikację** w **różnych** środowiskach.
- Aktywujesz je parametrem, zmienną środowiskową lub w pliku YAML.
- **@Profile** umożliwia warunkowe ładowanie beanów.
- **Profile groups** łączą kilka profili w jeden.
- **Domyślny profil** to **„default"**.
- Ułatwiają zarządzanie konfiguracją i zwiększają elastyczność aplikacji.



MAPA 18/100

ENV VARS, SEKRETY I KONTEJNERY

Jak bezpiecznie przekazywać konfigurację aplikacji poza kodem

Trzymaj sekrety poza repozytorium. Bezpieczniejszy kod, spokojniejsza głowa!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 DLACZEGO ENV VARS?

KONFIGURACJA POZA KODEM

- Rozdziela konfigurację od kodu aplikacji.
- Zwiększa bezpieczeństwo (wrażliwe dane nie w repo).
- Ułatwia wdrażanie w różnych środowiskach (dev, test, prod).
- Pozwala szybko zmieniać ustawienia bez przebudowy aplikacji.
- To standard w kontenerach i chmurze.

2 MAPOWANIE NAZW ZMIENNYCH

OD ENV VAR DO WŁAŚCIWOŚCI SPRINGA

Zmienna środowiskowa (ENV VAR)	Właściwość Spring Boot (application.properties)
SPRING_DATASOURCE_URL	spring.datasource.url
SPRING_DATASOURCE_USERNAME	spring.datasource.username
SPRING_DATASOURCE_PASSWORD	spring.datasource.password
SERVER_PORT	server.port
SPRING_PROFILES_ACTIVE	spring.profiles.active

Zasada: nazwy ENV VAR są UPPER_CASE i używają podkreślników (_), a Spring Boot automatycznie mapuje je na zapis kropkowy.

3 SEKRETY (SECRETS)

WRAŻLIWE DANE

- Hasła do baz danych
- Klucze API, tokeny, certyfikaty
- Dane dostępowe do zewnętrznych usług (np. AWS, Stripe)
- **Nie przechowuj sekretów w repo!**
- Używaj zmiennych środowiskowych, plików sekretów lub narzędzi (np. Docker secrets, Kubernetes Secrets, Vault).

4 DOCKER I KONTENERY

ENV VARS W KONTENERACH

- Przekazujesz konfigurację przez zmienne środowiskowe.
- Ta sama aplikacja działa w różnych środowiskach bez zmian w kodzie.
- Łatwe wdrażanie w CI/CD i chmurze.

5 MONTOWANE PLIKI LUB SEKRETY

INNE SPOSOBY PRZEKAZANIA DANYCH

- Możesz montować pliki z konfiguracją (np. application.properties).
- W Dockerze: użyj volume z plikiem lub mechanizmu Docker secrets.
- W Kubernetes: użyj ConfigMap i Secret (montowane jako pliki lub zmienne środowiskowe).
- Spring Boot automatycznie wczyta pliki z typowych lokalizacji (np. /config, /secrets).

6 PRZYKŁADY UŻYCIA

RÓŻNE SPOSOBY

```
# Przekazanie ENV VAR w Dockerze
docker run -e SPRING_PROFILES_ACTIVE=prod \\
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/app \
  -e SPRING_DATASOURCE_USERNAME=app \
  -e SPRING_DATASOURCE_PASSWORD=secret \
  myapp:latest

# Lub w docker-compose.yml
environment:
    - SPRING_PROFILES_ACTIVE=prod
    - SERVER_PORT=8080
```

7 W PRAKTYCE

TYPOWY PRZEPŁYW KONFIGURACJI

1. Deweloper uruchamia lokalnie (z .env lub ENV VAR).
2. CI/CD przekazuje zmienne do kontenera.
3. Aplikacja wczytuje konfigurację automatycznie.
4. Ta sama aplikacja działa w dev, test i prod – bez zmian w kodzie.

8 PUŁAPKI

NA CO UWAŻAĆ?

- Przechowywanie sekretów w repo (np. application.properties).
- Brak szyfrowania wrażliwych danych.
- Nadpisywanie zmiennych w nieoczekiwany sposób.
- Różne wartości w różnych środowiskach mogą powodować błędy.
- Logowanie wrażliwych danych (np. haseł).

9 DOBRE PRAKTYKI

BEZPIECZNIE I WYGODNIE

- Trzymaj sekrety poza repozytorium.
- Używaj ENV VAR lub mechanizmów sekretów platformy.
- Stosuj różne profile (dev, test, prod).
- Używaj .env tylko lokalnie (i dodaj do .gitignore).
- Minimalne uprawnienia dla kont i kluczy.
- Rotuj sekrety regularnie.
- Monitoruj dostęp do wrażliwych zasobów.

10 MINI PRZYKŁAD

PODSTAWOWE ZMIENNE W PRAKTYCE

```
# Uruchom aplikację z podstawową konfiguracją
docker run -e SPRING_PROFILES_ACTIVE=prod \
           -e SERVER_PORT=8080 \
           -e SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/app \
           -e SPRING_DATASOURCE_USERNAME=app \
           -e SPRING_DATASOURCE_PASSWORD=secret \
           myapp:latest
```

SCHEMAT DZIAŁANIA



11 SZYBKIE PODSUMOWANIE

- Env vars pozwalają bezpiecznie konfigurować aplikację.
- Spring Boot automatycznie mapuje zmienne na właściwości.
- Używaj mechanizmów sekretów w kontenerach i chmurze.
- Ta sama aplikacja działa w różnych środowiskach.
- Dobre praktyki i ostrożność chronią Twoje dane.
- Konfiguracja poza kodem = większe bezpieczeństwo i elastyczność.



MAPA 19/100

LOGGING W SPRING BOOT

Jak konfigurować logi, poziomy i podstawową diagnostykę aplikacji

Dobre logi oszczędzają czas i nerwy!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO LOGOWANIE?

WIĘCEJ NIŻ TYLKO KOMUNIKATY

- Pomaga zrozumieć, co dzieje się w aplikacji.
- Ułatwia diagnozowanie błędów i problemów.
- Wspiera monitorowanie i analizę działania systemu.
- Jest niezbędne w środowisku produkcyjnym.
- Daje lepszą widoczność dla całego zespołu.

2 DOMYŚLNY LOGGING W SPRING BOOT

LOGBACK POD MASKĄ

Spring Boot domyślnie używa Logback – wydajnego i elastycznego frameworka do logowania.

- Logback jest częścią Spring Boot.
- Automatycznie konfigurowany (z rozsądnymi ustawieniami).
- Można go łatwo dostosować przez plik application.properties, application.yml lub własny plik logback-spring.xml.
- Współpracuje z SLF4J (jednolity interfejs logowania).

3 POZIOMY LOGOWANIA

OD NAJBARDZIEJ SZCZEGÓŁOWYCH DO NAJWAŻNIEJSZYCH

TRACE	Bardzo szczegółowe informacje dla zaawansowanej diagnostyki.
DEBUG	Informacje pomocne dla programistów podczas rozwiązywania problemów.
INFO	Ogólne informacje o działaniu aplikacji (domyślny poziom).
WARN	Ostrzeżenia – coś nie działa tak, jak oczekiwano, ale aplikacja działa.
ERROR	Błędy – poważne problemy, które wymagają uwagi.

4 LOGGING.LEVEL.ROOT I DLA PAKIETÓW

KONTROLA POZIOMU LOGÓW

logging.level.root
Ustawia poziom logowania dla całej aplikacji.

logging.level.com.example
Ustawia poziom tylko dla wybranego pakietu.

Dzięki temu możesz mieć szczegółowe logi w swoim kodzie i mniej logów z bibliotek zewnętrznych.

5 GDZIE TRAFIAJĄ LOGI?

KONSOLA, PLIK, SYSTEMY ZEWNĘTRZNE

- Domyślnie logi są wypisywane do **konsoli** (stdout).
- Możesz zapisać logi do pliku.
- W środowisku produkcyjnym logi często trafiają do systemów centralnego logowania (np. ELK, Splunk, Grafana Loki).
- W kontenerach (np. Docker) zwykle używa się logów w konsoli.

6 LOGGING.FILE.NAME I LOGGING.FILE.PATH

ZAPIS LOGÓW DO PLIKU

logging.file.name
Określa dokładną nazwę pliku z logami (np. app.log).

logging.file.path
Określa katalog, w którym będą zapisywane logi.

Jeśli ustawisz tylko ścieżkę (path), Spring Boot utworzy w niej domyślny plik o nazwie **spring.log**.

7 CZYTELNE LOGI

DOBRE PRAKTYKI

- Używaj zrozumiałych komunikatów.
- Dodawaj istotne konteksty (np. ID użytkownika).
- Korzystaj z parametrów zamiast konkatenacji napisów ({}).
- Stosuj jednolity format logów.
- Unikaj logowania wrażliwych danych.

8 CZEGO UNIKAĆ?

NAJCZĘSTSZE BŁĘDY

- Logowania zbyt dużej ilości informacji (nadmiar logów).
- Logowania danych wrażliwych (np. haseł, tokenów, danych osobowych).
- Używania logów do kontroli przepływu logiki aplikacji.
- Pozostawiania poziomu DEBUG w środowisku produkcyjnym.
- Nieczytelnych lub zbyt ogólnych komunikatów.

9 W PRAKTYCE

GDZIE LOGOWANIE JEST POMOCNE?

- Diagnozowanie błędów w produkcji.
- Monitorowanie wydajności aplikacji.
- Śledzenie działań użytkowników.
- Analiza problemów z integracjami.
- Szybsze rozwiązywanie incydentów.

10 PRZYKŁADOWA KONFIGURACJA

APPLICATION.PROPERTIES

```
# Poziom logów dla całej aplikacji
logging.level.root=INFO
# Bardziej szczegółowe logi dla naszego pakietu
logging.level.com.example=DEBUG
# Zapis logów do pliku
logging.file.name=logs/app.log
# Alternatywnie: tylko ścieżka (utworzy spring.log)
# logging.file.path=logs
```

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- Spring Boot używa Logbacka.
- Możesz łatwo konfigurować poziomy logów.
- Logi mogą trafiać do konsoli, pliku lub systemów zewnętrznych.
- Czytelne logi przyspieszają diagnozowanie problemów.
- Unikaj logowania wrażliwych danych.

LEGENDA POZIOMÓW LOGÓW

TRACE	bardzo szczegółowe
DEBUG	diagnostyczne
INFO	informacyjne
WARN	ostrzeżenia
ERROR	błędy



Lepsze logi, lepsze aplikacje!

MAPA 20/100

APPLICATION EVENTS, RUNNERY I HOOKI STARTOWE

Jak reagować na etapy uruchamiania aplikacji Spring Boot

Reaguj we właściwym momencie. Twój kod ma znaczenie!

MODUŁ 02 • AUTO-KONFIGURACJA I KONFIGURACJA APLIKACJI

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO HOOKI STARTOWE?

WIĘCEJ MOŻLIWOŚCI, LEPSZA KONTROLA

- Pozwalają wykonywać kod w określonych momentach uruchamiania aplikacji.
- Ułatwiają inicjalizację zasobów, ładowanie danych i konfigurację.
- Dają większą kontrolę nad cyklem życia aplikacji.
- Pomagają w integracji z zewnętrznymi systemami.
- Umożliwiają reagowanie na sukces lub błąd uruchomienia.

2 NAJWAŻNIEJSZE EVENTY

KLUCZOWE ETAPY STARTU APLIKACJI

1. `ApplicationStartingEvent`
2. `ApplicationEnvironmentPreparedEvent`
3. `ApplicationPreparedEvent`
4. `ApplicationStartedEvent`
5. `ApplicationReadyEvent`
6. `ApplicationFailedEvent`

3 CO OZNACZA KAŻDY ETAP?

ApplicationStartingEvent
Bardzo wczesny etap. Aplikacja zaczyna się uruchamiać. Brak dostępu do kontekstu Spring.

ApplicationEnvironmentPreparedEvent
Środowisko (Environment) jest już gotowe. Można czytać właściwości konfiguracyjne.

ApplicationPreparedEvent
Kontekst aplikacji jest przygotowany, ale jeszcze nie zostały wywołane runnery.

ApplicationStartedEvent
Kontekst został odświeżony. Aplikacja wystartowała, ale może jeszcze nie być gotowa do obsługi żądań.

ApplicationReadyEvent
Aplikacja jest w pełni gotowa do działania. Wszystkie runnery zostały wykonane.

ApplicationFailedEvent
Uruchomienie aplikacji zakończyło się błędem.

4 APPLICATIONRUNNER

DOSTĘP DO ARGUMENTÓW APLIKACJI

- Interfejs: `ApplicationRunner`
- Metoda: `run(ApplicationArguments args)`
- Wykonywany po uruchomieniu kontekstu, ale przed `ApplicationReadyEvent`.
- Umożliwia dostęp do argumentów aplikacji (`--param=value`).
- Świetny do inicjalizacji zależnej od parametrów startowych.

5 COMMANDLINERUNNER

PROSIGE PODEJŚCIE

- Interfejs: `CommandLineRunner`
- Metoda: `run(String... args)`
- Wykonywany po uruchomieniu kontekstu, ale przed `ApplicationReadyEvent`.
- Otrzymuje surowe argumenty z linii poleceń.
- Dobry wybór do prostych zadań inicjalizacyjnych.

6 @EVENTLISTENER

REAKCJA NA KONKRETNE EVENTY

- Adnotacja: `@EventListener`
- Może nasłuchiwać dowolnego eventu (z np. `ApplicationReadyEvent`).
- Metoda może przyjmować obiekt eventu jako parametr.
- Pozwala precyzyjnie reagować na wybrane etapy startu.
- Bardzo elastyczne rozwiązanie, działa z wszystkimi eventami Spring Boot.

7 RUNNER CZY LISTENER?

KIEDY UŻYĆ KTÓREGO?

Cecha	ApplicationRunner / CommandLineRunner	@EventListener
Dostęp do argumentów	Tak	Nie (chyba że w evencie)
Wykonanie	Po starcie kontekstu	Dowolny etap
Elastyczność	Ograniczona	Bardzo duża
Zastosowanie	Prosta inicjalizacja	Reakcja na konkretne eventy
Wiele implementacji	Można (kolejność!)	Można (kolejność!)

8 TYPOWE ZASTOSOWANIA

GDZIE HOOKI SPRAWDZAJĄ SIĘ NAJLEPIEJ?

- Ładowanie danych startowych (np. słowników, cache).
- Sprawdzanie połączeń z zewnętrznymi systemami (DB, API, kolejki).
- Inicjalizacja klientów (np. REST, Kafka).
- Walidacja konfiguracji i środowiska.
- Planowanie zadań, które mają ruszyć po starcie.
- Wysyłanie powiadomień o starcie aplikacji.

9 PUŁAPKI I NA CO UWAŻAĆ?

- Zbyt długi czas wykonywania – wydłuża start aplikacji.
- Złożona logika – lepiej przenieść do serwisu.
- Zależności od zasobów, które jeszcze nie są gotowe (np. przy wczesnych eventach).
- Brak obsługi błędów – wyjątek może zatrzymać start.
- Uruchamianie zadań blokujących w głównym wątku – rozważ `@Async`.
- Nieprawidłowa kolejność wielu runnerów – użyj `@Order` lub `Ordered`.

10 PRZYKŁAD – REAKCJA NA APPLICATIONREADYEVENT

Java

```java
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class StartupNotification {
    @EventListener(ApplicationReadyEvent.class)
    public void onReady(ApplicationReadyEvent event) {
        System.out.println("✅ Aplikacja wystartowała i jest gotowa!");
    }
}
```

Prosty sposób na reakcję po pełnym starcie aplikacji.

11 LINIA CZASU STARTU APLIKACJI

KOLEJNOŚĆ WYWOŁYWANIA EVENTÓW I RUNNERÓW



Wszystkie te mechanizmy pozwalają Ci uruchamiać kod wtedy, gdy naprawdę ma to sens. Wybierz odpowiedni moment i buduj lepsze, bardziej niezawodne aplikacje!

Lepszy moment. Lepszy kod. Większe możliwości!

MAPA 21/100

SERVLET STACK I WBUDOWANY SERWER

Jak działa klasyczny model webowy Spring Boot od żądania HTTP do odpowiedzi

Ten sam kod, większe możliwości! Uruchom aplikację bez serwera zewnętrznego!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST SERVLET STACK?

KLASYCZNY MODEL WEBOWY W SPRING BOOT

- Servlet Stack to model oparty na serwletach (Jakarta Servlet) i architekturze request-response.
- Aplikacja działa w kontenerze serwletów, który obsługuje żądania HTTP.
- Spring MVC działa na warstwie serwletów, upraszczając obsługę żądań i odpowiedzi.
- To sprawdzony, stabilny i szeroko używany model twórczenia aplikacji webowych.

2 ROLA WBUDOWANEGO SERWERA

TOMCAT, JETTY LUB UNDERTOW

- Domyślnie Spring Boot używa wbudowanego serwera Tomcat (wersja embedded).
- Możesz również użyć Jetty lub Undertow – wystarczy zmienić zależności.
- Aplikacja uruchamia się jako samodzielny plik JAR (executable JAR) – bez potrzeby instalowania zewnętrznego serwera aplikacji (np. Tomcat).
- Wystarczy polecenie: `java -jar app.jar`

3 OD PRZEGLĄDARKI DO KONTROLERA

PRZEPŁYW ŻĄDANIA HTTP

1. **Klient (przeglądarka)**
   Wysyła żądanie HTTP (np. GET /produkty).
2. **Wbudowany serwer**
   Tomcat/Jetty/Undertow odbiera żądanie.
3. **DispatcherServlet**
   Przekazuje żądanie do odpowiedniego kontrolera (na podstawie mapowania).
4. **Kontroler**
   Obsługuje żądanie, wywołuje serwis.
5. **Serwis**
   Logika biznesowa, dostęp do danych.
6. **Odpowiedź**
   Zwracana do klienta (np. JSON lub widok).

4 DISPATCHERSERVLET

FRONT CONTROLLER W SPRING MVC

- Centralny punkt obsługi żądań w Spring MVC (Front Controller).
- Odpowiada za mapowanie żądań do odpowiednich kontrolerów.
- Obsługuje wiązanie danych (data binding) i walidację argumentów.
- Zajmuje się obsługą wyjątków oraz zwraca widok (np. Thymeleaf) lub odpowiedź REST (np. JSON).

5 CO BOOT KONFIGURUJE AUTOMATYCZNIE?

MNIEJ KONFIGURACJI, WIĘCEJ DZIAŁANIA

- Wbudowany serwer (Tomcat/Jetty/Undertow) z sensownymi ustawieniami.
- Podstawową konfigurację Spring MVC (m.in. DispatcherServlet, konwertery).
- Bibliotekę Jackson do automatycznej konwersji JSON.
- Obsługę stron błędów (np. /error).
- Obsługę zasobów statycznych (np. /static, /public, /resources).

6 TOMCAT VS JETTY VS UNDERTOW

PORÓWNANIE WBUDOWANYCH SERWERÓW

Cecha	Tomcat	Jetty	Undertow
Domyślny w Spring Boot	Tak	Nie	Nie
Wydajność	Dobra	Bardzo dobra	Bardzo dobra
Rozmiar	Średni	Mniejszy	Mały
Stabilność	Bardzo wysoka	Wysoka	Wysoka
Zastosowanie	Ogólne zastosowanie	Lekka alternatywa	Wysoka wydajność
Konfiguracja w Boot	Domyślna	Zmiana zależności	Zmiana zależności

7 PAKOWANIE I URUCHOMIENIE

SAMODZIELNA APLIKACJA JAR

- Spring Boot tworzy wykonywalny plik JAR z wbudowanym serwerem.
- Nie potrzebujesz zewnętrznego serwera aplikacji.
- Wystarczy uruchomić aplikację poleceniem: `java -jar app.jar`
- Aplikacja jest od razu gotowa do obsługi żądań HTTP.

8 KIEDY WYBRAĆ SERVLET STACK?

SPRAWDZONY WYBÓR W WIELU SCENARIUSZACH

- Tworzysz klasyczne aplikacje REST API.
- Potrzebujesz aplikacji renderujących widoki (np. Thymeleaf).
- Korzystasz z szerokiego ekosystemu bibliotek i narzędzi.
- Zależy Ci na stabilnym i dojrzałym stosie technologicznym.
- Aplikacja ma obsługiwać wiele integracji i tradycyjnych bibliotek (np. Spring Security, Spring Data).

9 W PRAKTYCE

TYPOWA KONFIGURACJA ZALEŻNOŚCI

Najczęściej używane zależności dla aplikacji webowej:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframoeork.boot</groupId>
    <artifactId>spring-boot-starter-validaton</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframeeork.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
</dependencies>
```

10 PUŁAPKI

NA CO UWAŻAĆ?

- Mieszanie blokującego modelu servletów z oczekiwaniami modelu reaktywnego (WebFlux).
- Zbyt duża liczba filtrów i interceptorów – może obniżyć wydajność.
- Nieporozumienia dotyczące wbudowanego serwera – to nadal pełnoprawny kontener servletów!
- Brak zrozumienia cyklu życia żądania i roli DispatcherServlet.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Servlet Stack opiera się na servletach i modelu request-response.
- Spring Boot dostarcza wbudowany serwer (Tomcat domyślnie).
- DispatcherServlet obsługuje routing, wiązanie danych i wyjątki.
- Boot automatycznie konfiguruje kluczowe elementy Spring MVC.
- To stabilny, dojrzały i wszechstronny model twórczenia aplikacji webowych.



MAPA 22/100

DROGA ŻĄDANIA PRZEZ SPRING MVC

Co dzieje się od wejścia żądania HTTP do odesłania odpowiedzi

Ten sam mechanizm działa dla widoków HTML i REST API!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 POCZĄTEK: KLIENT HTTP

PRZEGLĄDARKA LUB APLIKACJA

Klient (np. przeglądarka, aplikacja mobilna, inny serwis) wysyła żądanie HTTP do naszej aplikacji.

Żądanie zawiera metodę HTTP (GET, POST, PUT, DELETE), adres URL, nagłówki i ewentualnie ciało żądania (body).

Przykład:

```
GET /users/5
Accept: application/json
```

2 SERWER I ŁAŃCUCH FILTRÓW

PRZYJĘCIE ŻĄDANIA I WSTĘPNA OBSŁUGA

Wbudowany serwer (np. Tomcat w Spring Boot) przyjmuje żądanie HTTP.

Żądanie przechodzi przez łańcuch filtrów (Filter), które mogą je przetwarzać (np. logowanie, uwierzytelnianie, CORS, kompresja).

Filtry mogą modyfikować żądanie i odpowiedź lub zablokować dalsze przetwarzanie.

3 DISPATCHERSERVLET

CENTRALNY PUNKT WEJŚCIA

DispatcherServlet to front controller w Spring MVC.

Przyjmuje żądanie po filtrach i koordynuje dalsze przetwarzanie.

Deleguje znalezienie odpowiedniego kontrolera do komponentów Spring MVC (HandlerMapping).

Jeden DispatcherServlet obsługuje wszystkie żądania dla aplikacji (standardowo pod ścieżką "/").

4 HANDLERMAPPING

ZNALEZIENIE ODPOWIEDNIEGO KONTROLERA

HandlerMapping na podstawie URL, metody HTTP i innych kryteriów znajduje odpowiednią metodę w kontrolerze.

W Spring MVC działa wiele implementacji, np.:

- RequestMappingHandlerMapping
- BeanNameUrlHandlerMapping
- RouterFunctionMapping (dla WebFlux)

Wynikiem jest obiekt HandlerExecutionChain z wybraną metodą kontrolera i interceptorami.

5 HANDLERADAPTER

WYWOŁANIE METODY KONTROLERA

HandlerAdapter odpowiedzialny jest za faktyczne wywołanie metody w kontrolerze.

Najczęściej używany jest RequestMappingHandlerAdapter, który obsługuje adnotacje @RequestMapping i @*Mapping.

Adapter przygotowuje argumenty metody, wywołuje ją i zwraca wynik (np. obiekt, nazwę widoku, ResponseEntity).

6 BINDOWANIE DANYCH

PRZEKAZANIE DANYCH DO METODY

Dane z żądania (ścieżka, parametry zapytania, ciało JSON, nagłówki) są wiązane z argumentami metody kontrolera.

Spring automatycznie konwertuje typy (np. String → Long, JSON → obiekt) przy pomocy konwerterów (Converter).

Możliwa jest walidacja danych przy użyciu adnotacji, np. @Valid i Bean Validation (JSR-380).

W przypadku błędów walidacji rzucane są wyjątki (np. MethodArgumentNotValidException).

7 WARSTWA SERWISÓW I REPOZYTORIÓW

LOGIKA BIZNESOWA I DOSTĘP DO DANYCH

Kontroler wywołuje warstwę serwisów (Service), która zawiera logikę biznesową.

Serwisy korzystają z repozytoriów (np. Spring Data JPA) do dostępu do bazy danych.

Ta warstwa odpowiada za przetwarzanie danych, transakcje i integrację z innymi systemami.

Kontroler powinien być cienki – logika biznesowa powinna znajdować się w serwisach.

8 TWORZENIE ODPOWIEDZI

WIDOK HTML LUB JSON (REST API)

Wynik metody kontrolera jest przetwarzany na odpowiedź HTTP.

Dla REST API obiekty są automatycznie serializowane do JSON (np. przez Jacksona) – zwykle przy użyciu @ResponseBody lub @RestController.

Dla aplikacji web z widokami wynik może być nazwą widoku (np. "users"), który jest renderowany przez silnik szablonów (np. Thymeleaf).

Dodawane są odpowiednie nagłówki HTTP i kod statusu (np. 200, 201, 404).

9 OBSŁUGA WYJĄTKÓW

PRZECHWYTYWANIE I OBSŁUGA BŁĘDÓW

Wyjątki rzucone w kontrolerze lub serwisach są przechwytywane przez mechanizmy Spring MVC.

HandlerExceptionResolver i @ControllerAdvice pozwalają obsłużyć wyjątki w spójny sposób.

Możesz zwrócić własny kod błędu, komunikat i strukturę odpowiedzi (np. ErrorResponse).

Dzięki globalnej obsłudze wyjątków aplikacja zwraca czytelne i spójne błędy dla klienta (np. w formacie JSON).

10 W PRAKTYCE

PRZYKŁAD PRZEPŁYWU DLA GET /users/5

1. Klient wysyła żądanie: `GET /users/5`
2. Serwer i filtry przyjmują i przetwarzają żądanie
3. DispatcherServlet przekazuje żądanie dalej
4. HandlerMapping znajduje metodę kontrolera
5. HandlerAdapter wywołuje metodę
6. Dane są bindowane do argumentów (np. `id = 5`)
7. Metoda wywołuje serwis i repozytorium
8. Wynik jest zamieniany na JSON (ResponseEntity)
9. Odpowiedź wraca do klienta z kodem 200 OK

PRZYKŁAD KONTROLERA (REST API)

PROSTY PRZYKŁAD Z @RESTCONTROLLER

```
@RestController
@RequestMapping("/users")
public class UserController {
   private final UserService userService;
   public UserController(UserService userService) {
      this.userService = userService;
   }
   @GetMapping("/{id}")
   public ResponseEntity<UserDto> getUser(
         @PathVariable Long id) {
      UserDto user = userService.findById(id);
      return ResponseEntity.ok(user);
   }
}
```

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- DispatcherServlet to front controller.
- Filtry mogą przetwarzać żądanie przed MVC.
- HandlerMapping znajduje odpowiedni kontroler.
- HandlerAdapter wywołuje metodę kontrolera.
- Dane są automatycznie bindowane i walidowane.
- Serwisy i repozytoria realizują logikę biznesową.
- Wynik jest renderowany lub serializowany do JSON.
- Istnieje globalna obsługa wyjątków.
- Ten sam mechanizm działa dla widoków i REST API.

MAPA 23/100

@CONTROLLER, @RESTCONTROLLER I REQUEST MAPPING

Jak definiować endpointy i kierować żądania do właściwych metod

Ten sam endpoint. Wielkie możliwości!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 @Controller

TRADYCYJNE APLIKACJE WEB (MVC)

- Oznacza klasę jako kontroler w architekturze MVC po stronie serwera.
- Metody zwykle zwracają nazwę widoku (np. Thymeleaf, JSP).
- Dane przekazywane do widoku przez obiekt Model.
- Używane w klasycznych aplikacjach web z renderowaniem stron po stronie serwera.

2 @RestController

WYGODNE ROZWIĄZANIE DLA API REST

- Łączy w sobie @Controller i @ResponseBody.
- Metody zwracają dane (np. JSON) bezpośrednio w odpowiedzi HTTP.
- API — Idealny do budowy REST API.
- Automatycznie konwertuje zwracane obiekty do formatu JSON (dzięki Jackson).
- Najczęściej używany w nowoczesnych aplikacjach SPA, aplikacjach mobilnych i integracjach między systemami.

3 @RequestMapping

MAPOWANIE ŻĄDAŃ NA KLASY I METODY

- Określa ścieżkę URL, która obsługuje dana klasa lub metoda.
- Może być używany na poziomie klasy (wspólny prefiks) lub metody (konkretna ścieżka).
- Pozwala określić także metodę HTTP (GET, POST, itp.), parametry, nagłówki i warunki.
- Jest bazą dla skrótów takich jak @GetMapping, @PostMapping itd.

4 NAJWAŻNIEJSZE SKRÓTY

WYGODNE ANOTACJE DLA RÓŻNYCH METOD HTTP

@GetMapping	Obsługuje żądania GET (pobieranie danych).
@PostMapping	Obsługuje żądania POST (tworzenie danych).
@PutMapping	Obsługuje żądania PUT (aktualizacja danych).
@DeleteMapping	Obsługuje żądania DELETE (usuwanie danych).
@PatchMapping	Obsługuje żądania PATCH (częściowa aktualizacja).

5 MAPOWANIE ŚCIEŻEK

WZORCE URL I ZMIENNE ŚCIEŻKI

- Ścieżki mogą zawierać stałe fragmenty i zmienne, np. /users/{id}.
- Zmienne ścieżki pobieramy za pomocą @PathVariable.
- Możemy używać wzorców, np.: /users/*, /api/**.
- Można mapować wiele ścieżek do jednej metody.
- Dobrze zaprojektowane ścieżki są czytelne i zgodne z zasadami REST (np. rzeczowniki w liczbie mnogiej).

6 PARAMETRY METODY

JAK POBIERAĆ DANE Z ŻĄDANIA

- @RequestParam – pobiera parametr z query string, np. /users?active=true.
- @PathVariable – pobiera zmienną ze ścieżki, np. /users/{id}.
- @RequestBody – pobiera dane z ciała żądania (np. JSON) i mapuje na obiekt.
- @RequestHeader – pobiera wartość nagłówka HTTP, np. Authorization.

7 STATUS I ODPOWIEDŹ

KONTROLA NAD ODPOWIEDZIĄ HTTP

- ResponseEntity<T> – pełna kontrola nad kodem statusu, nagłówkami i ciałem odpowiedzi.
- @ResponseStatus – pozwala określić stały kod statusu dla metody.
- Najczęściej używane kody: 200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error.
- Zwracaj odpowiednie kody statusu, aby API było czytelne i łatwe w integracji.

8 PRAKTYCZNY PRZYKŁAD

KONTROLER REST Z ENDPOINTAMI CRUD

Java

```java
@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(
        @PathVariable Long id) {
        return userService.findById(id)
          .map(user -> ResponseEntity.ok(user))
          .orElse(ResponseEntity.notFound().build());
    }
}
```

9 KIEDY UŻYWAĆ @CONTROLLER, A KIEDY @RESTCONTROLLER?

PORÓWNANIE PODEJŚĆ

Cecha	@Controller	@RestController
Zastosowanie	Aplikacje web z widokami	API REST (zwraca dane)
Zwracana wartość	Nazwa widoku (String)	Dane (np. JSON)
@ResponseBody	Trzeba dodać ręcznie	Zawiera domyślnie
Typ aplikacji	MVC (Thymeleaf, JSP, itp.)	REST API (SPA, mobilne)
Przykład	`return "users/list";`	`return users;`

10 PUŁAPKI

CZEGO UNIKAĆ W KONTROLERACH

- Zbyt dużo logiki biznesowej w kontrolerze – logika powinna być w serwisach.
- Nieczytelne i niespójne ścieżki URL – stosuj spójną konwencję.
- Brak walidacji danych wejściowych – używaj @Valid i sprawdzaj dane.
- Zwracanie nieodpowiednich kodów statusu – dbaj o poprawne kody HTTP.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- @Controller – do aplikacji MVC z widokami.
- @RestController – do budowy REST API.
- @RequestMapping i jego skróty mapują ścieżki URL do metod.
- Parametry żądania pobieramy przez adnotacje (@RequestParam, @PathVariable, @RequestBody).
- ResponseEntity daje pełną kontrolę nad odpowiedzią HTTP.
- Projektuj czytelne, spójne i zgodne z REST endpointy.



MAPA 24/100

PATH, QUERY, HEADER I BODY

Jak Spring MVC wiąże dane z żądania HTTP do argumentów metody

Różne źródła danych, jeden endpoint. Większe możliwości!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 CZTERY GŁÓWNE ŹRÓDŁA DANYCH

SKĄD SPRING MVC POBIERA DANE?

Path (ścieżka URL)
Wartości w ścieżce adresu, np. /users/42

Query (parametry zapytania)
Parametry po znaku ?, np. ?page=1&size=10

Header (nagłówki HTTP)
Dane z nagłówków żądania, np. X-Trace-Id

Body (ciało żądania)
Dane w treści żądania, np. JSON lub XML

2 @PathVariable

WARTOŚCI ZE ŚCIEŻKI URL

Użyj @PathVariable, aby pobrać fragment ścieżki URL i przypisać go do parametru metody.

```
@GetMapping("/users/{id}")
public User getUser(
    @PathVariable Long id) {
    return userService.findById(id);
}
```

- Obsługuje różne typy (Long, String, UUID).
- Nazwa zmiennej może być jawna lub wynikająca z nazwy parametru.
- Można używać kilku @PathVariable.

3 @RequestParam

PARAMETRY ZAPYTANIA (QUERY)

Użyj @RequestParam, aby pobrać parametry z ciągu zapytania (?param=value).

```
@GetMapping("/users")
public List<User> getUsers(
  @RequestParam(defaultValue = "0") int page,
  @RequestParam(defaultValue = "10") int size,
  @RequestParam(required = false) String sort) {
   return userService.findAll(page, size, sort);
}
```

- Można określić wartość domyślną (defaultValue).
- Ustaw required = false dla parametrów opcjonalnych.
- Wspiera tablice i listy, np. ?id=1&id=2.

4 @RequestHeader

DOSTĘP DO NAGŁÓWKÓW HTTP

Użyj @RequestHeader, aby odczytać wartości nagłówków żądania.

```
@GetMapping("/info")
public String getInfo(
  @RequestHeader("X-Trace-Id") String traceId,
  @RequestHeader(value = "User-Agent",
          required = false) String userAgent) {
   return "Trace: " + traceId;
}
```

- Można pobierać dowolne nagłówki (np. Authorization).
- Użyj required = false, jeśli nagłówek jest opcjonalny.
- Nazwy nagłówków nie są czułe na wielkość liter.

5 @RequestBody

DANE Z CIAŁA ŻĄDANIA

Użyj @RequestBody, aby automatycznie zdeserializować JSON lub XML do obiektu Java (DTO).

```
@PostMapping("/users")
public User createUser(
   @RequestBody CreateUserRequest request) {
   return userService.create(request);
}
```

- Wspiera JSON (domyślnie) i XML (po dodaniu odpowiednich zależności).
- Automatyczna konwersja do obiektów (DTO).
- Wymaga nagłówka Content-Type, np. application/json.

6 KONWERSJA TYPÓW

AUTOMATYCZNE WIĄZANIE I KONWERSJA

Spring MVC automatycznie konwertuje wartości z żądania do odpowiednich typów.

String → int, long, double
np. ?age=25 → int

String → UUID
np. ?id=550e8400-e29b... → UUID

String → Enum
np. ?status=ACTIVE → Status.ACTIVE

String → Data
np. ?date=2024-01-01 → LocalDate (z użyciem @DateTimeFormat)

7 WALIDACJA WEJŚCIA

SPRAWDZANIE POPRAWNOŚCI DANYCH

Użyj @Valid do walidacji obiektów przekazanych w ciele żądania (@RequestBody).

```
@PostMapping("/users")
public ResponseEntity<?> createUser(
   @Valid @RequestBody CreateUserRequest request){
   return ResponseEntity.ok().build();
}
```

- Wykorzystuje adnotacje, np. @NotNull, @Size.
- W przypadku błędów zwraca 400 Bad Request.
- Można obsłużyć błędy za pomocą @ExceptionHandler.

8 PRZYKŁAD ENDPOINTU

WYKORZYSTANIE RÓŻNYCH ŹRÓDEŁ DANYCH

```
@PostMapping("/users/{id}/orders")
public ResponseEntity<OrderResponse> createOrder(
   @PathVariable Long id,
   @RequestParam(defaultValue = "1") int page,
   @RequestHeader("X-Trace-Id") String traceId,
   @Valid @RequestBody CreateOrderRequest request) {
   // logika tworzenia zamówienia
   return ResponseEntity.ok(
         new OrderResponse("OK"));
}
```

PRZYKŁADOWE ŻĄDANIE HTTP

```
POST /users/42/orders?page=1 HTTP/1.1
Host: api.example.com
Content-Type: application/json
X-Trace-Id: abc-123

{
  "productId": 1001,
  "quantity": 2
}
```

Jeden endpoint, cztery źródła danych!

9 KIEDY CZEGO UŻYWAĆ?

PORÓWNANIE TYPOWYCH ZASTOSOWAŃ

Źródło	Kiedy używać?	Przykłady
Path	Identyfikacja konkretnego zasobu	/users/42 /products/{id}
Query	Opcje filtrowania, stronicowanie, sortowanie	?page=1 ?sort=name
Header	Informacje techniczne, autoryzacja, identyfikacja	X-Trace-Id Authorization User-Agent
Body	Większe dane, tworzenie lub aktualizacja zasobów	JSON z danymi XML

10 PUŁAPKI

NAJCZĘSTSZE PROBLEMY

- **Brak nagłówka Content-Type**
  Dla @RequestBody wymagany jest np. application/json.
- **Nieprawidłowe typy danych**
  Niezgodność typu (np. String zamiast liczby) powoduje 400.
- **Nadużywanie nagłówków**
  Nie przesyłaj danych biznesowych w nagłówkach.
- **Brak walidacji**
  Ryzyko niepoprawnych danych w systemie.
- **Zbyt dużo parametrów query**
  Warto rozważyć użycie obiektu filtrującego (DTO).

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- Spring MVC wiąże dane z żądania do argumentów metody za pomocą adnotacji.
- Używaj @PathVariable dla wartości ze ścieżki URL.
- Używaj @RequestParam dla parametrów query.
- Używaj @RequestHeader do odczytu nagłówków HTTP.
- Używaj @RequestBody do danych w formacie JSON/XML.
- Spring automatycznie konwertuje typy i wspiera walidację (@Valid).
- Dobierz odpowiednie źródło danych do konkretnego przypadku.
- Dzięki poprawnemu wiązaniu dane trafiają tam, gdzie ich potrzebujesz!



MAPA 25/100

WALIDACJA DANYCH WEJŚCIOWYCH

Jak sprawdzać poprawność danych w Spring MVC i zwracać czytelne błędy

Lepsze dane dziś. Mniej błędów jutro!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO WALIDACJA?

CZYSTSZE DANE, BEZPIECZNIEJSZA APLIKACJA

- Zapewnia wysoką jakość danych w aplikacji.
- Chroni przed niepoprawnymi i potencjalnie szkodliwymi danymi.
- Ułatwia obsługę błędów i zwraca czytelne komunikaty.
- Jest niezbędna dla poprawnego działania API i lepszego UX.

2 JAK TO DZIAŁA?

JAK SPRING WYKONUJE WALIDACJĘ

- Używamy **Jakarta Bean Validation** (np. Hibernate Validator).
- Spring automatycznie integruje walidację z MVC i REST.
- Podczas obsługi żądania dane są sprawdzane zgodnie z adnotacjami.
- Jeśli dane są niepoprawne, rzucany jest wyjątek lub wypełniany jest BindingResult z błędami.

3 NAJCZĘSTSZE ADNOTACJE

PODSTAWOWE OGRANICZENIA WALIDACJI

@NotNull	Wartość nie może być null.
@NotBlank	Tekst nie może być pusty (usuwa spacje).
@Size	Określa minimalną i maksymalną długość (np. napisu, kolekcji).
@Email	Musi być poprawnym adresem e-mail.
@Min	Minimalna wartość (np. liczby).
@Max	Maksymalna wartość (np. liczby).
@Pattern	Musi pasować do wyrażenia regularnego (regex).

4 @Valid i @Validated

PODOBIEŃSTWA I RÓŻNICE

@Valid

- Standardowa adnotacja Jakarta Bean Validation.
- Używana najczęściej w kontrolerach (np. @RequestBody, @ModelAttribute).
- Obsługuje podstawową walidację.

@Validated

- Adnotacja od Springa.
- Dodatkowo obsługuje grupy walidacji (np. tworzenie vs. aktualizacja).
- Może być używana na klasach (np. do walidacji parametrów).

5 WALIDACJA DTO

OBIEKTY W ŻĄDANIACH HTTP

- Stosujemy adnotacje w klasach DTO (np. request body).
- Używamy @Valid przy parametrze @RequestBody w kontrolerze.
- Spring automatycznie waliduje obiekt i zwraca błędy.
- To najczęstszy sposób walidacji danych w aplikacjach REST API.

DTO (Data Transfer Object) to najlepsze miejsce na reguły walidacji, a nie encje JPA.

6 WALIDACJA PARAMETRÓW

PARAMETRY ŚCIEŻKI, ZAPYTAŃ I INNE

- Możemy walidować parametry ścieżki (@PathVariable) i zapytania (@RequestParam).
- Używamy adnotacji bezpośrednio na parametrach (np. @Min, @NotBlank).
- Na klasie kontrolera stosujemy @Validated (wymagane do walidacji parametrów).
- Spring automatycznie sprawdza wartości i zwraca błędy, jeśli są niepoprawne.

Przykład:

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable @Min(1) Long id) { ... }
```

7 OBSŁUGA BŁĘDÓW WALIDACJI

JAK PRZECHWYTYWAĆ I ZWRACAĆ BŁĘDY

MethodArgumentNotValidException

- Rzucany, gdy walidacja @RequestBody zakończy się niepowodzeniem.
- Zawiera listę błędów (FieldError).
- Można obsłużyć globalnie @ControllerAdvice.

BindingResult

- Alternatywa dla wyjątków.
- Dodajemy jako parametr po obiekcie @Valid.
- Pozwala ręcznie sprawdzić błędy i zbudować własną odpowiedź.

8 PRAKTYCZNY PRZYKŁAD

DTO I KONTROLER + PRZYKŁAD ODPOWIEDZI

```
// DTO z adnotacjami walidacji
public class CreateUserRequest {
  @NotBlank(message = "Imię jest wymagane")
  private String firstName;
  @NotBlank(message = "Email jest wymagany")
  @Email(message = "Nieprawidłowy format email")
  private String email;
  @Size(min = 6, max = 20,
      message = "Hasło musi mieć 6-20 znaków")
  private String password;
  @Min(value = 18, message = "Musisz mieć 18 lat")
  private Integer age;
}
// Kontroler
@PostMapping("/users")
public ResponseEntity<?> createUser(
  @Valid @RequestBody CreateUserRequest request) {
  return ResponseEntity.ok().build();
}
```

Przykład odpowiedzi z błędami (JSON):

```
{
  "status": 400,
  "errors": [
    {
      "field": "email",
      "message": "Nieprawidłowy
        format email"
    },
    {
      "field": "age",
      "message": "Musisz
        mieć 18 lat"
    }
  ]
}
```

9 DOBRE PRAKTYKI

SPRAWDDONE ZASADY

- Trzymaj reguły walidacji w DTO, a nie w encjach.
- Stosuj czytelne komunikaty błędów (message).
- Zwracaj ustrukturyzowane odpowiedzi z błędami.
- Używaj grup walidacji przy różnych scenariuszach (np. tworzenie/edycja).
- Obsługuj błędy globalnie (@ControllerAdvice).
- Testuj walidację w testach automatycznych.

10 PUŁAPKI

CZEGO UNIKAĆ?

- Walidacja encji JPA zamiast DTO.
- Zapomnienie o @Valid / @Validated.
- Zbyt ogólne komunikaty błędów.
- Brak obsługi wyjątków – surowe błędy dla klienta.
- Walidacja tylko po stronie frontendu (zawsze waliduj również po stronie serwera).

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- Używaj Jakarta Bean Validation z Spring.
- Waliduj DTO, parametry i dane wejściowe.
- Stosuj odpowiednie adnotacje (@NotBlank, @Size itd.).
- Obsługuj błędy i zwracaj czytelne komunikaty.
- Przestrzegaj dobrych praktyk i unikaj typowych pułapek.

Poprawne dane to stabilniejsza aplikacja i zadowoleni użytkownicy!



MAPA 26/100

DTO, ENTITY I RECORD

Jak oddzielać warstwy i nie mieszać modelu domenowego z danymi wejścia i wyjścia

Czyste warstwy. Lepszy kod. Łatwiejszy rozwój!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 ENTITY

MODEL PERSISTENCJI

- Encja (entity) to model domenowy mapowany na bazę danych za pomocą JPA (np. @Entity).
- Reprezentuje stan w bazie danych i jest powiązana z tabelami.
- Zawiera adnotacje JPA (np. @Entity, @Table, @Id).
- Nie powinna być używana direbtt w API (np. w kontrolerach).

2 DTO

OBIEKT TRANSPORTU DANYCH

- DTO (Data Transfer Object) służy do przenoszenia danych między warstwami, szczególnie w API.
- Określa jakie dane są przyjmowane lub zwracane przez kontrolery (np. żądania i odpowiedzi).
- Pozwala ukryć wewnętrzną strukturę encji i chronić wrażliwe dane.
- Może mieć różne wersje dla różnych przypadków użycia (np. Create, Update, Response).

3 RECORD

ZWIĘZŁY, NIEZMIENNY NOŚNIK DANYCH

- **Record** (od Java 16) to zwięzła postać klasy służąca do przechowywania danych.
- Automatycznie generuje konstruktor, gettery, equals(), hashCode() i toString().
- Jest niezmienny (pola są final), idealny do prostych DTO.
- Świetnie sprawdza się w danych wejścia/wyjścia w API.

4 DLACZEGO ROZDZIELAĆ WARSTWY?

KORZYŚCI I DOBRE PRAKTYKI

- Większe bezpieczeństwo – nie ujawniasz wewnętrznego modelu domenowego.
- Czytelniejszy kod i wyraźny podział odpowiedzialności.
- Możliwość niezależnego rozwoju modelu domenowego i kontraktu API.
- Łatwiejsze wprowadzanie zmian i wersjonowanie API.

5 WEJŚCIE vs WYJŚCIE

RÓŻNE DTO DLA RÓŻNYCH POTRZEB

Request DTO (dane wejściowe)	Response DTO (dane wyjściowe)
Dane od klienta	Dane do klienta
Walidacja (@Valid)	Zwykle inny zestaw pól
Tylko potrzebne pola	Może zawierać dane zagregowane
Brak pól technicznych (np. id, createdAt)	Bez wrażliwych informacji

6 MAPOWANIE

JAK PRZEKSZTAŁCAĆ DANE?

- Mapowanie między encjami a DTO można pisać ręcznie lub użyć biblioteki **MapStruct**.
- **MapStruct** automatycznie generuje kod mapujący na etapie kompilacji.
- Mapowanie zwykle odbywa się w warstwie serwisu.
- Unikaj logiki mapowania w kontrolerach – trzymaj się zasady pojedynczej odpowiedzialności.

7 PRZYKŁAD PRZEPŁYWU

OD ŻĄDANIA DO ODPOWIEDZI

CreateUserRequest
(żądanie z API)
↓
UserService
(mapowanie, logika biznesowa)
↓
UserEntity
(zapis do bazy danych)
↓
UserResponse
(odpowiedź do klienta)

8 KIEDY RECORD SIĘ SPRAWDZA?

NAJLEPSZE ZASTOSOWANIA

- Proste, niezmienne DTO (np. żądania i odpowiedzi w API).
- Gdy potrzebujesz tylko przechowywać dane – bez dodatkowej logiki.
- Świetny do małych obiektów o kilku polach.
- Nie zastępuje encji ani klas z logiką biznesową.

9 PUŁAPKI

CZEGO UNIKAĆ?

- Wystawianie encji bezpośrednio w API.
- Ujawnianie wewnętrznych pól (np. haseł, znaczników czasowych, relacji).
- Problemy z lazy loading przy serializacji encji (np. LazyInitialization Exception).
- Używanie jednej klasy (np. encji) wszędzie – brak separacji odpowiedzialności.

10 W PRAKTYCE

SPRAWDZONE WSKAZÓWKI

- Twórz osobne DTO dla żądań i odpowiedzi.
- Waliduj dane wejściowe (@Valid, adnotacje walidacji).
- Mapuj dane w warstwie serwisu (ręcznie lub MapStruct).
- Nie zwracaj encji z kontrolerów.
- Używaj record do prostych, niezmiennych DTO.
- Dbaj o spójność i wersjonowanie kontraktu API.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Entity – model bazy danych (JPA).
- DTO – dane wejścia/wyjścia w API.
- Record – zwięzły, niezmienny DTO.
- Rozdzielanie warstw zwiększa bezpieczeństwo, czytelność i elastyczność.
- Mapowanie realizuj w warstwie serwisu.
- Unikaj najczęstszych pułapek (encje w API, lazy loading, jedna klasa do wszystkiego).

PRZYKŁAD KODU

ENTITY vs REQUEST vs RESPONSE

```
// Encja JPA
@Entity
public class UserEntity {
    @Id @GeneratedValue
    private Long id;
    private String username;
    private String email;
}

// DTO żądania (record)
public record CreateUserRequest(
    String username,
    String email) {}

// DTO odpowiedzi (record)
public record UserResponse(
    Long id,
    String username,
    String email) {}
```

Różne modele. Wspólny cel – lepsze API!



MAPA 27/100

JSON I JACKSON 3 W SPRING BOOT 4

Jak Spring serializuje i deserializuje dane oraz co warto rozumieć w praktyce

Te same dane. Większe możliwości!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 ROLA JACKSONA

KONWERSJA OBIEKTÓW ↔ JSON

- Jackson to domyślna biblioteka do obsługi JSON w Spring MVC i Spring Boot.
- Automatycznie konwertuje obiekty Java na JSON i z JSON na obiekty Java.
- Wykorzystywany w kontrolerach REST (@RestController, @RespouseBody).
- Dzięki Jacksonowi łatwo wymieniamy dane między aplikacjami (np. frontend ↔ backend).

2 SERIALIZACJA

OBIEKT JAVA → JSON

- Proces zamiany obiektu Java na tekst w formacie JSON.
- Spring automatycznie serializuje obiekty zwracane z kontrolerów (do odpowiedzi HTTP).
- Uwzględnia adnotacje, nazwy pól i konfigurację ObjectMapper.
- Wynikiem jest czytelny JSON, który może zostać przesłany do klienta (np. przeglądarki, aplikacji mobilnej).

3 DESERIALIZACJA

JSON → OBIEKT JAVA

- Proces zamiany danych JSON na obiekt Java.
- Spring automatycznie deserializuje dane z żądań HTTP (np. @RequestBody).
- Wymaga dopasowania struktury JSON do pól w klasie Java (nazwy, typy).
- Obsługuje zagnieżdżone obiekty, kolekcje oraz walidację danych (np. @Valid).

4 OBJECTMAPPER

CENTRALNY PUNKT KONFIGURACJI

- ObjectMapper to główna klasa w bibliotece Jackson.
- Pozwala dostosować sposób serializacji i deserializacji.
- Można konfigurować moduły, formaty dat, nazewnictwo, widoczność pól i wiele innych ustawień.
- Spring Boot automatycznie konfiguruje ObjectMapper, ale można dodać własne ustawienia (np. @Bean).

5 TYPOWE PRZYPADKI

CO WARTO OBSŁUŻYĆ?

- **Daty i czas** – np. LocalDate, LocalDateTime (format ISO-8601).
- **Enumy** – domyślnie nazwa, można użyć własnych wartości.
- **Wartości null** – można je pomijać (@JsonInclude).
- **Nazewnictwo pól** – np. snake_case vs camelCase (PropertyNamingStrategy).
- **Kolekcje i mapy** – listy, zbiory, mapy oraz zagnieżdżone struktury JSON.

6 ADNOTACJE

NAJCZĘŚCIEJ UŻYWANE

@JsonProperty	Określa nazwę pola w JSON (np. "user_name").
@JsonIgnore	Pomija pole w serializacji i deserializacji.
@JsonFormat	Pozwala określić format danych (np. daty).
@JsonInclude	Określa, kiedy pole ma być uwzględniane (np. tylko, gdy nie jest null).

7 JACKSON 3 W BOOT 4

NOWA WERSJA, NOWE MOŻLIWOŚCI

- Spring Boot 4 jest dostosowany do Jackson 3.
- Jackson 3 wprowadza zmiany i ulepszenia w stosunku do wersji 2.x (m.in. aktualizacje API, poprawki bezpieczeństwa).
- Jeśli używasz własnych modułów lub niestandardowej konfiguracji (ObjectMapper, własne serializery), sprawdź ich kompatybilność z Jackson 3.

8 PRAKTYCZNY PRZYKŁAD

DTO I PRZYKŁADOWY JSON

UserDto.java — Java

```java
import com.fasterxml.jackson.annotation.*;
import java.time.LocalDate;
public class UserDto {
    @JsonProperty("user_name")
    private String name;
    private int age;
    @JsonFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;
    @JsonInclude(JsonInclude.Include.NON_NULL)
    private String email;
    // konstruktory, gettery, settery
}
```

Przykładowy JSON — JSON

```json
{
  "user_name": "Anna Kowalska",
  "age": 25,
  "birthDate": "1999-05-20",
  "email": "anna@example.com"
}
```

10 NAJCZĘSTSZE BŁĘDY

CZEGO UNIKAĆ?

- Niezgodne nazwy pól między JSON a klasą.
- Błędne formaty dat (np. brak @JsonFormat).
- Brak konstruktora domyślnego (lub niewłaściwe użycie rekordów).
- Problemy z referencjami cyklicznymi (między obiektami) – mogą prowadzić do błędów serializacji.

11 DOBRE PRAKTYKI

JAK PISAĆ LEPSZY KOD?

- Używaj dedykowanych klas DTO dla API.
- Nie wystawiaj bezpośrednio encji JPA w odpowiedziach REST.
- Stosuj spójny format dat (np. ISO-8601).
- Konfiguruj ObjectMapper świadomie, gdy masz niestandardowe wymagania.
- Testuj serializację i deserializację (np. przy użyciu testów jednostkowych).

SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- Jackson konwertuje obiekty ↔ JSON.
- Spring Boot 4 używa Jackson 3.
- Konfiguracja odbywa się przez ObjectMapper.
- Warto znać kluczowe adnotacje.
- Stosuj dobre praktyki i DTO.
- Sprawdzaj kompatybilność własnych modułów.
- Unikaj typowych błędów.



Lepsze dane, lepsze API!

MAPA 28/100

OBSŁUGA BŁĘDÓW REST

Jak projektować spójne odpowiedzi błędów i centralnie obsługiwać wyjątki

Lepsza obsługa błędów, lepsze API, zadowoleni klienci!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO SPÓJNA OBSŁUGA BŁĘDÓW?

PRZEWIDYWALNOŚĆ I LEPSZE DOŚWIADCZENIE

- Klienci API wiedzą, czego się spodziewać (ten sam format odpowiedzi).
- Łatwiejsza obsługa błędów po stronie klienta (frontend, aplikacje mobilne, integracje).
- Lepsza komunikacja o problemach i szybsze rozwiązywanie incydentów.
- Profesjonalne i przewidywalne API buduje zaufanie.

2 TYPOWE ŹRÓDŁA BŁĘDÓW

GDZIE NAJCZĘŚCIEJ POJAWIAJĄ SIĘ WYJĄTKI?

- **Walidacja danych**
  Np. niepoprawne dane wejściowe, naruszenie ograniczeń (@Valid).
- **Reguły biznesowe**
  Np. niedozwolona operacja, konflikt danych, błędny stan.
- **Brak zasobu (Not Found)**
  Np. nie znaleziono użytkownika, zamówienia, produktu (404).
- **Błędy po stronie serwera**
  Np. problemy z bazą danych, zewnętrznym serwisem (500).

3 @ExceptionHandler

LOKALNA OBSŁUGA WYJĄTKÓW W KONTROLERZE

- Adnotacja **@ExceptionHandler** pozwala obsłużyć wyjątki w obrębie jednego kontrolera.
- Można zdefiniować metody obsługi dla różnych typów wyjątków.
- Dobre rozwiązanie dla specyficznych przypadków w danym kontrolerze.
- Dla wielu kontrolerów lepiej użyć **@ControllerAdvice**.

4 @ControllerAdvice / @RestControllerAdvice

CENTRALNA OBSŁUGA WYJĄTKÓW

- Obsługuje wyjątki globalnie dla całej aplikacji (wszystkich kontrolerów).
- **@RestControllerAdvice** to wersja dla aplikacji REST (zwraca JSON).
- Pozwala na spójny format błędów i łatwiejsze utrzymanie kodu.
- Można obsługiwać zarówno własne wyjątki, jak i wyjątki Springa (np. MethodArgumentNotValidException).

5 ResponseEntity i statusy

NAJCZĘŚCIEJ UŻYWANE KODY HTTP

400	BAD_REQUEST	Niepoprawne dane wejściowe (np. błąd walidacji).
404	NOT_FOUND	Nie znaleziono zasobu (np. użytkownik nie istnieje).
409	CONFLICT	Konflikt danych (np. użytkownik już istnieje).
500	INTERNAL_SERVER_ERROR	Nieoczekiwany błąd po stronie serwera.

W Springu używamy **ResponseEntity** lub odpowiednich adnotacji, aby zwrócić właściwy kod HTTP i treść błędu.

6 ProblemDetail

NOWOCZESNA STRUKTURA BŁĘDÓW (SPRING 6+/BOOT 3+)

- **ProblemDetail** to wbudowana klasa w Spring 6+, zgodna ze standardem **RFC 7807** (Problem Details for HTTP APIs).
- Zawiera ustandaryzowane pola: type, title, status, detail, instance.
- Ułatwia tworzenie spójnych i czytelnych odpowiedzi błędów.
- Można rozszerzać o własne pola, jeśli potrzebujemy dodatkowych informacji (np. traceId).

7 PRAKTYCZNY MODEL BŁĘDU

PROPONOWANA STRUKTURA ODPOWIEDZI

timestamp	Data i czas wystąpienia błędu
status	Kod HTTP
error	Krótka nazwa błędu (np. "Not Found")
message	Opis błędu (czytelny dla klienta)
path	Ścieżka żądania, które spowodowało błąd
traceId	Identyfikator żądania (do logów)

8 PRZYKŁAD

OBSŁUGA WYJĄTKU NotFoundException

```
@RestControllerAdvice
public class GlobalExceptionHandler {

   @ExceptionHandler(NotFoundException.class)
   public ResponseEntity<ProblemDetail>
     handleNotFound(NotpServleException ex,
                    HttpServletRequest request) {
      ProblemDetail problem = ProblemDetail
         .forStatus(HttpStatus.NOT_FOUND);
      problem.setTitle("Nie znaleziono zasobu");
      problem.setDetail(ex.getMessage());
      problem.setProperty("path", request.getRequestURI());
      problem.setProperty("traceId", getTraceId());

      return ResponseEntity.status(HttpStatus.NOT_FOUND)
         .body(problem);
   }
}
```

9 PRZYKŁADOWA ODPOWIEDŹ JSON

TAK WYGLĄDA BŁĄD 404 DLA KLIENTA

```
{
  "timestamp": "2024-01-15T10:30:45Z",
  "status": 404,
  "error": "Not Found",
  "message": "Użytkownik o podanym ID
           nie został znaleziony.",
  "path": "/api/uzytkownicy/123",
  "traceId": "a1b2c3d4-5678-90ab-cdef-1234567890ab"
}
```

Taki format jest czytelny, spójny i zawiera wszystkie najważniejsze informacje dla klienta API.

10 PUŁAPKI

CZEGO UNIKAĆ?

- **Ujawnianie wewnętrznych szczegółów**
  Nie zwracaj stack trace, zapytań SQL, ścieżek plików itp.
- **Niespójne formaty odpowiedzi**
  Używaj jednego, zdefiniowanego modelu błędu.
- **Nieprawidłowe kody statusu**
  Zwracaj odpowiednie kody HTTP (np. 404, a nie 200).
- **Zbyt ogólne komunikaty**
  Unikaj komunikatów typu „Wystąpił błąd".
- **Brak identyfikatora błędu**
  Warto dodawać traceId do korelacji z logami.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE INFORMACJE

- Stosuj @ControllerAdvice do centralnej obsługi.
- Zwracaj odpowiednie kody HTTP.
- Używaj **ProblemDetail** (lub własnego modelu).
- Nie ujawniaj wrażliwych informacji.
- Dbaj o spójny format odpowiedzi.
- Dodawaj traceId do każdej odpowiedzi.
- Testuj scenariusze błędów.
- Dobrze opisane błędy = lepsze API!

Błędy zdarzają się w każdej aplikacji. Ważne, jak je obsługujesz!

Dobre API to także dobre komunikaty o błędach!



MAPA 29/100

FILTER VS INTERCEPTOR VS CONTROLLERADVICE

Gdzie w łańcuchu żądania działa każde z tych narzędzi i do czego służy

Różne warstwy, ten sam cel – lepsze aplikacje!

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 FILTER

WARSTWA SERWLETÓW (PRZED SPRING MVC)

- Działa w warstwie serwletów (jako jaxax/jakarta.servlet.Filter).
- Uruchamia się przed Spring MVC i działa dla wszystkich żądań.
- Może modyfikować żądanie i odpowiedź (request/response wrapper).
- Typowe zastosowania to CORS, podstawowa autoryzacja, logowanie, kompresja, nagłówki bezpieczeństwa.

2 HANDLERINTERCEPTOR

W RAMACH SPRING MVC (WOKÓŁ KONTROLERA)

- Działa w Spring MVC, pomiędzy otrzymaniem żądania a wywołaniem kontrolera.
- Pozwala wykonać logikę przed wywołaniem kontrolera, po jego zakończeniu i po wyrzuceniu wyjątku.
- Ma dostęp do informacji o handlerze (metodzie kontrolera) i parametrach żądania.
- Typowe zastosowania to autoryzacja na poziomie endpointów, logowanie, pomiar czasu, sprawdzanie uprawnień.

3 CONTROLLERADVICE

LOGIKA PRZEKROJOWA DLA KONTROLERÓW

- Działa w warstwie Spring MVC, dla kontrolerów REST/MVC.
- Służy głównie do globalnej obsługi wyjątków (@ExceptionHandler).
- Może obsługiwać również wiązanie danych i walidację (@InitBinder).
- Umożliwia centralne zarządzanie błędami i jednolite odpowiedzi dla całej aplikacji.

4 GDZIE DZIAŁA KAŻDE Z NICH?

ŁAŃCUCH PRZETWARZANIA ŻĄDANIA



5 TYPOWE ZASTOSOWANIA FILTER

- Obsługa CORS (nagłówki)
- Podstawowa autoryzacja (np. token w nagłówku)
- Logowanie wszystkich żądań i odpowiedzi
- Kompresja, modyfikacja nagłówków
- Obsługa żądań statycznych, ścieżek publicznych

6 TYPOWE ZASTOSOWANIA INTERCEPTOR

- Sprawdzanie autoryzacji na poziomie endpointów
- Pomiar czasu wykonania żądania
- Logowanie informacji o kontrolerze i parametrach
- Wielotenantowość, wybór kontekstu
- Modyfikacja modelu lub atrybutów żądania

7 TYPOWE ZASTOSOWANIA CONTROLLERADVICE

- Globalna obsługa wyjątków i mapowanie na odpowiednie kody HTTP
- Obsługa błędów walidacji (MethodArgumentNotValid)
- Inicjalizacja binderów (@InitBinder)
- Jednolity format odpowiedzi z błędami
- Obsługa specyficznych wyjątków biznesowych

8 PORÓWNANIE

Cecha	Filter	Interceptor	ControllerAdvice
Warstwa	Servlet	Spring MVC	Spring MVC
Dostęp do ciała żądania	Tak (pełny)	Ograniczony	Nie (dane już zbindowane)
Związek z kontrolerami	Brak	Tak (zna handlera)	Tak (dla kontrolerów)
Główne zastosowanie	CORS, logowanie, bezpieczeństwo	Autoryzacja, logowanie, pomiar czasu	Obsługa wyjątków, walidacja, bindowanie

9 PRAKTYCZNY PRZYKŁAD

LOGUJĄCY FILTER + INTERCEPTOR AUTORYZACJI + GLOBALNE OBSŁUGI WANIE WYJĄTKÓW

LogFilter.java

```
@Component
public class LogFilter
    implements Filter {

  @Override
  public void doFilter(
     ServletRequest req,
     ServletResponse res,
     FilterChain chain),
     throws IOException,
          ServletException {
   System.out.println(
    "[FILTER] " +
    ((HttpServletRequest) req)
    .getRequestURI());
  chain.doFilter(req, res);
  }
}
```

AuthInterceptor.java

```
@Component
public class AuthInterceptor
   implements HandlerInterceptor {

 @Override
 public boolean preHandle(
    HttpServletRequest request,
    HttpServletResponse response,
    Object handler) throws Exception {
   String token = request.getHeader(
     "Autherization");
   if (token == null) {
    response.sendError(401, "Brak tokena");
    return false;
   }
  return true;
  }
}
```

GlobalExceptionHandler.java

```
@RestControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(Exception.class)
  public ResponseEntity<Map<String, Object>>
     handleException(Exception ex) {
Map<String, Object> body = new HashMap<>();
body.put("error", ex.getMessage());
body.put("timestamp",
   LocalDateTime.now());

return ResponseEntity
  .status(HttpStatus.INTERNAL_SERVER_ERROR)
  .body(body);
  }
}
```

10 PUŁAPKI

- Używanie niewłaściwego narzędzia do danego zadania.
- Dublowanie tej samej logiki w filtrze i interceptorze.
- Próba użycia interceptora poza kontekstem Spring MVC (np. w aplikacji WebFlux).
- Zbyt ciężka logika w interceptorze – może spowolnić aplikację.
- Obsługa wyjątków w filtrze zamiast w ControllerAdvice.

11 SZYBKIE PODSUMOWANIE

- Fliter dziala najwcześniej (warstwa serwletów).
- Interceptor dziala w Spring MVC (wokół kontrolera).
- ControllerAdvice obsługuje wyjątki i wiązanie danych.
- Każde z nich ma inne odpowiedzialności i się uzupełnia.
- Dobór narzędzia zależy od miejsca w łańcuchu i potrzeb aplikacji.
- Razem pozwalają budować bezpieczne, czytelne i łatwe w utrzymaniu aplikacje.



Dobrze dobrane narzędzia to czystszy kod i spokojniejszy sen!

MAPA 30/100

DOBRZE ZAPROJEKTOWANE REST API

Najważniejsze zasady projektowania czytelnych, przewidywalnych i praktycznych endpointów

Dobre API to lepsze aplikacje, zadowoleni klienci i mniej problemów! ☺

MODUŁ 03 • SPRING MVC I REST API

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO PROJEKTOWAĆ ŚWIADOMIE?

LEPSZE API = WIĘKSZA WARTOŚĆ

- Łatwiejsze utrzymanie i rozwój aplikacji.
- Lepsze doświadczenia klientów (frontend, mobile, inne zespoły).
- Czytelna i spójna dokumentacja.
- Mniejsze ryzyko błędów i nieporozumień.
- API to kontrakt — warto zaprojektować je raz, a dobrze.

2 ZASOBY I ŚCIEŻKI

RZECZOWNIKI ZAMIAST CZASOWNIKÓW

- Używaj rzeczowników (zasobów), nie czasowników.
- Stosuj liczbę mnogą dla kolekcji.
- Używaj logicznej, hierarchicznej struktury ścieżek.
- Unikaj zbyt głębokiej zagnieżdżoności.
- Stosuj czytelne, małe litery i myślniki w nazwach.

Dobra ścieżka:

```
/api/users/123/orders
```

Słaba ścieżka:

```
/api/getUserOrders?id=123
/api/userOrder/list
```

3 METODY HTTP

JASNO OKREŚLONY CEL

GET	Pobiera dane (nie zmienia stanu).
POST	Tworzy nowy zasób.
PUT	Całkowicie aktualizuje istniejący zasób.
PATCH	Częściowo aktualizuje istniejący zasób.
DELETE	Usuwa zasób.

4 KODY STATUSU

INFORMUJĄ O WYNIKU OPERACJI

200	Sukces – zwrócono dane.
201	Utworzono zasób (np. po POST).
204	Sukces – brak treści (np. po DELETE).
400	Błędne żądanie (np. niepoprawne dane).
404	Nie znaleziono zasobu.
409	Konflikt (np. duplikat, naruszenie unikalności).

5 IDEMPOTENCJA

TEN SAM EFEKT PRZY WIELOKROTNYM WYWOŁANIU

- Idempotentne operacje zawsze prowadzą do tego samego rezultatu, bez względu na liczbę wywołań.
- GET, PUT, DELETE są idempotentne.
- POST zwykle nie jest idempotentne (każde wywołanie tworzy nowy zasób).
- Idempotencja ułatwia obsługę błędów, ponowne żądania i integracje.

Przykład:

PUT /api/users/123 — wielokrotne wywołanie ustawia te same dane.

6 PAGINACJA, FILTROWANIE, SORTOWANIE

PRACA NA WIĘKSZYCH ZBIORACH DANYCH

- Używaj parametrów zapytania (query parameters).
- Stosuj spójne nazwy parametrów.
- Zwracaj metadane paginacji (np. totalElements, totalPages).
- Umożliwiaj filtrowanie po wielu polach.
- Pozwalaj na sortowanie (kierunek: asc/desc).

Przykład:

```
GET /api/users?page=0&size=20
&sort=createdAt,desc&role=ADMIN
&active=true
```

7 WERSJONOWANIE API

STABILNOŚĆ I KOMPATYBILNOŚĆ

- Wprowadzaj wersjonowanie, gdy wprowadzasz zmiany niekompatybilne wstecz.
- Najczęściej stosuje się wersję w ścieżce, np. /api/v1/...
- Alternatywnie: nagłówek (HTTP Header) lub parametr zapytania.
- Utrzymuj starsze wersje przez określony czas.

Przykłady:

```
/api/v1/users
/api/v2/users
Header: Accept: application/vnd.app.v2+json
```

8 SPÓJNOŚĆ ODPOWIEDZI

PRZEWIDYWALNA STRUKTURA DANYCH

- Używaj DTO (Data Transfer Objects).
- Stosuj spójne nazwy pól (np. camelCase).
- Zwracaj dane w jednakowym formacie (np. JSON).
- Używaj jednolitego formatu błędów (kod, komunikat, szczegóły).
- Nie ujawniaj wewnętrznych szczegółów aplikacji.

Przykład odpowiedzi błędu:

```
{
  "timestamp": "2025-01-15T10:00:00Z",
  "status": 400,
  "error": "Bad Request",
  "message": "Nieprawidłowe dane",
  "path": "/api/users"
}
```

9 BEZPIECZEŃSTWO I WALIDACJA

CHROŃ DANE I SWOJĄ APLIKACJĘ

- Waliduj dane wejściowe (np. @Valid, ograniczenia pól).
- Stosuj autoryzację (np. JWT, OAuth2).
- Nie ujawniaj wrażliwych informacji w odpowiedziach i logach.
- Ograniczaj dostęp do zasobów na poziomie uprawnień.
- Stosuj limity i ochronę przed nadużyciami (np. rate limiting).

10 PRAKTYCZNY PRZYKŁAD

PROJEKT API DLA UŻYTKOWNIKÓW I ZAMÓWIEŃ

Przykłady endpointów (users):

```
GET     /api/v1/users          # lista użytkowników
GET     /api/v1/users/123      # szczegóły
POST    /api/v1/users          # utworzenie
PUT     /api/v1/users/123      # pełna aktualizacja
PATCH   /api/v1/users/123      # częściowa aktualizacja
DELETE  /api/v1/users/123      # usunięcie
```

Przykłady endpointów (orders):

```
GET     /api/v1/orders         # lista zamówień
GET     /api/v1/orders/456     # szczegóły
POST    /api/v1/orders         # utworzenie
GET     /api/v1/users/123/orders # zamówienia użytkownika
```

11 NAJCZĘSTSZE BŁĘDY

CZEGO UNIKAĆ?

- Używanie czasowników w ścieżkach (np. /getUsers, /deleteUser).
- Niespójne kody statusu.
- Zbyt rozbudowane, „gadające" endpointy (zbyt wiele odpowiedzialności).
- Brak walidacji danych.
- Zwracanie niejednolitego formatu odpowiedzi.
- Ujawnianie wewnętrznych błędów (np. stack trace).

12 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE ZASADY

- Projektuj wokół zasobów, nie czasowników.
- Używaj odpowiednich metod HTTP i kodów statusu.
- Dbaj o idempotencję, paginację i filtrowanie.
- Stosuj wersjonowanie, gdy to konieczne.
- Zwracaj spójne i czytelne odpowiedzi.
- Waliduj dane i dbaj o bezpieczeństwo.
- Unikaj najczęstszych błędów.
- Pamiętaj – dobrze zaprojektowane API to inwestycja, która się zwraca!

DOBRE vs SŁABE NAZWY ENDPOINTÓW

Dobrze ✓		Źle ✗	
GET	/api/users	GET	/api/getUsers
GET	/api/users/123	GET	/api/user/123
GET	/api/users/123/orders	GET	/api/getUserOrders
POST	/api/orders	POST	/api/createOrder
DELETE	/api/orders/456	DELETE	/api/removeOrder/456
GET	/api/products	GET	/api/listProducts

MAPA 31/100

DATASOURCE, JDBC i CONNECTION POOL

Jak aplikacja Spring Boot łączy się z bazą danych i zarządza połączeniami

Dobra konfiguracja bazy danych to stabilna aplikacja i zadowoleni użytkownicy!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST DATASOURCE?

ABSTRAKCJA DOSTĘPU DO BAZY DANYCH

- DataSource to standardowy interfejs Java (javax.sql.DataSource) do uzyskiwania połączeń z bazą danych.
- Aplikacja nie tworzy połączeń samodzielnie – pobiera je z DataSource (np. z puli połączeń).
- Zapewnia niezależność od konkretnej implementacji (np. HikariCP, Tomcat JDBC, DBCP).
- Używany przez Spring Boot, Spring JDBC, Spring Data JPA i inne moduły.

2 JDBC W SPRING BOOT

NISKOPOZIOMOWY DOSTĘP DO DANYCH

- JDBC (Java Database Connectivity) to standard API do komunikacji z relacyjnymi bazami danych.
- W Spring Boot możesz używać klasycznego JDBC (np. JdbcTemplate) lub wyższych poziomów abstrakcji (np. Spring Data JPA).
- Spring Boot automatycznie konfiguruje DataSource, dzięki czemu JDBC działa „od razu" po dodaniu dependencji i właściwości.
- JdbcTemplate upraszcza korzystanie z JDBC i zarządza zasobami (np. zamykaniem połączeń).

3 AUTO-KONFIGURACJA DATASOURCE

SPRING BOOT ROBI TO ZA CIEBIE

- Spring Boot automatycznie konfiguruje DataSource na podstawie właściwości `spring.datasource.*`.
- Jeśli na classpath znajduje się sterownik bazy danych (np. PostgreSQL), tworzona jest instancja HikariDataSource.
- Nie musisz pisać dodatkowej konfiguracji – wystarczy dodać dependencję i ustawić właściwości.
- W razie potrzeby możesz nadpisać domyślną konfigurację, definiując własny bean DataSource.

4 WŁAŚCIWOŚCI spring.datasource.*

PODSTAWOWA KONFIGURACJA

- Wszystkie ustawienia związane z połączeniem z bazą danych znajdują się w przestrzeni nazw `spring.datasource.*`.
- Określasz m.in. URL, użytkownika, hasło oraz parametry puli połączeń.
- Spring Boot automatycznie mapuje te właściwości na obiekt DataSource (np. HikariDataSource).
- Możesz używać plików application.yml lub application.properties.

5 HIKARICP – DOMYŚLNY CONNECTION POOL

SZYBKI, LEKKI I NIEZAWODNY

- HikariCP to domyślny pool połączeń w Spring Boot (od wersji 2.x).
- Wysoka wydajność, niskie zużycie zasobów, szybkie uzyskiwanie połączeń.
- Automatycznie konfigurowany na podstawie `spring.datasource.*`.
- Możesz dostosować wiele parametrów (np. rozmiar puli, timeouts, walidację).
- W większości przypadków nie musisz zmieniać implementacji – HikariCP działa świetnie.

6 JAK DZIAŁA POOL POŁĄCZEŃ?

PONOWNE UŻYCIE POŁĄCZEŃ

1. Podczas startu aplikacji tworzona jest pula połączeń z określoną liczbą połączeń.
2. Aplikacja pobiera połączenie z puli, gdy go potrzebuje.
3. Po zakończeniu pracy połączenie wraca do puli (nie jest zamykane).
4. Pool zarządza cyklem życia połączeń, waliduje je i usuwa nieaktywne.

Dzięki puli połączeń unikasz kosztownego otwierania i zamykania połączeń dla każdego zapytania.

7 TYPOWY PRZEPŁYW: APLIKACJA → POOL → BAZA

JAK TO DZIAŁA W PRAKTYCE



- Aplikacja używa połączeń z puli, a nie tworzy ich za każdym razem.
- Pool dba o dostępność, wydajność i stabilność.
- Ty skupiasz się na logice biznesowej.

8 NAJWAŻNIEJSZE USTAWIENIA

CO WARTO SKONFIGUROWAĆ?

`url`	Adres bazy danych (JDBC URL).
`username`	Nazwa użytkownika.
`password`	Hasło (najlepiej z zmiennych środowiskowych).
`maximum-pool-size`	Maksymalna liczba połączeń w puli.
`connection-timeout`	Czas oczekiwania na połączenie (w milisekundach).

Dostosuj wartości do obciążenia aplikacji i możliwości bazy danych.

9 W PRAKTYCE – POSTGRESQL

PRZYKŁADOWA KONFIGURACJA `application.yml`

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/sklep
    username: sklep_user
    password: ${DB_PASSWORD}
    driver-class-name: org.postgresql.Driver
    hikari:
      maximum-pool-size: 10     # maks. liczba połączeń
      minimum-idle: 5           # min. liczba połączeń
      connection-timeout: 30000 # 30 sekund
      idle-timeout: 600000      # 10 minut
      max-lifetime: 1800000     # 30 minut
      pool-name: SklepHikariCP
```

Hasło najlepiej przechowywać w zmiennych środowiskowych lub w zewnętrznym systemie (sekretów), np. Vault, AWS Secrets Manager.

10 PUŁAPKI

NA CO UWAŻAĆ?

- Za mały rozmiar puli – wąskie gardło, wolne działanie.
- Za duży rozmiar puli – niepotrzebne zużycie zasobów po stronie bazy.
- Zbyt krótkie timeouty – błędy przy większym obciążeniu.
- Przechowywanie haseł w repozytorium (brak bezpieczeństwa).
- Brak monitorowania – nie wiesz, co się dzieje z połączeniami.
- Nieprawidłowy URL lub brak sterownika bazy danych.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- DataSource to standardowy sposób na uzyskiwanie połączeń z bazą danych.
- Spring Boot automatycznie konfiguruje HikariCP na podstawie właściwości.
- Pool połączeń znacząco poprawia wydajność i stabilność aplikacji.
- Warto dostosować kluczowe parametry (np. `maximum-pool-size`, `timeouts`).
- Używaj bezpiecznego przechowywania sekretów.
- Monitoruj i testuj konfigurację w warunkach zbliżonych do produkcji.



MAPA 32/100

SPRING DATA REPOSITORY

Jak działa warstwa repozytoriów i co daje Spring Data JPA

Dobre repozytoria to czystszy kod, mniej boilerplate i więcej czasu na to, co ważne!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST REPOZYTORIUM?

WARSTWA DOSTĘPU DO DANYCH

Repozytorium to interfejs zapewniający dostęp do danych w sposób uporządkowany i abstrakcyjny.

Spring Data JPA dostarcza gotowych implementacji, więc nie musisz pisać kodu obsługi bazy danych.

Działa na encjach JPA i wykorzystuje mechanizmy ORM (Hibernate).

Umożliwia wykonywanie podstawowych operacji CRUD, zapytania po nazwie metod, paginację, sortowanie i inne. To standardowy sposób dostępu do danych w aplikacjach Spring Boot.

2 CrudRepository vs PagingAndSortingRepository vs JpaRepository

KOLEJNE POZIOMY MOŻLIWOŚCI

CrudRepository<T, ID>

- podstawowe operacje CRUD
- `save()`, `findById()`, `findAll()`, `delete()`, itp.
- wystarczający do prostych aplikacji

PagingAndSortingRepository<T, ID>

- rozszerza CrudRepository
- dodaje paginację i sortowanie
- `findAll(Pageable)`, `findAll(Sort)`
- przydatny przy większych zbiorach danych

JpaRepository<T, ID>

- rozszerza PagingAndSortingRepository
- dodaje operacje specyficzne dla JPA (np. `flush()`, `saveAllAndFlush()`)
- najczęściej używany – pełna funkcjonalność

3 JAK SPRING TWORZY IMPLEMENTACJĘ INTERFEJSU

WSZYSTKO DZIAŁA AUTOMATYCZNIE

1. Definiujesz interfejs repozytorium, który rozszerza np. JpaRepository.
2. Spring podczas startu aplikacji analizuje interfejs i metody.
3. Na podstawie nazwy metod tworzy zapytania (metody query) lub używa domyślnych implementacji.
4. Generuje klasę proxy z implementacją w czasie działania (runtime).
5. Wstrzykujesz repozytorium jako bean i używasz go w serwisie – bez pisania implementacji!

4 PODSTAWOWE OPERACJE CRUD

GOTOWE METODY W REPOZYTORIUM

save(entity)	Zapisuje lub aktualizuje encję.
findById(id)	Zwraca Optional z encją.
findAll()	Zwraca wszystkie encje.
delete(entity)	Usuwa encję.
deleteById(id)	Usuwa encję po identyfikatorze.
existsById(id)	Sprawdza, czy encja istnieje.
count()	Zwraca liczbę wszystkich encji.

5 QUERY METHODS PO NAZWIE METODY

AUTOMATYCZNE GENEROWANIE ZAPYTAŃ

Spring Data analizuje nazwę metody i automatycznie tworzy zapytanie na podstawie własnych reguł.

- findByNazwisko(String nazwisko)
- findByEmailAndAktywny(String email, boolean aktywny)
- countByStatus(Status status)
- deleteByDataUtworzeniaBefore(LocalDate data)
- findTop5ByOrderByDataUtworzeniaDesc()

Nazwy metod muszą odnosić się do pól encji i używać słów kluczowych (np. `Like`, `Between`, `GreaterThan`, `OrderBy`, `Top`, `First`,...).

6 PAGINACJA I SORTOWANIE

PRACA Z DUŻYMI ZBIORAMI DANYCH

Do paginacji używamy interfejsów Pageable, Pageoraz Sort.

- findAll(Pageable pageable) – zwraca stronę wyników (Page<T>).
- findAll(Sort sort) – zwraca posortowaną listę (List<T>).

Przykład:

```
Page<User> page = userRepository
    .findAll(PageRequest.of(0, 10,
      Sort.by("nazwisko").ascending()));
```

Paginacja pozwala ograniczyć liczbę danych wczytywanych do pamięci i poprawia wydajność aplikacji.

7 ZWRACANE TYPY

RÓŻNE SPOSOBY ZWRACANIA WYNIKÓW

Optional<T>	Pojedynczy obiekt, który może nie istnieć (np. findById).
List<T>	Lista wyników (np. findAll()).
Page<T>	Strona wyników z paginacją (metadane: liczba stron, itp.).
Slice<T>	Lżejsza wersja Page – tylko informacja, czy jest kolejna strona.

8 PRAKTYCZNY PRZYKŁAD INTERFEJSU USERREPOSITORY

TYPOWE METODY W REALNEJ APLIKACJI

```
import org.springframework.data.jpa.repository.*;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository
        extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    List<User> findByNazwiskoContaining(String nazwisko);

    Page<User> findByAktywny(boolean aktywny, Pageable pageable);

    long countByRola(String rola);

    void deleteByDataUtworzeniaBefore(LocalDate data);
}
```

9 W PRAKTYCE

KIEDY UŻYWAĆ REPOZYTORIUM?

- Gdy potrzebujesz prostego i szybkiego dostępu do danych.
- Do typowych operacji CRUD.
- Gdy Twoje zapytania można wyrazić nazwą metody.
- W większości aplikacji biznesowych – to domyślny wybór.
- Bardziej złożone operacje (np. zaawansowane zapytania, agregacje) możesz realizować przez @Query, Specification, QueryDSL lub własne implementacje.
- Repozytorium używasz w serwisach, nie w kontrolerach – zachowujesz dobrą architekturę warstwową.

10 PUŁAPKI

NA CO UWAŻAĆ?

- Zbyt grube repozytorium – setki metod utrudnia utrzymanie kodu.
- Umieszczanie logiki biznesowej w repozytorium – repo powinno obsługiwać tylko dostęp do danych.
- Nieczytelne nazwy metod – trudne do zrozumienia i utrzymania.
- Nadmierne użycie `findAll()` – może prowadzić do problemów z wydajnością.
- Brak transakcji w warstwie serwisowej tam, gdzie jest to potrzebne.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Repozytorium to prosty i wydajny sposób dostępu do danych.
- JpaRepository daje pełną funkcjonalność (CRUD, paginacja, sortowanie).
- Metody query są generowane automatycznie na podstawie nazw metod.
- Używaj odpowiednich typów zwracanych (Optional, List, Page).
- Stosuj paginację i sortowanie przy większych zbiorach danych.
- Trzymaj logikę biznesową w serwisach.
- Dobrze zaprojektowane repozytoria to czystszy, łatwiejszy w utrzymaniu kod.



MAPA 33/100

ENCJA JPA I PERSISTENCE CONTEXT

Jak Hibernate zarządza obiektami, stanem encji i synchronizacją z bazą

Trwałe dane to fundament aplikacji! Zrozum JPA i buduj lepsze systemy!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST ENCJA?

OBIEKT MAPOWANY NA TABELĘ W BAZIE

- Encja to zwykła klasa Java mapowana na tabelę w bazie danych za pomocą JPA (najczęściej z Hibernate).
- Każda instancja encji odpowiada jednemu wierszowi w tabeli.
- Encje są zarządzane przez Persistence Context, który śledzi ich stan i synchronizuje zmiany z bazą danych.
- Dzięki JPA pracujemy na obiektach, a nie na surowym SQL.

2 KLUCZOWE ADNOTACJE

PODSTAWOWA KONFIGURACJA ENCJI

@Entity	Oznacza klasę jako encję JPA (mapowaną na tabelę).
@Table	Określa nazwę tabeli i dodatkowe ustawienia (nieobowiązkowe).
@Id	Wskazuje pole będące kluczem głównym.
@GeneratedValue	Definiuje strategię generowania klucza głównego (np. IDENTITY, SEQUENCE, AUTO).
@Column	Mapuje pole na kolumnę i pozwala skonfigurować jej właściwości (nazwa, długość, null, itp.).

3 PERSISTENCE CONTEXT

KONTEKST ZARZĄDZANIA ENCJAMI

- Persistence Context (kontekst trwałości) to mechanizm JPA, który przechowuje zarządzane encje i śledzi ich zmiany.
- Działa w obrębie pojedynczej transakcji (EntityManager) i utrzymuje tożsamość encji (tę samą encję zwraca jako ten sam obiekt).
- Wykrywa zmiany w encjach (dirty checking) i automatycznie synchronizuje je z bazą podczas flush lub commit.
- Dzięki temu nie musimy ręcznie pisać SQL – wystarczy modyfikować pola encji.

4 STANY ENCJI

CYKL ŻYCIA OBIEKTU W JPA

NEW (TRANSIENT)	Nowa encja, niezarządzana, nie ma jeszcze ID.
↓	
MANAGED	Encja zarządzana przez Persistence Context (w ramach transakcji).
↓	
DETACHED	Encja odłączona od kontekstu (po zakończeniu transakcji lub serialize).
↓	
REMOVED	Encja oznaczona do usunięcia. Zostanie usunięta z bazy po flush/commit.

5 DIRTY CHECKING

AUTOMATYCZNE WYKRYWANIE ZMIAN

- Hibernate automatycznie wykrywa zmiany w zarządzanych encjach (porównuje stan pól).
- Jeśli pole encji zostało zmienione, Hibernate generuje odpowiedni SQL (UPDATE) podczas flush lub commit.
- Nie musimy ręcznie wywoływać update – wystarczy modyfikować pola obiektu.
- Działa tylko dla encji zarządzanych (managed).

6 ENTITYMANAGER – NAJWAŻNIEJSZE OPERACJE

PODSTAWOWE METODY PRACY Z ENCJAMI

persist(entity)	Dodaje nową encję do kontekstu. Po flush/commit zostanie zapisana w bazie (INSERT).
find(Class, id)	Pobiera encję po identyfikatorze. Zwraca zarządzaną encję (lub null, jeśli nie istnieje).
merge(entity)	Scalą stan encji (np. odłączonej). Zwraca zarządzaną kopię encji z wprowadzonymi zmianami.
remove(entity)	Oznacza encję do usunięcia. Po flush/commit zostanie wykonany DELETE.

7 FLUSH VS COMMIT

KIEDY ZMIANY TRAFIAJĄ DO BAZY?

FLUSH

- Wysyła zmiany do bazy danych (wykonuje SQL – INSERT/UPDATE/DELETE).
- Nie kończy transakcji.
- Może być wywołany automatycznie (np. przed zapytaniem) lub ręcznie (entityManager.flush()).

COMMIT

- Kończy transakcję.
- Wykonuje flush (jeśli nie został wcześniej wykonany).
- Zatwierdza zmiany w bazie danych (trwałe zapisanie).
- Po commicie kontekst jest czyszczony (encje stają się detached).

8 PRZYKŁAD KLASY ENCJI USER

PROSTA ENCJA Z PODSTAWOWYMI ADNOTACJAMI

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 100)
    private String username;

    @Column(nullable = false, length = 100)
    private String email;

    private LocalDateTime createdAt;

    // konstruktory, gettery, settery
}
```

9 W PRAKTYCE – CO DAJE PERSISTENCE CONTEXT

PRAKTYCZNE KORZYŚCI W APLIKACJI

- Automatyczne śledzenie zmian w encjach.
- Tożsamość encji – ten sam obiekt w obrębie jednej transakcji.
- Mniej kodu SQL, prostsza logika aplikacji.
- Spójność danych i kontrola transakcji.
- Obsługa relacji między encjami (np. @OneToMany).
- Integracja z mechanizmem cache (pierwszego poziomu – w persistence context).
- Lepsza wydajność dzięki odroczonemu wykonywaniu zapytań.

10 PUŁAPKI

NA CO UWAŻAĆ?

- Detached entity – modyfikacja encji po zakończeniu transakcji nie zostanie zapisana (trzeba użyć merge).
- Mieszanie encji z DTO – encje nie powinny być zwracane bezpośrednio do warstwy widoku (ryzyko niechcianych zmian, rekurencji).
- Nieświadome side effects – zmiana pola encji w serwisie może automatycznie wywołać UPDATE.
- Długotrwałe sesje – zbyt wiele encji w kontekście może prowadzić do problemów z pamięcią.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Encja to klasa Java mapowana na tabelę.
- Używaj kluczowych adnotacji JPA.
- Persistence Context zarządza encjami i ich stanem.
- Dirty checking automatycznie wykrywa zmiany.
- Korzystaj z metod EntityManager: persist, find, merge, remove.
- Flush wysyła zmiany, commit zatwierdza transakcję.
- Unikaj typowych pułapek i projektuj świadomie.



MAPA 34/100

RELACJE JPA

OneToOne, OneToMany, ManyToOne i ManyToMany bez chaosu

Dobre relacje to stabilna baza dla Twojej aplikacji!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO RELACJE?

LEPSZA STRUKTURA DANYCH

- Relacje pozwalają modelować rzeczywiste powiązania między encjami (tabelami).
- Dzięki nim unikasz duplikacji danych i zachowujesz spójność.
- Ułatwiają wykonywanie złożonych zapytań i raportów.
- Dobrze zaprojektowane relacje to czytelny, elastyczny i skalowalny model danych.

2 @OneToOne

JEDEN DO JEDNEGO

Jedna encja jest powiązana z dokładnie jedną inną encją. Np. User ma jedną kartę profilu (Profile).

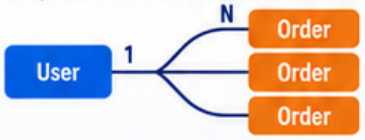



```
@Entity
class User {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "profile_id")
    private Profile profile;
}
```

3 @OneToMany

JEDEN DO WIELU

Jedna encja jest powiązana z wieloma innymi encjami. Np. User ma wiele zamówień (Order).




```
@Entity
class User {
    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "user",
               cascade = CascadeType.ALL)
    private List<Order> orders = new ArrayList<>();
}
```

4 @ManyToOne

WIELE DO JEDNEGO

Wiele encji jest powiązanych z jedną encją. Np. wiele zamówień (Order) należy do jednego użytkownika (User).




```
@Entity
class Order {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id")
    private User user;
}
```

5 @ManyToMany

WIELE DO WIELU

Wiele encji jest powiązanych z wieloma innymi encjami. Np. User ma wiele ról (Role), a rola może należeć do wielu użytkowników.




```
@Entity
class User {
   @Id @GeneratedValue
   private Long id;

   @ManyToMany
   @JoinTable(
     name = "user_roles",
     joinColumns = @JoinColumn(name = "user_id"),
     inverseJoinColumns = @JoinColumn(name = "role_id")
   )
} private Set<Role> roles = new HashSet<>();
```

6 mappedBy I WŁAŚCICIEL RELACJI

KTO ZARZĄDZA RELACJĄ?

- mappedBy wskazuje stronę „odwrotną" relacji (nie jest właścicielem).
- Właściciel (strona bez mappedBy) zarządza kluczem obcym w bazie danych.
- mappedBy zapobiega tworzeniu dodatkowej tabeli pośredniej i utrzymuje spójność.

Przykład (OneToMany / ManyToOne):

```
class User {
   @OneToMany(mappedBy = "user") // strona odwrotna
   private List<Order> orders;
}

class Order {
   @ManyToOne                    // właściciel relacji
   @JoinColumn(name = "user_id")
   private User user;
}
```

7 CASCADE I ORPHANREMOVAL

AUTOMATYZACJA OPERACJI

- Cascade określa, jakie operacje na encji „rozchodzą się" na powiązane encje (PERSIST, MERGE, REMOVE itp.).
- orphanRemoval = true usuwa osierocone encje z bazy (np. usunięte z kolekcji).
- używaj ich świadomie – ułatwiają zarządzanie danymi, ale mogą prowadzić do nieoczekiwanych usunięć.

```
@OneToMany(mappedBy = "user",
           cascade = CascadeType.ALL,
           orphanRemoval = true)
private List<Order> orders = new ArrayList<>();
```

8 FETCH TYPE W RELACJACH

KIEDY ŁADOWAĆ DANE?

LAZY
- Domyślne dla @OneToMany i @ManyToMany
- Dane ładowane na żądanie
- Lepsza wydajność

EAGER
- Domyślne dla @ManyToOne i @OneToOne
- Dane ładowane od razu
- Może powodować problemy z wydajnością

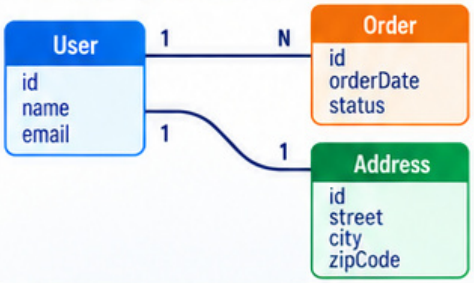
Uwaga: nadmierne użycie EAGER może prowadzić do problemu N+1 i pobierania zbyt wielu danych.

Przykład:

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```

9 PRZYKŁADOWY MODEL USER – ORDER – ADDRESS

KOMPLETNY PRZYKŁAD



- User ma wiele Order (OneToMany)
- Order należy do User (ManyToOne)
- User ma jeden Address (OneToOne)

10 PUŁAPKI

NA CO UWAŻAĆ?

- Cykliczna serializacja JSON (np. StackOverflowError) – używaj @JsonManagedReference /@JsonBackReference lub @JsonIgnore.
- Zbyt wiele relacji EAGER – problemy z wydajnością, N+1.
- Chaos w określaniu właściciela relacji – niepoprawne mapowanie, duplikaty danych.
- Nieuważne usuwanie encji – brak orphanRemoval lub złe cascade może zostawić „śmieci" w bazie.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Relacje odwzorowują rzeczywiste powiązania między danymi.
- Używaj odpowiednich adnotacji: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.
- Pamiętaj o mappedBy i właścicielu relacji.
- Stosuj cascade i orphanRemoval tam, gdzie to ma sens.
- Domyślnie używaj LAZY, kontroluj EAGER.
- Unikaj pułapek: cykliczna serializacja, N+1, niepoprawne mapowanie.
- Testuj model na rzeczywistych scenariuszach.



MAPA 35/100

QUERY METHODS, JPQL I NATIVE QUERY

Trzy sposoby pobierania danych i kiedy używać każdego z nich

Trzy sposoby na dane – wybierz ten, który najlepiej pasuje do Twojego problemu!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 QUERY METHODS – PODSTAWOWA IDEA

AUTOMATYCZNE ZAPYTANIA NA PODSTAWIE NAZW

- Query methods to najprostszy sposób pobierania danych w Spring Data JPA.
- Spring analizuję nazwę metody i automatycznie generuje odpowiednie zapytanie SQL.
- Nie musisz pisać żadnego JPQL ani SQL – wystarczy odpowiednia nazwa metody.
- Świetne do prostych zapytań i typowych przypadków.

2 PRZYKŁADY NAZW METOD

CZYTELNIE I WYRAŹNIE

- `findByEmail(String email)` – wyszukuje po adresie e-mail
- `findByStatus(OrderStatus status)` – wyszukuje po statusie
- `findByStatusAndCreatedAtAfter(OrderStatus status, LocalDateTime date)` – wiele warunków
- `findTop10ByOrderByCreatedAtDesc()` – ograniczenie liczby wyników
- `findDistinctByUserId(Long userId)` – unikalne wyniki

Przykład:

```
List<User> findByEmail(String email);
List<Order> findByStatusAndCreatedAtAfter
    (OrderStatus status, LocalDateTime date);
```

3 JPQL – ZAPYTANIA PO ENCJACH

PRACUJESZ NA OBIEKTACH, A NIE TABELACH

- JPQL (Java Persistence Query Language) pozwala pisać zapytania w oparciu o encje i ich pola, a nie nazwy tabel.
- Jest niezależny od bazy danych (i przenośny między systemami).
- Obsługuje złożone zapytania, złączenia, projekcje i funkcje agregujące.
- Używaj JPQL, gdy potrzebujesz większej kontroli niż daje nazwa metody.

Przykład:

```
@Query("SELECT u FROM User u WHERE u.status = :status")
List<User> findByStatus(@Param("status") UserStatus status);
```

4 @QUERY I PARAMETRY

PEŁNA KONTROLA NAD ZAPYTANIEM

- Adnotacja @Query pozwala zdefiniować własne zapytanie JPQL lub SQL.
- Używaj @Param do przekazywania parametrów do zapytania.
- Możesz zwracać encje, projekcje (DTO) lub pojedyncze wartości.
- Sprawdza się w bardziej złożonych scenariuszach niż query methods.

Przykład (JPQL):

```
@Query("SELECT u FROM User u
        WHERE u.email = :email")
Optional<User> findByEmail(@Param("email")
               String email);
```

5 NATIVE QUERY – KIEDY MA SENS?

ZAPYTANIA W NATYWNYM SQL

- Native query pozwala pisać zapytania w czystym SQL.
- Używaj, gdy potrzebujesz funkcji specyficznych dla danej bazy danych (np. CTE, funkcje okna, JSON, full-text).
- Jest mniej przenośny, ale daje pełną moc i najczęściej lepszą wydajność w złożonych przypadkach.
- Stosuj rozważnie, gdy JPQL nie wystarcza.

Przykład (native query):

```
@Query(value = "SELECT * FROM users
                WHERE status = :status",
                nativeQuery = true)
List<User> findByStatusNative(
           @Param("status") String status);
```

6 PORÓWNANIE PODEJŚĆ

KIEDY UŻYĆ KTÓREGO ROZWIĄZANIA?

Cecha	Query Methods	JPQL (@Query)	Native Query
Składnia	Nazwa metody	JPQL (encje)	Czysty SQL
Złożoność zapytań	Proste	Średnie i złożone	Bardzo złożone
Przenośność między bazami	Wysoka	Wysoka	Niska
Wydajność	Dobra	Dobra	Najlepsza (w specyficznych przypadkach)
Obsługa funkcji bazodanowych	Ograniczona	Średnia	Pełna
Czytelność	Bardzo wysoka	Wysoka	Niższa
Kiedy używać	Proste wyszukiwania	Większa kontrola, złączenia	Funkcje specyficzne DB, optymalizacja

7 PAGINACJA I SORTOWANIE

DUŻE ZBIORY DANYCH POD KONTROLĄ

- Spring Data JPA obsługuje paginację i sortowanie w query methods, JPQL i native query.
- Używaj typów Pageable i Sort.
- Zwraca się obiekty Page<T> lub List<T>.
- Nie pobieraj całych tabel – stosuj paginację dla lepszej wydajności.

Przykład:

```
Page<User> findByStatus(UserStatus status,
           Pageable pageable);
List<User> findByStatus(String status,
           Sort sort);
```

8 PRAKTYCZNY PRZYKŁAD REPOZYTORIUM

TRZY PODEJŚCIA W JEDNYM MIEJSCU

```
public interface UserRepository
      extends JpaRepository<User, Long> {
  // 1. Query method
  Optional<User> findByEmail(String email);
  List<User> findByStatusAndCreatedAtAfter(
     UserStatus status, LocalDateTime date);

  // 2. JPQL
  @Query("SELECT u FROM User u WHERE u.status = :status")
  List<User> findByStatus(@Param("status") UserStatus status);

  // 3. Native query
  @Query(value = "SELECT * FROM users u \
  WHERE u.email = :email", nativeQuery = true)
  Optional<User> findByEmailNative(@Param("email") String email);
}
```

9 W PRAKTYCE – WYBÓR NARZĘDZIA

DOPASUJ DO POTRZEB PROJEKTU

- Zacznij od query methods – są najprostsze i wystarczające w wielu przypadkach.
- Użyj JPQL, gdy potrzebujesz bardziej złożonych zapytań, projekcji lub złączeń.
- Sięgnij po native query, gdy potrzebujesz funkcji specyficznych dla bazy danych lub maksymalnej wydajności.
- Zawsze bierz pod uwagę czytelność, przenośność i możliwość utrzymania kodu.

10 PUŁAPKI

NA CO UWAŻAĆ?

- Zbyt długie i skomplikowane nazwy metod – trudne w utrzymaniu.
- Używanie native query bez potrzeby – ryzyko vendor lock-in.
- Pobieranie zbyt dużej ilości danych bez paginacji.
- Brak testów zapytań – mogą działać wolno lub zwracać nieoczekiwane wyniki.
- Niepoprawne mapowanie wyników przy zapytaniach native (np. kolumny, aliasy).

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Query methods – proste i szybkie zapytania bez pisania JPQL/SQL.
- JPQL – większa kontrola, praca na encjach, wysoka przenośność.
- Native query – pełna moc SQL, gdy JPQL nie wystarcza.
- Stosuj paginację i sortowanie dla dużych zbiorów danych.
- Wybieraj narzędzie adekwatnie do potrzeb, dbając o czytelność i utrzymanie kodu.



MAPA 36/100

PROJECTION, SPECIFICATION I QUERY BY EXAMPLE

Elastyczne pobieranie danych bez pisania dziesiątek metod

Mniej kodu, więcej możliwości! Elastyczne zapytania = lepsze aplikacje!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO BARDZIEJ ELASTYCZNE ZAPYTANIA?

LEPSZE DOPASOWANIE DO POTRZEB APLIKACJI

- Różne ekrany potrzebują różnych danych – nie zawsze całej encji.
- Unikasz dziesiątek metod w repozytoriach.
- Możesz dynamicznie budować warunki wyszukiwania.
- Aplikacja jest bardziej elastyczna i łatwiejsza w utrzymaniu.
- Lepsza wydajność dzięki pobieraniu tylko potrzebnych danych.

2 PROJECTION – CO ZWRACAĆ MNIEJ DANYCH

WYBIERAJ TYLKO TO, CO POTRZEBNE

- Projection pozwala pobierać tylko wybrane pola encji.
- Mniejsze zapytania = lepsza wydajność.
- Może zwracać dane jako interfejs, klasę DTO lub rekord.
- Sprawdza się w listach, podglądach, raportach.
- Nie modyfikuje danych – służy tylko do odczytu.

3 INTERFACE-BASED PROJECTION

PROSTE I EFEKTYWNE ROZWIĄZANIE

```
public interface UserSummary {
   Long getId();
   String getUsername();
   String getEmail();
}

public interface UserRepository
   extends JpaRepository<User, Long> {
 List<UserSummary> findByActiveTrue();
}
```

Wynik zapytania:

```
List<UserSummary> users =
   userRepository.findByActiveTrue();
users.forEach(u ->
   System.out.println(u.getUsername()));
```

4 DTO / CLASS PROJECTION

WIĘKSZA KONTROLA NAD DANYMI

```
public class UserDto {
  private Long id;
  private String username;
  private String email;

  public UserDto(Long id,
                 String username,
                 String email) {
      this.id = id;
      this.username = username;
      this.email = email;
   }
     // gettery
}}

public interface UserRepository
   extends JpaRepository<User, Long> {
 @Query("select new com.example.UserDto
        +"(u.id, u.username,u.email) "
        +"from User u where u.active = true")
  List<UserDto> findActiveUsers();
}
```

5 SPECIFICATION – DYNAMICZNE WARUNKI

BUDUJ ZAPYTANIA W SPOSÓB ELASTYCZNY

- Specification to sposób na dynamiczne tworzenie kryteriów zapytań w Spring Data JPA.
- Implementuje interfejs Specification<T>.
- Pozwala dodawać warunki tylko gdy są potrzebne.
- Dobrze sprawdza się przy rozbudowanych wyszukiwarkach z wieloma opcjonalnymi filtrami.
- Kodu jest mniej, a repozytorium pozostaje czytelne.

6 ŁĄCZENIE SPECIFICATION

KOMPOZYCJA WARUNKÓW

Możesz łatwo łączyć wiele specyfikacji za pomocą metod and(), or(), not().

```
public class UserSpecifications {

 public static Specification<User>
 hasUsername(String username) {
    return (root, query, cb) ->
       username == null ? null :
       cb.like(cb.lower(root.get("username")),
            "%" + username.toLowerCase() + "%");
 }

 public static Specification<User>
 isActive(Boolean active) {
    return (root, query, cb) ->
       active == null ? null :
       cb.equal(root.get("active"), active);
 }
```

7 QUERY BY EXAMPLE (QBE)

PROSTE WYSZUKIWANIE PO PRZYKŁADZIE

- QBE pozwala wyszukiwać encje na podstawie wypełnionego przykładu obiektu.
- Wykorzystuje Example i ExampleMatcher.
- Idealne do prostych, elastycznych wyszukiwarek (np. formularze).
- Obsługuje dopasowanie częściowe, ignorowanie wielkości liter, pomijanie null.
- Nie wymaga pisania zapytań ani specyfikacji.

Przykład konfiguracji:

```
User probe = new User();
probe.setUsername("jan");
ExampleMatcher matcher = ExampleMatcher
   .matchingAll()
   .withIgnoreNullValues()
   .withStringMatcher(StringMatcher.CONTAINING)
   .withIgnoreCase();
Example<User> example = Example.of(probe, matcher);
List<User> users = userRepository.findAll(example);
```

8 PRAKTYCZNY PRZYKŁAD

WYSZUKIWARKA UŻYTKOWNIKÓW

Połączenie Projection, Specification i QBE w jednym scenariuszu.

1. DTO do wyników:

```
public class UserSearchResult {
  private Long id;
  private String username;
  private String email;
  private String fullName;
  // konstruktor, gettery
}
```

2. Dynamiczne warunki (Specification):

```
Specification<User> spec = Specification
  .where(UserSpecifications.hasUsername(f.getUsername()))
  .and(UserSpecifications.hasEmail(f.getEmail())
  .and(UserSpecifications.isActive(f.getActive()));
```

3. Wykonanie zapytania z projekcją:

```
List<UserSearchResult> users =
   userRepository.findAll(spec,
      q -> q.as(UserSearchResult.class));
```

9 KIEDY UŻYWAĆ KTÓREGO PODEJŚCIA?

DOPASUJ NARZĘDZIE DO PROBLEMU

Projection	• Gdy potrzebujesz tylko wybranych pól. • Do list, podglądów, raportów. • Szybkie i proste rozwiązanie.
Specification	• Gdy masz wiele opcjonalnych filtrów. • Złożone wyszukiwanie i logika. • Dobrze komponuje się z innymi warunkami.
Query by Example	• Proste wyszukiwanie na podstawie obiektu. • Gdy nie potrzebujesz skomplikowanej logiki. • Szybkie wdrożenie, mało kodu.

10 PUŁAPKI

NA CO UWAŻAĆ?

- **Nadmierna złożoność specyfikacji** – Zbyt wiele warunków może utrudnić czytanie i utrzymanie kodu.
- **Nieczytelne zapytania** – Duża liczba połączonych specyfikacji może stać się trudna do zrozumienia.
- **Projection ≠ encja** – Projekcja nie jest encją – nie możesz jej bezpośrednio modyfikować i zapisywać.
- **Nieodpowiednie użycie QBE** – QBE nie zastąpi złożonych zapytań z relacjami i agregacjami.
- **Brak testów** – Dynamiczne zapytania warto dobrze przetestować, aby uniknąć niespodzianek.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Projection pozwala pobierać tylko potrzebne dane.
- Interface-based i DTO projection są proste w użyciu.
- Specification umożliwia dynamiczne i elastyczne budowanie warunków zapytań.
- Możesz łączyć wiele specyfikacji (and, or, not).
- Query by Example to szybki sposób na wyszukiwanie po przykładzie obiektu.
- Wybieraj podejście odpowiednie do problemu.
- Pamiętaj o wydajności, czytelności i testach.
- Elastyczne zapytania to mniej kodu i większe możliwości!

MAPA 37/100

@TRANSACTIONAL BEZ MAGII

Granice transakcji, rollback i propagation w praktyce

Transakcje to nie magia, to zasady. Poznaj je, korzystaj mądrze i unikaj problemów!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 CO DAJE TRANSAKCJA?

ATOMICZNOŚĆ, SPÓJNOŚĆ, IZOLACJA, TRWAŁOŚĆ

- Grupuje wiele operacji w jedną logiczną całość – albo wszystkie się powiodą, albo żadna.
- Zapewnia spójność danych w bazie.
- Chroni przed częściowym zapisem danych (rollback w przypadku błędu).
- Daje izolację – równoległe transakcje nie zakłócają się (w zależności od poziomu izolacji).
- Po zatwierdzeniu (commit) zmiany są trwałe.

2 @TRANSACTIONAL – GDZIE DZIAŁA?

NA METODACH W BEANACH SPRINGA

- Działa na publicznych metodach beanów zarządzanych przez Springa (@Service, @Repository, @Component).
- Tworzy proxy, które zarządza transakcją wokół wywołania metody.
- Domyślnie używa PlatformTransaction Managera (np. JpaTransactionManager dla JPA).
- Nie zadziała przy wywołaniu metody z tej samej klasy (self-invocation).
- Może być stosowany na klasie (lub interfejsie) – wtedy dotyczy wszystkich publicznych metod.

3 COMMIT I ROLLBACK

CYKL ŻYCIA TRANSAKCJI



4 RUNTIME VS CHECKED EXCEPTIONS

CO POWODUJE ROLLBACK?

RuntimeException (domyślnie rollback)

- Niesprawdzane (unchecked)
- Domyślnie powodują rollback
- Np. RuntimeException, IllegalArgumentException, DataIntegrityViolationException

Checked Exception (domyślnie brak rollbacka)

- Sprawdzane (checked)
- Domyślnie NIE powodują rollbacka
- Np. IOException, SQLException
- Można wymusić rollback: `@Transactional(rollbackFor = Exception.class)`

5 PROPAGATION

NAJWAŻNIEJSZE TYPY

REQUIRED (domyślna)

- Dołącza do istniejącej transakcji lub tworzy nową, jeśli nie ma.
- Najczęściej używana.

REQUIRES_NEW

- Zawsze tworzy nową transakcję.
- Istniejąca transakcja zostaje zawieszona.
- Przydatna np. do logowania w osobnej transakcji.

SUPPORTS

- Działa w kontekście istniejącej transakcji, jeśli jest.
- Nie tworzy nowej transakcji.
- Operacje mogą być wykonane także poza transakcją.

6 ISOLATION (SKRÓTOWO)

POZIOMY IZOLACJI TRANSAKCJI

READ_UNCOMMITTED
Pozwala na odczyt niezatwierdzonych danych (brudne odczyty).

READ_COMMITTED
Zapobiega brudnym odczytom (domyślnie w wielu bazach, np. PostgreSQL).

REPEATABLE_READ
Zapewnia powtarzalność odczytów (ten sam wynik w ramach transakcji).

SERIALIZABLE
Najwyższy poziom izolacji – pełna izolacja, kosztem wydajności.

Przykład:

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
```

7 READONLY = TRUE

TRANSAKCJA TYLKO DO ODCZYTU

- Wskazuje, że transakcja wykonuje tylko operacje odczytu.
- Może pozwolić na optymalizacje (dla bazy danych i JPA).
- Hibernate może pominąć mechanizmy śledzenia zmian (dirty checking).
- Nie powinieneś wykonywać operacji zapisu (INSERT, UPDATE, DELETE).
- To wskazówka (hint), nie twarde ograniczenie – niektóre bazy mogą ją zignorować.

Przykład:

```
@Transactional(readOnly = true)
public List<User> findAll() { ... }
```

8 PRZYKŁAD SERWISU Z @TRANSACTIONAL

REALNY PRZYKŁAD UŻYCIA

```
@Service
@RequiedArgsConstructor
public class OrderService {

  private final OrderRepository
    orderRepository;
  private final ProductService
    productService;

  @Transactional
  public void placeOrder(Long userId,
   List<Long> productIds) {
    Order order = new Order(userId);
    for (Long productId : productIds) {
      Product p = productService
       .getProductForOrder(productId);
      order.addItem(p);
    }
    orderRepository.save(order);
  }
}
```

9 W PRAKTYCE – GDZIE STAWIAĆ TRANSAKCJE?

NA WARSTWIE SERWISOWEJ

- Najczęściej na metodach w warstwie serwisowej (logika biznesowa).
- Nie na kontrolerach (warstwa web).
- Repozytoria zwykle nie wymagają @Transactional – korzystają z transakcji wyżej.
- Transakcja powinna obejmować tylko to, co jest naprawdę potrzebne.
- Projektuj granice transakcji świadomie – zbyt szeroka transakcja blokuje zasoby.

10 PUŁAPKI

NA CO UWAŻAĆ?

Self-invocation
Wywołanie metody z tej samej klasy omija proxy – @Transactional nie zadziała.

Zbyt szerokie transakcje
Długie transakcje blokują zasoby i pogarszają wydajność.

Logika zdalna w transakcji
Wywołania REST, komunikacja z innymi serwisami – mogą wydłużać transakcję i powodować problemy.

Niewłaściwe obsługiwanie wyjątków
Złapanie wyjątku i brak rethrow powoduje, że nie nastąpi rollback.

Nieprawidłowa propagacja
Nieodpowiedni typ propagation może prowadzić do nieoczekiwanego zachowania.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Transakcja zapewnia atomiczność, spójność, izolację i trwałość.
- @Transactional działa na publicznych metodach beanów Springa.
- Domyślnie rollback następuje dla RuntimeException.
- Propagation określa, jak transakcja ma się zachować względem istniejącej transakcji.
- Isolation definiuje poziom izolacji i wpływa na widoczność danych.
- readOnly=true to optymalizacja dla operacji odczytu.
- Transakcje stawiaj w warstwie serwisowej.
- Unikaj pułapek: self-invocation, zbyt szerokie transakcje, logika zdalna w transakcji.
- Dobrze zaprojektowane transakcje to stabilna i spójna aplikacja.

MAPA 38/100

LAZY, EAGER I PROBLEM N+1

Jak działa ładowanie danych i skąd biorą się kosztowne zapytania

Dobre zapytania to szybsze aplikacje, zadowoleni użytkownicy i niższe koszty!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 FETCH TYPE – CO TO JEST?

OKREŚLA SPOSÓB ŁADOWANIA POWIĄZAŃ

- Fetch type mówi, kiedy JPA ma wczytać powiązane dane (asocjacje między encjami).
- Dotyczy relacji @OneToOne, @OneToMany, @ManyToOne i @ManyToMany.
- Możemy wybrać LAZY (na żądanie) lub EAGER (od razu).
- Ma ogromny wpływ na liczbę zapytań, wydajność aplikacji i zużycie pamięci.
- Domyślne ustawienia zależą od typu relacji i mogą być zaskakujące!

2 LAZY (LENIWE ŁADOWANIE)

DANE POBIERANE SĄ NA ŻĄDANIE

- Powiązane dane nie są wczytywane od razu.
- Pobierane dopiero, gdy faktycznie ich użyjesz.
- Mniejsza liczba danych na starcie.
- Lepsza wydajność, szczególnie przy dużych kolekcjach.
- Najczęściej zalecane podejście.

Przykład:

```
@OneToMany(mappedBy = "user",
           fetch = FetchType.LAZY)
private List<Order> orders;
```

3 EAGER (CHCIWE ŁADOWANIE)

DANE POBIERANE SĄ OD RAZU

- Powiązane dane są wczytywane natychmiast, razem z encją główną.
- Może prowadzić do dużej liczby danych i niepotrzebnych zapytań.
- Proste w użyciu, ale często powoduje problemy wydajnościowe.
- Domyślnie stosowane dla @ManyToOne i @OneToOne.
- Używaj ostrożnie — tylko gdy naprawdę potrzebujesz.

Przykład:

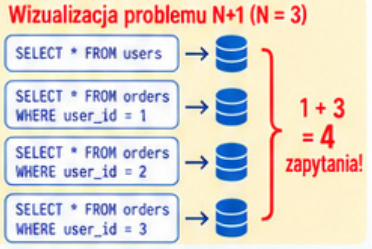
```
@ManyToOne(fetch = FetchType.EAGER)
private User user;
```

4 SKĄD BIERZE SIĘ N+1?

PROBLEM WYNIKA Z LENIWEGO ŁADOWANIA

1. Najpierw pobieramy listę encji (np. wszystkich użytkowników).
2. Dla każdej encji osobne zapytanie po powiązane dane (np. zamówienia).
3. Łącznie wykonuje się 1 + N zapytań (zamiast jednego).
4. Im więcej danych, tym większy koszt i wolniejsze działanie aplikacji.

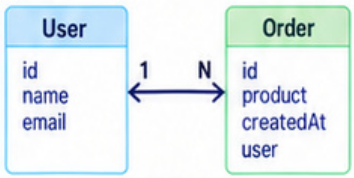
Wizualizacja problemu N+1 (N = 3)



5 PRZYKŁAD SCENARIUSZA USER – ORDER

KLASYCZNY PRZYKŁAD PROBLEMU N+1

Mamy użytkowników i ich zamówienia. Pobieramy wszystkich użytkowników, a następnie dla każdego wyświetlamy listę zamówień.



Kod (serwis):

```
List<User> users =
    userRepository.findAll();
for (User u : users ) {
    System.out.println(u.getOrders());
}
```

Wynik: 1 zapytanie po użytkowników + N zapytań po zamówienia każdego użytkownika.

6 JOIN FETCH

POBIERANIE POWIĄZANYCH DANYCH JEDNYM ZAPYTANIEM

- Używa klauzuli JOIN FETCH w JPQL.
- Ładuje powiązane encje w jednym zapytaniu.
- Eliminuje problem N+1.
- Dobre do pobierania pojedynczych kolekcji (dla wielu kolekcji mogą wystąpić duplikaty w wyniku).
- Najprostsze i najczęściej stosowane rozwiązanie.

Przykład (JPQL):

```
@Query("SELECT u FROM User u
        JOIN FETCH u.orders")
List<User> findAllWithOrders();
```

Wygenerowane SQL (uproszczone):

```
SELECT u.*, o.*
FROM users u
LEFT JOIN orders o ON o.user_id = u.id;
```

7 ENTITY GRAPH

ALTERNATYWA DLA JOIN FETCH

- Pozwala określić, które powiązania mają być wczytane.
- Można używać adnotacji @EntityGraph lub dynamicznie w zapytaniach.
- Bardziej elastyczne i wielokrotnego użytku.
- Dobrze współpracuje z repozytoriami Spring Data.
- Umożliwia ładowanie różnych grafów obiektów w zależności od potrzeb.

Przykład (adnotacja):

```
@EntityGraph(attributePaths = {"orders"})
@Query("SELECT u FROM User u")
List<User> findAllWithOrders();
```

Przykład (dynamicznie):

```
EntityGraph<User> graph = em.createEntityGraph(User.class);
graph.addAttributeNodes("orders");
Map<String, Object> hints = new HashMap<>();
hints.put("jakarta.persistence.fetchgraph", graph);
return em.createQuery("SELECT u FROM User u", User.class)
        .setHints(hints)
        .getResultList();
```

8 KIEDY LAZY, KIEDY EAGER?

PRAKTYCZNE WSKAZÓWKI

Używaj LAZY, gdy:

- Powiązane dane nie są zawsze potrzebne.
- Masz kolekcje (np. List<Order>).
- Chcesz uniknąć dużej liczby danych na starcie.
- Zależy Ci na wydajności i kontroli nad zapytaniami.

Używaj EAGER, gdy:

- Powiązane dane są prawie zawsze potrzebne.
- To relacja @ManyToOne lub @OneToOne i zestaw danych jest mały.
- Chcesz uprościć kod i uniknąć dodatkowych zapytań.

Wskazówka:

Domyślnie LAZY dla kolekcji, a EAGER dla @ManyToOne/@OneToOne. Warto jednak świadomie to zmieniać.

9 PUŁAPKI

NA CO UWAŻAĆ?

- **LazyInitializationException** – dostęp do leniwych danych poza kontekstem transakcji (brak otwartej sesji Hibernate).
- **Over-fetching** – wczytywanie niepotrzebnie dużej ilości danych.
- **Wiele kolekcji z JOIN FETCH** – może powodować duplikaty i nieoczekiwane wyniki.
- **Niejawne zapytania** – wywołanie gettera w szablonie (np. Thymeleaf) może uruchomić dodatkowe zapytania.
- **Niespodziewane ustawienia domyślne** – EAGER dla @ManyToOne i @OneToOne.

10 W PRAKTYCE

TECHNIKI OGRANICZANIA ZAPYTAŃ

- **JOIN FETCH** – najprostsze rozwiązanie.
- **Entity Graph** – elastyczne ładowanie powiązań.
- **DTO / projekcje** – pobieraj tylko potrzebne dane (np. interfejsy projekcji w Spring Data).
- **Batch Size** – @BatchSize, hibernate.default_batch_fetch_size – grupowe pobieranie.
- **Paginacja** – ogranicz ilość pobieranych danych.
- **Monitorowanie SQL** – włącz logowanie zapytań (spring.jpa.show-sql=true) i analizuj ich liczbę.
- **Indeksy** – dbaj o indeksy w bazie danych.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE ZASADY

- LAZY ładuje dane na żądanie, EAGER od razu.
- Problem N+1 powstaje przez leniwe ładowanie kolekcji.
- JOIN FETCH i Entity Graph eliminują N+1.
- Używaj LAZY domyślnie, EAGER tylko gdy ma to sens.
- Stosuj DTO, projekcje i paginację, aby ograniczać dane.
- Uważaj na LazyInitializationException i over-fetching.
- Monitoruj liczbę zapytań i analizuj wygenerowane SQL.
- Dobre zarządzanie ładowaniem danych to szybsza, bardziej stabilna i skalowalna aplikacja.



MAPA 39/100

OPTIMISTIC VS PESSIMISTIC LOCKING

Jak chronić dane przy współbieżnym dostępie i aktualizacjach

Te same dane, wielu użytkowników, zero problemów! Dobrze dobrany locking to stabilna aplikacja

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 PROBLEM WSPÓŁBIEŻNOŚCI

WIELU UŻYTKOWNIKÓW, TE SAME DANE

W aplikacji wiele użytkowników może jednocześnie odczytywać i modyfikować te same dane (np. stan zamówienia, saldo konta).

Bez odpowiedniej synchronizacji może dochodzić do utraty aktualizacji i niespójnych danych.

Problem dotyczy zarówno aplikacji webowych, jak i mikroserwisów.

Rozwiązaniem są mechanizmy blokowania (locking) na poziomie JPA i bazy danych.

2 CO TO JEST LOCKING?

MECHANIZM OCHRONY DANYCH

Locking to mechanizm kontrolujący dostęp do danych podczas ich odczytu lub modyfikacji.

Zapobiega konfliktom i utracie danych przy współbieżnym dostępie.

W JPA dostępne są dwa podejścia:

- optimistic locking (optymistyczne blokowanie)
- pessimistic locking (pesymistyczne blokowanie)

Wybór zależy od scenariusza biznesowego, częstotliwości konfliktów i wymagań wydajności.

3 OPTIMISTIC LOCKING I @VERSION

OPTYMISTYCZNE PODEJŚCIE

Zakłada, że konflikty zdarzają się rzadko – nie blokuje dostępu do danych.

Wykorzystuje pole z adnotacją @Version, które jest automatycznie zwiększane przy każdej aktualizacji.

Jeśli inny użytkownik zmienił dane, przy zapisie zostanie zgłoszony wyjątek OptimisticLockException.

Proste w użyciu, nie wpływa znacząco na wydajność – idealne dla większości aplikacji biznesowych.

4 PRZYKŁAD KONFLIKTU AKTUALIZACJI

DWÓCH UŻYTKOWNIKÓW, JEDEN REKORD

1. Użytkownik A odczytuje encję (wersja = 1).
2. Użytkownik B odczytuje tę samą encję (wersja = 1).
3. Użytkownik A aktualizuje dane i zapisuje (encja ma teraz wersję = 2).
4. Użytkownik B próbuje zapisać swoje zmiany (wciąż ma wersję = 1).

Efekt: OptimisticLockException – dane zostały zmienione przez innego użytkownika.

5 PESSIMISTIC LOCKING

PESYMISTYCZNE PODEJŚCIE

Blokuje dane w bazie na czas operacji (odczyt lub modyfikacja).

Inni użytkownicy muszą poczekać aż blokada zostanie zwolniona (komit lub rollback).

Sprawdza się w scenariuszach z dużą liczbą konfliktów lub krytycznych aktualizacjach (np. systemy finansowe).

Może obniżać wydajność – należy używać ostrożnie i z odpowiednim czasem blokady (timeout).

6 TYPY LOCKÓW W JPA

PESSIMISTIC_READ I PESSIMISTIC_WRITE

PESSIMISTIC_READ

- Blokada do odczytu (shared lock).
- Inni mogą odczytywać dane, ale nie mogą ich modyfikować.
- Przydatna, gdy często odczytujemy dane, a rzadziej je zmieniamy.

PESSIMISTIC_WRITE

- Blokada do zapisu (exclusive lock).
- Uniemożliwia innym odczyt i modyfikację danych.
- Stosowana, gdy dane są intensywnie modyfikowane lub wymagają pełnej kontroli podczas update'u.

7 KIEDY UŻYWAĆ KTÓREGO PODEJŚCIA?

DOPASUJ DO SWOJEGO SCENARIUSZA

OPTIMISTIC LOCKING	PESSIMISTIC LOCKING
Rzadkie konflikty.	Częste konflikty.
Więcej odczytów niż zapisów.	Krytyczne dane (np. finanse, stany magazynowe).
Aplikacje webowe, systemy biznesowe, CRUD.	Długie operacje aktualizacji.
Lepsza wydajność (brak blokowania).	Akceptowalny spadek wydajności w zamian za spójność danych.

8 PRAKTYCZNY PRZYKŁAD Z JPA

ENCJA Z @VERSION I REPOZYTORIUM Z @LOCK

Encja z @Version:

```
@Entity
public class Produkt {
    @Id
    private Long id;
    private String nazwa;
    private BigDecimal cena;
    @Version
    private Long wersja;
}
```

Repozytorium z @Lock:

```
public interface ProduktRepository
        extends JpaRepository<Produkt, Long> {
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("select p from Produkt p where p.id = :id")
    Optional<Produkt> znajdzDoAktualizacji(@Param("id") Long id);
}
```

9 W PRAKTYCE

KOMPROMISY WYDAJNOŚĆ VS BEZPIECZEŃSTWO

ZALETY

- Większa spójność danych.
- Ochrona przed utratą aktualizacji.
- Kontrola nad współbieżnym dostępem.
- Możliwość dostosowania strategii do scenariusza biznesowego.

WADY / KOMPROMISY

- Pessimistic locking może obniżać wydajność.
- Ryzyko zakleszczeń (deadlocków).
- Konieczność obsługi wyjątków (np. OptimisticLockException).
- Dodatkowa złożoność w kodzie i testach.

10 PUŁAPKI

NA CO UWAŻAĆ?

- **Deadlocki** – mogą wystąpić przy zbyt wielu pessymistycznych blokadach i różnych kolejnościach dostępu do danych.
- **Zbyt agresywne blokady** – nie blokuj danych dłużej niż to konieczne.
- **Brak obsługi** OptimisticLockException – może prowadzić do błędów i utraty danych.
- Mieszanie strategii locków w jednym procesie biznesowym – powoduje nieprzewidywalne zachowanie.
- Nieodpowiednie timeouty blokad – mogą zawieszać aplikację i powodować długie oczekiwanie na zasoby.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Locking chroni dane przed współbieżnymi modyfikacjami.
- Optimistic locking z @Version jest prosty i wydajny – idealny dla większości aplikacji.
- Pessimistic locking blokuje dane w bazie – używaj go, gdy konflikty są częste lub dane są krytyczne.
- W JPA dostępne są typy locków: PESSIMISTIC_READ i PESSIMISTIC_WRITE.
- Zawsze obsługuj wyjątki i testuj scenariusze współbieżne.
- Dobierz strategię do realnych potrzeb biznesowych.



MAPA 40/100

FLYWAY | LIQUIBASE

Jak bezpiecznie wersjonować schemat bazy danych i migracje

Dobra baza danych to stabilna aplikacja! Migracje = mniej problemów w przyszłości!

MODUŁ 04 • BAZY DANYCH, JPA I TRANSAKCJE

100 MAP MYŚLI ZE SPRING BOOTA

1 PO CO MIGRACJE BAZY?

KONTROLOWANY ROZWÓJ SCHEMATU

- Baza danych ewoluuje razem z aplikacją – zmiany w tabelach, kolumnach, indeksach.
- Migracje pozwalają wprowadzać zmiany w sposób uporządkowany, powtarzalny i bezpieczny.
- Ten sam schemat w każdym środowisku (dev, test, prod) – brak „działa na moim komputerze".
- Łatwiejszy rollback, audyt zmian i współpraca w zespole.
- Migracje to element dobrych praktyk DevOps i Continuous Delivery.

2 FLYWAY – PODSTAWY

PROSTY, LEKKI I SKUTECZNY

Flyway

- Oparty na prostych plikach SQL.
- Śledzi wykonane migracje w tabeli `flyway_schema_history`.
- Obsługuje wersjonowanie i migracje przy starcie aplikacji.
- Prosty w konfiguracji i użyciu.
- Świetny wybór dla większości projektów.

Tabela historii migracji:
`flyway_schema_history`
(zawiera informacje o wersjach, czasie wykonania, checksumie itp.)

3 KONWENCJA PLIKÓW FLYWAY

V<Wersja>__<Opis>.sql

Migracje SQL umieszczamy w katalogu:
`src/main/resources/db/migration`

Przykładowe nazwy plików:

```
V1__create_users_table.sql
V2__add_email_to_users.sql
V3__create_orders_table.sql
V4__add_index_to_orders.sql
V5__insert_initial_data.sql
```

Ważne zasady:

- Wersje rosnące (V1, V2, V3 ...).
- Dwa podkreślenia po numerze wersji.
- Opis w nazwie powinien być zwięzły i zrozumiały.
- Każdy plik zawiera jedną logiczną zmianę.

4 LIQUIBASE – PODSTAWY

WIĘCEJ MOŻLIWOŚCI I ELASTYCZNOŚCI

Liquibase

- Obsługuje wiele formatów changelog (XML, YAML, JSON, SQL).
- Więcej funkcji i zaawansowane opcje.
- Śledzi wykonane zmiany w tabeli databasechangelog.
- Pozwala na warunki, preconditions i złożone scenariusze.
- Dobrze sprawdza się w większych i bardziej złożonych projektach.

Tabela historii migracji:
`databasechangelog`
oraz `databasechangeloglock`
(blokada równoległego uruchomienia)

5 XML / YAML / SQL CHANGELOGS

KILKA FORMATÓW – WIĘKSZA ELASTYCZNOŚĆ

Liquibase obsługuje różne formaty definicji zmian:

- **XML** – najpopularniejszy, bardzo rozbudowany.
- **YAML** – prostszy i bardziej czytelny.
- **SQL** – podobny do podejścia Flyway.
- **JSON** – alternatywny format dla zaawansowanych.

Przykład pliku changelog (YAML):

```yaml
databaseChangelog:
  - changeSet:
      id: 1
      author: jan
      changes:
        - createTable:
            tableName: users
            columns:
              - column:
                  name: id
                  type: BIGINT
                  constraints:
                    primaryKey: true
```

6 PORÓWNANIE FLYWAY VS LIQUIBASE

KTÓRE NARZĘDZIE WYBRAĆ?

Cecha	Flyway	Liquibase
Główne podejście	Pliki SQL	Wiele formatów (XML, YAML, SQL)
Konfiguracja	Prosta	Bardziej złożona
Możliwości	Podstawowe, wystarczające dla większości	Bardzo rozbudowane (preconditions, warunki, itp.)
Czytelność	Wysoka (prosty SQL)	Wysoka, ale bardziej rozbudowana składnia
Społeczność	Duża	Duża
Zastosowanie	Małe i średnie projekty	Duże, złożone projekty, wiele baz
Krzywa nauki	Łagodna	Stromsza
Rozbudowane funkcje	Ograniczone	Bardzo szerokie
Wersjonowanie	Tak	Tak
Rollback	Ręcznie (skrypty)	Tak, wbudowane mechanizmy
Warunki wykonania	Ograniczone	Tak (preconditions)

7 INTEGRACJA ZE SPRING BOOT

MINIMALNA KONFIGURACJA

Spring Boot automatycznie uruchamia migracje przy starcie aplikacji.

Flyway:

```yaml
spring:
  flyway:
    enabled: true
    locations: classpath:db/migration
    baseline-on-migrate: true
```

Liquibase:

```yaml
spring:
  liquibase:
    enabled: true
    change-log: classpath:db/changelog/
                db.changelog-master.yaml
```

- Brak dodatkowego kodu – wystarczy konfiguracja.
- Migracje uruchamiane automatycznie przy starcie.
- Możliwość dostosowania lokalizacji plików i innych ustawień.
- Działa z dowolną bazą danych (H2, PostgreSQL, MySQL itd.).

8 TYPOWY WORKFLOW DEV → TEST → PROD

TEN SAM PROCES W KAŻDYM ŚRODOWISKU

- **DEV** (programista) – Tworzy migrację (np. V6_add_status.sql). Testuje lokalnie.
- **TEST** (CI/CD) – Automatyczne uruchomienie migracji na środowisku testowym. Weryfikacja poprawności zmian.
- **PROD** (produkcja) – Te same migracje uruchamiane automatycznie przy wdrożeniu. Identyczny schemat bazy.

Zawsze te same pliki migracji w każdym środowisku! Zero ręcznych zmian w bazie.

9 PRAKTYCZNY PRZYKŁAD KONFIGURACJI

KOMPLETNA KONFIGURACJA W application.yml

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/shop
    username: scott
    password: tiger

  flyway:
    enabled: true
    locations: classpath:db/migration
    baseline-on-migrate: true
    out-of-order: false

# Alternatywnie dla Liquibase:
  liquibase:
    enabled: false
    change-log: classpath:db/changelog/
                db.changelog-master.yaml
```

Struktura katalogów:

```
src/main/resources/
├── db/migration/      (dla Flyway)
└── db/changelog/      (dla Liquibase)
```

10 PUŁAPKI

NA CO UWAŻAĆ?

- Ręczne zmiany w bazie na produkcji (poza migracjami).
- Brak kontroli wersji plików migracji (w repozytorium Git).
- Nieodwracalne zmiany (np. usunięcie kolumny).
- Duże, monolityczne migracje – lepiej małe kroki.
- Ignorowanie błędów migracji w logach.
- Różnice w środowiskach (inna wersja bazy, inne ustawienia).
- Brak testowania migracji na środowisku zbliżonym do produkcji.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE ZASADY

- Używaj migracji bazy danych – to standard.
- Wybierz narzędzie odpowiednie do projektu.
- Trzymaj migracje w kontroli wersji (Git).
- Testuj migracje w każdym środowisku.
- Nie wprowadzaj ręcznych zmian w produkcji.
- Dbaj o małe, czytelne i odwracalne zmiany.
- Migracje = stabilność, bezpieczeństwo i spokój!

MAPA 41/100

SPRING SECURITY FILTER CHAIN

Jak działa łańcuch zabezpieczeń i co dzieje się z żądaniem zanim trafi do endpointu

Bezpieczne aplikacje to stabilne aplikacje na lata!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST FILTER CHAIN?

UPORZĄDKOWANY ŁAŃCUCH FILTRÓW

Filter Chain to uporządkowany łańcuch filtrów bezpieczeństwa, które przetwarzają żądanie HTTP przed dotarciem do kontrolera w aplikacji.

Odpowiada za uwierzytelnianie, autoryzację oraz ochronę aplikacji przed nieautoryzowanym dostępem.

Działa jako część Spring Security i jest wykonywany dla każdego przychodzącego żądania HTTP.

2 SKĄD SIĘ BIERZE?

REJESTRACJA I KONFIGURACJA

Spring Security rejestruje własny filtr (DelegatingFilterProxy), który deleguje żądania do wewnętrznego łańcucha SecurityFilterChain.

Konfiguracja odbywa się zwykle przez bean typu SecurityFilterChain i za pomocą obiektu HttpSecurity.

Spring Boot automatycznie konfiguruje podstawowy łańcuch, ale można go dostosować do potrzeb aplikacji.

Kluczowe elementy:

- DelegatingFilterProxy
- SecurityFilterChain
- HttpSecurity (DSL)

3 PRZEPŁYW ŻĄDANIA

KROK PO KROKU OD KLIENTA DO ENDPOINTU

1. **Klient**
   Wysyła żądanie HTTP (np. GET /api/produkty).
2. **Serwer**
   Żądanie trafia do aplikacji (kontener serwletów).
3. **Filtry Spring Security**
   Żądanie przechodzi przez łańcuch filtrów bezpieczeństwa.
4. **Decyzja uwierzytelniania i autoryzacji**
   Filtry sprawdzają dane logowania i reguły dostępu.

5a. Kontroler
Jeśli wszystko jest OK, żądanie trafia do endpointu (kontrolera).

5b. Odpowiedź błędu
Jeśli uwierzytelnienie lub autoryzacja się nie powiedzie – zwracana jest odpowiedź 401 lub 403.

4 WAŻNE FILTRY

PRZYKŁADOWE ELEMENTY ŁAŃCUCHA

- **SecurityContextHolderFilter**
  Inicjalizuje i zarządza SecurityContext na początku żądania.
- **CsrfFilter**
  Chroni przed atakami CSRF (domyślnie włączony dla aplikacji sesyjnych).
- **UsernamePasswordAuthenticationFilter**
  Obsługuje logowanie za pomocą formularza (login i hasło).
- **BasicAuthenticationFilter**
  Obsługuje uwierzytelnianie HTTP Basic.
- **BearerTokenAuthenticationFilter**
  Obsługuje uwierzytelnianie za pomocą tokena Bearer (np. JWT).
- **AuthorizationFilter**
  Sprawdza reguły dostępu (role, uprawnienia, request matchers).
- **ExceptionTranslationFilter**
  Obsługuje wyjątki bezpieczeństwa i zamienia je na odpowiednie odpowiedzi HTTP (401/403).

5 UWIERZYTELNIANIE

JAK FILTRY UWIERZYTELNIAJĄ UŻYTKOWNIKA

Filtry odczytują dane uwierzytelniające np. login i hasło z formularza, nagłówek Authorization (JWT, Basic) lub inny mechanizm (np. cookie, token).

Na ich podstawie tworzy obiekt Authentication (np. UsernamePasswordAuthenticationToken).

Obiekt Authentication jest przekazywany do AuthenticationManager, który deleguje do odpowiedniego AuthenticationProvider.

Jeśli uwierzytelnienie powiedzie się, użytkownik zostaje zapisany w SecurityContext.

6 AUTORYZACJA

SPRAWDZANIE DOSTĘPU DO ZASOBÓW

Po pomyślnym uwierzytelnieniu filtry sprawdzają, czy użytkownik ma dostęp do żądanego zasobu.

Weryfikowane są reguły dostępu, takie jak: role, authorities lub dopasowania żądań (request matchers).

Dostęp konfiguruje się za pomocą metod np. .requestMatchers(...).hasRole("ADMIN") lub .hasAuthority("produkt:read").

Jeśli użytkownik ma wymagane uprawnienia – żądanie trafia do kontrolera, w przeciwnym razie zwracana jest odpowiedź 403 Forbidden.

7 SECURITY CONTEXT

PRZECHOWYWANIE INFORMACJI O UŻYTKOWNIKU

SecurityContext zawiera obiekt Authentication dla bieżącego żądania.

Przechowuje informacje o uwierzytelnionym użytkowniku, jego rolach i uprawnieniach.

Dostępny jest przez SecurityContextHolder, z którego można skorzystać w kontrolerach, serwisach i innych komponentach aplikacji.

Po zakończeniu żądania kontekst jest czyszczony (dla kolejnych żądań).

Przykład użycia:

```
Authentication auth =
    SecurityContextHolder.getContext().getAuthentication();
String username = auth.getName();
```

8 PRAKTYCZNY PRZYKŁAD KONFIGURACJI

PROSTA KONFIGURACJA SECURITYFILTERCHAIN

```java
@Bean
SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {
    return http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        .httpBasic(Customizer.withDefaults())
        .build();
}
```

Co tutaj się dzieje?

- Adresy /public/** są dostępne dla wszystkich.
- Pozostałe żądania wymagają uwierzytelnienia.
- Włączamy uwierzytelnianie HTTP Basic.
- To tworzy kompletny łańcuch filtrów bezpieczeństwa.

9 CO JEŚLI COŚ PÓJDZIE ŹLE?

OBSŁUGA BŁĘDÓW I WYJĄTKÓW

401 Unauthorized
Zwracana, gdy użytkownik nie jest uwierzytelniony (brak lub nieprawidłowe dane).

403 Forbidden
Zwracana, gdy użytkownik jest uwierzytelniony, ale nie ma wymaganych uprawnień.

- **ExceptionTranslationFilter**
  Przechwytuje wyjątki bezpieczeństwa i deleguje obsługę do odpowiednich komponentów.
- **AuthenticationEntryPoint**
  Odpowiada za zwracanie odpowiedzi 401 (rozpoczęcie procesu uwierzytelniania).
- **AccessDeniedHandler**
  Odpowiada za zwracanie odpowiedzi 403 (brak uprawnień).

10 PUŁAPKI

NA CO UWAŻAĆ?

- Zła kolejność reguł dostępu – ogólne reguły mogą zablokować bardziej szczegółowe.
- Brak zrozumienia różnicy między uwierzytelnianiem a autoryzacją.
- Mieszanie konfiguracji sesyjnej i JWT bez świadomości konsekwencji (np. nieoczekiwane zachowanie).
- Przypadkowe otwarcie wrażliwych endpointów (np. /admin/**).
- Ignorowanie CSRF, gdy aplikacja używa sesji i formularzy.
- Brak testów zabezpieczeń na środowisku zbliżonym do produkcji.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Filter Chain to łańcuch filtrów przetwarzających żądanie HTTP.
- Odpowiada za uwierzytelnianie, autoryzację i ochronę aplikacji.
- Konfiguracja odbywa się przez SecurityFilterChain i HttpSecurity.
- Filtry odczytują dane logowania, tworzą Authentication i przekazują do AuthenticationManager.
- Po uwierzytelnieniu sprawdzane są reguły dostępu.
- SecurityContext przechowuje informacje o użytkowniku.
- W przypadku błędów zwracane są odpowiedzi 401 lub 403.
- Właściwa konfiguracja i testy to podstawa bezpiecznej aplikacji.



MAPA 42/100

AUTHENTICATION VS AUTHORIZATION

Czym różni się uwierzytelnianie od autoryzacji i gdzie w tym wszystkim jest SecurityContext

Ten sam użytkownik. Różne uprawnienia. Większe możliwości!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST AUTHENTICATION?

Uwierzytelnianie to proces potwierdzenia tożsamości użytkownika.

- Użytkownik podaje dane logowania (login i hasło), token (np. JWT) lub certyfikat.
- Spring Security weryfikuje dostarczone dane.
- W wyniku pomyślnej weryfikacji tworzony jest obiekt **Authentication**.

Authentication = potwierdzona tożsamość użytkownika.

2 CO TO JEST AUTHORIZATION?

Autoryzacja to proces decyzyjny, który określa, czy uwierzytelniony użytkownik ma dostęp do danego zasobu.

- Na podstawie ról, uprawnień (authorities) i reguł sprawdzane jest, czy użytkownik może wykonać określoną operację.
- Wykorzystywane są m.in. adnotacje @PreAuthorize, konfiguracja HttpSecurity i reguły dostępu.
- Wynik to decyzja: **dozwolone** lub brak dostępu (403).

Authorization = sprawdzenie, co wolno użytkownikowi.

3 NAJWAŻNIEJSZA RÓŻNICA

Authentication = kim jesteś
Authorization = co wolno

	Authentication (Uwierzytelnianie)	Authorization (Autoryzacja)
Cel	Potwierdzenie tożsamości	Decyzja o dostępie do zasobu
Odpowiedź na pytanie	Kim jesteś?	Co wolno Ci zrobić?
Dane	Login, hasło, token, certyfikat	Role, uprawnienia, reguły dostępu
Wynik	Obiekt Authentication	Dostęp do zasobu lub 403 Forbidden
Moment	Na początku żądania	Po uwierzytelnieniu przed dostępem do zasobu

4 SECURITYCONTEXT

SecurityContext przechowuje obiekt Authentication dla bieżącego żądania.

- Zawiera informacje o uwierzytelnionym użytkowniku.
- Dostęp przez SecurityContextHolder (na wątku obsługującym żądanie).
- Automatycznie wypełniany przez filtry Spring Security po pomyślnym uwierzytelnieniu.

```
SecurityContextHolder.getContext()
    .getAuthentication();
```

5 SKŁADNIKI AUTHENTICATION

Obiekt Authentication zawiera kilka ważnych elementów:

- **Principal** – reprezentuje użytkownika (np. UserDetails lub String z nazwą).
- **Credentials** – dane uwierzytelniające (np. hasło, token – zwykle po weryfikacji są usuwane).
- **Authorities** – role lub uprawnienia (np. ROLE_USER, ROLE_ADMIN).
- **Authenticated** – flaga informująca, czy uwierzytelnienie przebiegło pomyślnie.

6 TYPOWY PRZEPŁYW ŻĄDANIA

Jak wygląda proces od żądania do odpowiedzi?

1. **Klient**
   Wysyła żądanie z danymi logowania lub tokenem (np. JWT).
2. **Uwierzytelnienie**
   Spring Security weryfikuje dane (przez AuthenticationManager).
3. **Zapis w SecurityContext**
   Po pomyślnym uwierzytelnieniu obiekt Authentication trafia do SecurityContext.
4. **Autoryzacja**
   Sprawdzane są uprawnienia (role, reguły).
5. **Dostęp do endpointu**
   Żądanie jest obsługiwane.

W przypadku błędów
401 – brak lub błędne uwierzytelnienie.
403 – brak uprawnień.

7 WAŻNE KOMPONENTY

Kluczowe komponenty odpowiedzialne za uwierzytelnianie w Spring Security.

- **AuthenticationManager**
  Koordynuje proces uwierzytelniania.
- **AuthenticationProvider**
  Weryfikuje dane logowania (np. z bazy danych).
- **UserDetailsService**
  Ładuje dane użytkownika (UserDetails) na podstawie loginu.
- **PasswordEncoder**
  Szyfruje i weryfikuje hasła (np. BCryptPasswordEncoder).

8 PRAKTYCZNY PRZYKŁAD

Pobranie danych użytkownika z SecurityContext.

Java

```
Authentication authentication =
    SecurityContextHolder.getContext()
        .getAuthentication();

if (authentication != null) {
    String username = authentication.getName();
    boolean isAdmin = authentication.getAuthorities()
            .stream()
            .anyMatch(a -> a.getAuthority()
              .equals("ROLE_ADMIN"));

    System.out.println("Użytkownik: " + username);
    System.out.println("Czy admin: " + isAdmin);
}
```

9 401 VS 403

Dwa najczęstsze statusy w Spring Security.

401 Unauthorized
- Brak uwierzytelnienia lub błędne dane logowania.
- Użytkownik nie został poprawnie uwierzytelniony.
- Często zwracany przy dostępie do chronionego zasobu bez tokena.

403 Forbidden
- Użytkownik jest uwierzytelniony, ale nie ma wymaganych uprawnień.
- Brak odpowiedniej roli lub authority.
- Serwer rozumie kto to jest, ale odmawia dostępu.

10 PUŁAPKI

Czego unikać podczas pracy z uwierzytelnianiem i autoryzacją?

- **Mylenie ról z użytkownikiem** (np. sprawdzanie nazwy roli zamiast principal).
- Brak rozróżnienia między authentication a authorization.
- Ręczne odczytywanie danych użytkownika z żądania, zamiast używania SecurityContext.
- Zbyt szerokie role (np. jedna rola z pełnymi uprawnieniami).
- Przechowywanie haseł w postaci jawnej.

11 SZYBKIE PODSUMOWANIE

Najważniejsze wnioski.

- Authentication potwierdza tożsamość użytkownika.
- Autheriication decyduje o dostępie do zasobu.
- SecurityContext przechowuje Authentication dla bieżącego żądania.
- Wykorzystuj role i reguły dostępu.
- Korzystaj z gotowych komponentów Spring Security.
- Zwracaj uwagę na statusy 401 i 403.
- Dobrze zaprojektowane uprawnienia to bezpieczniejsza aplikacja.

Bezpieczeństwo to nie funkcja, to podstawa każdej aplikacji!

Bezpieczny kod to lepsze jutro!



MAPA 43/100

UŻYTKOWNIK, HASŁO I PASSWORDENCODER

Jak Spring Security przechowuje użytkowników i bezpiecznie obsługuje hasła

Bezpieczni użytkownicy to bezpieczna aplikacja!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 UŻYTKOWNIK W SYSTEMIE

CO PRZECHOWUJEMY O UŻYTKOWNIKU?

- Użytkownik loguje się za pomocą nazwy użytkownika (loginu) lub adresu e-mail.
- W systemie przechowujemy e-mail, który często jest unikalny.
- Użytkownik ma przypisane role i uprawnienia (np. USER, ADMIN).
- Konto może być aktywne lub zablokowane.

Przykładowe pola w bazie danych:

```
id, username, email, password,
role, enabled, account_non_locked
```

2 USERDETAILS

CO REPREZENTUJE?

UserDetails to interfejs z Spring Security, który reprezentuje użytkownika w systemie.

NAJWAŻNIEJSZE METODY:

- → getUsername() – nazwa użytkownika
- → getPassword() – zakodowane hasło
- → getAuthorities() – role/uprawnienia
- → isEnabled() – czy konto jest aktywne
- → isAccountNonLocked() – czy konto nie jest zablokowane

Najczęściej używamy własnej klasy (np. User), która implementuje UserDetails lub korzystamy z gotowych klas (np. `org.springframmenk.security.core.userdetails.User`).

3 USERDETAILSSERVICE

ŁADOWANIE UŻYTKOWNIKA PO LOGINIE

Interfejs UserDetailsService ma jedną główną metodę:

`loadUserByUsername(String username)`

- Na podstawie loginu (np. nazwy użytkownika lub e-maila) ładuje użytkownika z bazy danych.
- Jeśli użytkownik nie istnieje, rzucany jest wyjątek UsernameNotFoundException.
- Najczęściej implementujemy własną klasę UserDetailsService, która wyszukuje użytkownika w repozytorium.

4 DLACZEGO HASŁA SIĘ HASHUJE?

NIE PRZECHOWUJEMY HASEŁ JAWNIE!

- Hasła nie są przechowywane w postaci jawnej, tylko w postaci skrótu (hasha).
- Jeśli baza danych wycieknie, atakujący nie pozna rzeczywistych haseł.
- Hashowanie z użyciem soli (salt) sprawia, że każde hasło jest inne, nawet jeśli użytkownicy mają takie samo hasło.
- Współczesne algorytmy (np. BCrypt) są odporne na ataki słownikowe i brute force.

5 PASSWORDENCODER

INTERFEJS DO KODOWANIA I WERYFIKACJI

PasswordEncoder to interfejs, który służy do kodowania haseł i ich weryfikacji.

- encode(String rawPassword) – zwraca zakodowane hasło.
- matches(String rawPassword, String encodedPassword) – sprawdza, czy hasło pasuje do skrótu.

Weryfikacja polega na zakodowaniu podanego hasła i porównaniu go z zapisanym skrótem.

6 BCRYPT

POPULARNY I BEZPIECZNY WYBÓR

- BCrypt to domyślny i rekomendowany algorytm w Spring Security.
- Automatycznie generuje unikalną sól (salt) dla każdego hasła.
- Jest adaptacyjny – można zwiększać koszt obliczeniowy wraz z rozwojem sprzętu.
- Łatwy w użyciu – wystarczy utworzyć obiekt BCryptPasswordEncoder. To bezpieczny domyślny wybór w nowych projektach.

7 REJESTRACJA UŻYTKOWNIKA

JAK ZAPISAĆ NOWEGO UŻYTKOWNIKA?

1. Pobierz dane z formularza (np. username, email, hasło).
2. Zakoduj hasło za pomocą passwordEncoder.encode(rawPassword).
3. Utwórz encję użytkownika (z rolą np. USER).
4. Zapisz użytkownika do bazy danych.

W bazie zapisujemy tylko zakodowane hasło, nigdy w postaci jawnej!

8 LOGOWANIE UŻYTKOWNIKA

JAK SPRAWDZANE JEST HASŁO?

1. Spring Security wywołuje loadUserByUsername(login).
2. Pobiera zakodowane hasło użytkownika z bazy danych.
3. Porównuje podane hasło z zapisanym skrótem za pomocą passwordEncoder.matches().
4. Jeśli hasło jest poprawne, tworzony jest obiekt Authentication i użytkownik zostaje zalogowany.

9 PRAKTYCZNY PRZYKŁAD

KONFIGURACJA ENCODERA I ZAPIS UŻYTKOWNIKA

Java

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
// Zapis nowego użytkownika
public void registerUser(String username, String email,
                         String rawPassword) {
    String encodedPassword = passwordEncoder
        .encode(rawPassword);
    User user = new User();
    user.setUsername(username);
    user.setEmail(email);
    user.setPassword(encodedPassword);
    user.setRoles(Set.of("USER"));
    userRepository.save(user);
}
```

10 DOBRE PRAKTYKI I PUŁAPKI

NA CO UWAŻAĆ?

- ✗ Nie loguj haseł ani nie wyświetlaj ich w logach.
- ✗ Nie używaj NoOpPasswordEncoder w produkcji.
- ✗ Waliduj złożoność hasła (długość, znaki specjalne).
- ✗ Nie przechowuj haseł w plikach konfiguracyjnych.
- ✗ Ostrożnie aktualizuj role i uprawnienia użytkowników.
- ✓ Używaj BCryptPasswordEncoder i aktualizuj dependencje Spring Security.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- ✓ Użytkownik loguje się za pomocą loginu lub e-maila.
- ✓ UserDetails reprezentuje dane użytkownika.
- ✓ UserDetailsService ładuje użytkownika z bazy.
- ✓ Hasła są przechowywane w postaci hashy.
- ✓ PasswordEncoder koduje i weryfikuje hasła.
- ✓ BCrypt to bezpieczny i zalecany algorytm.
- ✓ Podczas rejestracji zawsze koduj hasło.
- ✓ Przy logowaniu używany jest matches().
- ✓ Stosuj dobre praktyki, unikaj typowych błędów.

HASŁO JAWNE vs ZAKODOWANE

Hasło jawne (raw password)	Hasło zakodowane (hash – BCrypt)
password123	$2a$10$uK9...7H3x
MojeHasło!	$2a$10$eX8...9kLm
admin2026	$2a$10$YpQ...4ZtR

PRZEPŁYW REJESTRACJI I LOGOWANIA



MAPA 44/100

HTTPSECURITY I REGUŁY DOSTĘPU

Jak budować SecurityFilterChain i kontrolować dostęp do endpointów

Bezpieczne aplikacje to lepsze aplikacje na lata!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CZYM JEST HTTPSECURITY?

DSL do konfiguracji bezpieczeństwa.

HttpSecurity to główny obiekt konfiguracyjny w Spring Security. Zapewnia płynny interfejs (DSL) do definiowania reguł dostępu, autentykacji i innych ustawień bezpieczeństwa.

Konfigurujesz HTTP security poprzez wywołania metod na obiekcie HttpSecurity, a wynik konfiguracji jest używany do utworzenia łańcucha filtrów.

Dzięki temu możesz łatwo i czytelnie określić, które zasoby są publiczne, a które wymagają autoryzacji.

2 SECURITYFILTERCHAIN

Wynik konfiguracji; bean zwracany przez metodę @Bean.

SecurityFilterChain to rezultat konfiguracji HttpSecurity. Jest to bean rejestrowany w kontekście Spring i zawiera cały łańcuch filtrów odpowiedzialnych za bezpieczeństwo aplikacji.

Tworzysz go zwykle w klasie konfiguracyjnej z adnotacją @Bean i przekazując skonfigurowany obiekt HttpSecurity.

W aplikacji może istnieć jeden lub więcej beanów SecurityFilterChain (z różnymi matcherami).

3 AUTHORIZEHTTPREQUESTS

Definiowanie reguł dostępu do endpointów.

Metoda authorizeHttpRequests()) pozwala określić, które żądania HTTP wymagają uwierzytelnienia, a które są dozwolone dla wszystkich lub tylko określonych użytkowników.

Reguły są definiowane w postaci łańcucha wywołań, gdzie każdy matcher ma przypisaną odpowiednią akcję autoryzacyjną (np. permitAll, authenticated, hasRole).

Reguły są sprawdzane w kolejności, w jakiej zostały zdefiniowane.

4 REQUESTMATCHERS

Dopasowanie ścieżek, metod HTTP i wzorców.

RequestMatchers pozwalają określić, które żądania HTTP mają być objęte daną regułą dostępu.

Możesz dopasowywać ścieżki URL, metody HTTP (GET, POST, itd.) oraz używać wzorców z symbolami wieloznacznymi (**, *).

Najczęściej używane metody:

- requestMatchers("/api/**").
- requestMatchers(HttpMethod.GET, "/users")
- requestMatchers("/admin/**").

5 PODSTAWOWE REGUŁY

Najczęściej używane reguły dostępu.

permitAll()	Dostęp dla wszystkich (użytkownik nie musi być zalogowany).
authenticated()	Wymaga uwierzytelnienia (dowolny zalogowany użytkownik).
denyAll()	Zawsze odmawia dostępu.
hasRole("ROLE")	Wymaga roli użytkownika (np. hasRole("ADMIN")).
hasAuthority("AUTH")	Wymaga określonego uprawnienia (np. hasAuthority("user:read")).

6 DODATKOWE OPCJE

Przydatne konfiguracje HttpSecurity.

httpBasic()	Włącza uwierzytelnianie HTTP Basic (dla API i prostych rozwiązań).
formLogin()	Włącza logowanie formularzowe (dla aplikacji webowych).
logout()	Konfiguracja wylogowania (np. adres, przekierowanie).
sessionManagement()	Ustawienia zarządzania sesją (np. stateless dla API).
csrf()	Konfiguracja ochrony CSRF (można wyłączyć, np. dla API).
cors()	Konfiguracja CORS (dla aplikacji frontendowych).

7 PUBLICZNE VS CHRONIONE ENDPOINTY

Prosty schemat podziału zasobów.

/public/**	Dostęp publiczny (bez uwierzytelnienia) np. strona główna, zasoby statyczne.
/api/**	Wymaga uwierzytelnienia (dowolny zalogowany użytkownik) np. endpoints dla zalogowanych.
/admin/**	Wymaga określonej roli (np. ADMIN) np. panel administracyjny.

8 PRAKTYCZNY PRZYKŁAD KONFIGURACJI

Kompletna konfiguracja SecurityFilterChain.

```java
@Bean
public SecurityFilterChain securityFilterChain
        (HttpSecurity http) throws Exception {
    return http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/api/**").authenticated()
            .anyRequest().denyAll())
        .httpBasic()
        .csrf(csrf -> csrf.disable())
        .sessionManagement(session ->
            session.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS))
        .build();
}
```

9 JAK CZYTAĆ REGUŁY?

Kolejność ma znaczenie!

1. Reguły są sprawdzane od góry do dołu.
2. Bardziej szczegółowe reguły powinny być zdefiniowane przed ogólnymi.
3. Pierwszy pasujący matcher określa, jaka akcja zostanie wykonana.
4. Jeśli żadne reguły nie pasują, zostanie użyta reguła domyślna (np. denyAll()).

Wskazówka: Najpierw definiuj reguły dla konkretnych ścieżek, a na końcu regułę ogólną (np. anyRequest()).

10 PUŁAPKI

Najczęstsze błędy przy konfiguracji.

Zła kolejność reguł	Ogólne reguły przed szczegółowymi mogą zablokować dostęp.
Pomieszanie hasRole i hasAuthority	hasRole("ADMIN") oczekuje roli z przedrostkiem ROLE_. hasAuthority("ADMIN") sprawdza konkretne uprawnienie.
Wyłączenie CSRF bez zrozumienia	Może narazić aplikację na ataki. Wyłączaj tylko jeśli wiesz dlaczego (np. API stateless).
Brak testów dostępu	Nieprzetestowane reguły mogą powodować nieoczekiwane błędy 401/403.

11 SZYBKIE PODSUMOWANIE

Najważniejsze informacje do zapamiętania.

- HttpSecurity to DSL do konfiguracji bezpieczeństwa.
- SecurityFilterChain to wynik konfiguracji (bean).
- authorizeHttpRequests() pozwala definiować reguły dostępu.
- RequestMatchers dopasowują ścieżki, metody i wzorce.
- Dostęp kontrolujesz za pomocą permitAll, authenticated, denyAll, hasRole, hasAuthority.
- Możesz włączyć httpBasic, formLogin, logout, csrf, cors.
- Kolejność reguł ma znaczenie – bardziej szczegółowe przed ogólnymi.
- Zawsze testuj konfigurację i sprawdzaj dostęp do endpointów.

HASROLE vs HASAUTHORITY

hasRole()	hasAuthority()
Oczekuje roli	Oczekuje konkretnego uprawnienia
Automatycznie dodaje przedrostek ROLE_	Nie dodaje przedrostka
Przykład: hasRole("ADMIN") (sprawdza ROLE_ADMIN)	Przykład: hasAuthority("user:read") (sprawdza dokładnie to uprawnienie)

PRZEBIEG ŻĄDANIA (REQUEST FLOW)

Żądanie HTTP → Dopasowanie RequestMatcher → Sprawdzenie reguł dostępu → Wykonanie odpowiedniej akcji (permit / auth / deny)

MAPA 45/100

METHOD SECURITY

Jak chronić metody serwisów za pomocą adnotacji i wyrażeń SpEL

Bezpieczne metody, bezpieczniejsza aplikacja

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST METHOD SECURITY?

OCHRONA NA POZIOMIE METOD

Method Security pozwala chronić pojedyncze metody w serwisach, repozytoriach i innych komponentach za pomocą adnotacji.

Działa niezależnie od zabezpieczeń na poziomie URL – nawet jeśli metoda jest wywoływana wewnętrznie w aplikacji.

Umożliwia precyzyjną kontrolę dostępu do logiki biznesowej.

Sprawdza, czy aktualny użytkownik ma odpowiednie uprawnienia do wykonania metody.

Method Security chroni to, **CO robisz**, a nie tylko **DO CZEGO** się logujesz.

2 @ENABLEMETHODSECURITY

AKTYWACJA OBSŁUGI ADNOTACJI

Aby korzystać z adnotacji zabezpieczających metody, musisz włączyć obsługę Method Security w swojej aplikacji.

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
    // inne konfiguracje
}
```

- Włącza obsługę adnotacji takich jak @PreAuthorize, @PostAuthorize, @Secured, @RolesAllowed.
- Dostępne od Spring Security 6 (zastępuje @EnableGlobalMethodSecurity).
- Można dodatkowo skonfigurować własny MethodSecurityExpressionHandler.

3 @PREAUTHORIZE

SPRAWDZANIE DOSTĘPU PRZED WYKONANIEM

@PreAuthorize sprawdza, czy użytkownik ma odpowiednie uprawnienia zanim metoda zostanie wykonana.

```
@PreAuthorize("hasRole('ADMIN')")
public void usunUzytkownika(Long id) {
    // logika usuwania
}
```

Jeśli wyrażenie zwróci **false**, metoda nie zostanie wykonana, a zostanie rzucony wyjątek AccessDeniedException.

4 @POSTAUTHORIZE

SPRAWDZANIE DOSTĘPU PO WYKONANIU

@PostAuthorize pozwala sprawdzić wynik metody po jej wykonaniu oraz zdecydować, czy użytkownik może go otrzymać.

```
@PostAuthorize("returnObject.owner ==
authentication.name || hasRole('ADMIN')")
public Dokument pobierzDokument(Long id) {
    return dokumentService.findById(id);
}
```

Metoda zostanie wykonana, ale wynik zostanie zwrócony tylko wtedy, gdy wyrażenie jest spełnione.

5 @SECURED I @ROLESALLOWED

PROSTSZE ALTERNATYWY

@Secured i **@RolesAllowed** to prostsze adnotacje, które sprawdzają role użytkownika.

```
@Secured("ROLE_ADMIN")
public void operacjaAdmina() { ... }

@RolesAllowed("USER")
public void operacjaUzytkownika() { ... }
```

- **@Secured** – adnotacja z Spring Security, wymaga pełnej nazwy roli (np. "ROLE_ADMIN").
- **@RolesAllowed** – standard Java (JSR-250), również wymaga pełnej nazwy roli.
- Nie obsługują wyrażeń SpEL – są prostsze, ale mniej elastyczne niż @PreAuthorize.

6 WYRAŻENIA SPEL

ELASTYCZNE WARUNKI DOSTĘPU

W adnotacjach można używać wyrażeń SpEL (Spring Expression Language). Najczęściej używane wyrażenia:

```
@PreAuthorize("hasRole('ADMIN')")
@PreAuthorize("hasAuthority('app:read')")
@PreAuthorize("authentication.name == #id")
@PreAuthorize("#userId == authentication.name
              || hasRole('ADMIN')")
```

Popularne obiekty i funkcje SpEL:

- hasRole("ROLE_NAME") – sprawdza rolę użytkownika
- hasAuthority("permission") – sprawdza uprawnienie
- authentication.name – nazwa aktualnego użytkownika
- #id, #userId – parametry metody

7 KIEDY UŻYWAĆ METHOD SECURITY?

TYPOWE SCENARIUSZE

Gdy logika biznesowa wymaga różnych uprawnień do różnych operacji.

Gdy metoda może być wywoływana z wielu miejsc (np. z REST API, zadań planowanych, komunikatów).

Gdy chcesz zabezpieczyć operacje wewnątrz aplikacji, nie tylko przez URL.

Gdy potrzebujesz dostępu opartego o dane domenowe (np. właściciel zasobu).

Method Security to najlepszy wybór dla ochrony logiki biznesowej.

8 METHOD SECURITY VS URL SECURITY

PORÓWNANIE PODEJŚĆ

Cecha	Method Security	URL Security
Co chroni?	Metody w aplikacji	Endpointy HTTP (URL)
Gdzie działa?	Serwisy, repozytoria, komponenty	Warstwa web (kontrolery)
Granularność	Bardzo wysoka	Średnia
Obsługa SpEL	Tak	Ograniczona
Działa przy wywołaniach wewnętrznych?	Tak	Nie
Najlepsze zastosowanie	Logika biznesowa	Kontrola dostępu do zasobów HTTP

Najlepsze efekty daje połączenie obu podejść – **URL Security + Method Security**.

9 PRAKTYCZNY PRZYKŁAD

KOMPLETNA KONFIGURACJA I SERWIS

```
// Konfiguracja
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
    // inne konfiguracje bezpieczeństwa
}

// Serwis z zabezpieczoną metodą
@Service
public class UserService {

    @PreAuthorize("#id == authentication.name
                  || hasRole('ADMIN')")
    public User getUserById(String id) {
        return userRepository.findById(id);
    }
}
```

10 PUŁAPKI

CZEGO UNIKAĆ?

- Zbyt skomplikowane wyrażenia SpEL – trudne w utrzymaniu.
- Poleganie tylko na URL Security – brak ochrony przy wywołaniach wewnętrznych.
- Brak testów zabezpieczeń metod.
- Używanie ról bez konwencji (np. "ADMIN" zamiast "ROLE_ADMIN").

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Method Security chroni metody w aplikacji za pomocą adnotacji.
- Włącz obsługę adnotacji przez @EnableMethodSecurity.
- Używaj @PreAuthorize, @PostAuthorize, @Secured lub @RolesAllowed.
- Wyrażenia SpEL dają dużą elastyczność.
- Stosuj Method Security do ochrony logiki biznesowej.
- Najlepsze rezultaty daje połączenie z URL Security.
- Pamiętaj o testach i czytelnych wyrażeniach.



MAPA 46/100

SESSION VS STATELESS AUTHENTICATION

Kiedy używać sesji, a kiedy tokenów w nowoczesnych aplikacjach

Różne podejścia. Ten sam cel. Bezpieczne aplikacje!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 MODEL SESYJNY

Serwer przechowuje stan sesji użytkownika po stronie serwera.

- Po zalogowaniu serwer tworzy sesję i przechowuje jej stan (np. w pamięci, bazie danych lub klastrze).
- Identyfikator sesji (np. **JSESSIONID**) wysyłany jest do przeglądarki w formie ciasteczka (cookie).
- Przy kolejnych żądaniach przeglądarka automatycznie wysyła ciasteczko z identyfikatorem sesji.
- Serwer na podstawie identyfikatora odnajduje sesję i wie, kim jest użytkownik.

2 MODEL STATELESS

Brak stanu po stronie serwera. Używane są zwykle tokeny (JWT, Bearer).

- Po zalogowaniu serwer generuje token (najczęściej JWT) zawierający informacje o użytkowniku.
- Token jest zwracany do klienta, który przechowuje go i wysyła przy każdym żądaniu.
- Serwer nie przechowuje stanu sesji – weryfikuje token przy każdym żądaniu (np. sprawdza podpis).
- Token zwykle przekazywany jest w nagłówku HTTP: Authorization: Bearer <token>.

3 JAK DZIAŁA SESJA HTTP

Proces krok po kroku:

1. **Logowanie**
   Użytkownik podaje dane logowania (np. formularz w aplikacji web).
2. **Tworzenie sesji**
   Serwer weryfikuje dane i tworzy sesję (np. w pamięci lub bazie danych).
3. **Wysłanie ciasteczka**
   Serwer odsyła ciasteczko z identyfikatorem sesji (np. **JSESSIONID**).
4. **Kolejne żądania**
   Przeglądarka automatycznie wysyła ciasteczko przy każdym żądaniu.
5. **Uwierzytelniony użytkownik**
   Serwer odnajduje sesję i umożliwia dostęp do zasobów.

4 JAK DZIAŁAJĄ TOKENY

Proces krok po kroku:

1. **Logowanie**
   Użytkownik podaje dane (np. w aplikacji SPA, mobile lub przez API).
2. **Generowanie tokena**
   Serwer weryfikuje dane i zwraca token (np. JWT) zawierający informacje o użytkowniku.
3. **Wysyłanie tokena**
   Klient przechowuje token i dołącza go do każdego żądania w nagłówku: Authorization: Bearer <token>
4. **Weryfikacja tokena**
   Serwer sprawdza poprawność tokena (np. podpis i datę ważności).
5. **Dostęp do zasobów**
   Jeśli token jest poprawny, użytkownik otrzymuje dostęp do chronionych zasobów.

5 ZALETY I WADY SESJI

ZALETY	WADY
Prosta implementacja i użycie w aplikacjach web.	Stan przechowywany na serwerze (większe zużycie pamięci).
Łatwe unieważnianie sesji (np. przy wylogowaniu).	Trudniejsza skalowalność (wymaga współdzielenia sesji w klastrze).
Dobrze sprawdza się w aplikacjach monolitycznych (MVC).	Mniej wygodne w API i aplikacjach mobilnych.
Wbudowane mechanizmy w przeglądarkach (cookie).	Wrażliwość na ataki CSRF (wymaga dodatkowej ochrony).

6 ZALETY I WADY JWT / STATELESS

ZALETY	WADY
Brak stanu po stronie serwera (łatwa skalowalność).	Trudniejsze unieważnianie tokena przed wygaśnięciem (problem revocation).
Dobre dopasowanie do architektury REST i mikroserwisów.	Zwykle wymaga mechanizmu odświeżania tokenów (refresh token).
Może być używany w SPA, aplikacjach mobilnych i zewnętrznych klientach.	Większy rozmiar żądania (wysyłany w każdym nagłówku).
Zawiera informacje o użytkowniku (np. role, uprawnienia).	Wrażliwy na nieprawidłowe przechowywanie po stronie klienta (np. w localStorage).

7 KIEDY CO WYBRAĆ?

Najczęstsze scenariusze:

- **Aplikacja web (MVC)**
  Najczęściej **sesje** – proste i sprawdzone rozwiązanie.
- **Aplikacja SPA (np. Angular, React)**
  Zwykle **tokeny JWT** – brak stanu, łatwa integracja z API.
- **Aplikacja mobilna**
  Tokeny JWT – wygodna obsługa w aplikacjach natywnych.
- **Mikroserwisy / REST API**
  Tokeny JWT – lepsza skalowalność i brak współdzielenia sesji.
- **Systemy wewnętrzne (intranet)**
  Sesje – prostsza konfiguracja, wystarczające bezpieczeństwo.

8 PRAKTYCZNY PRZYKŁAD ARCHITEKTURY

Aplikacja web (sesje)



REST API (JWT)



9 PUŁAPKI

Częste błędy, których warto unikać:

- Przechowywanie tokena w **localStorage** (narażenie na ataki **XSS**).
- Brak **refresh tokenów** (prowadzi do częstego ponownego logowania).
- Brak ochrony **CSRF** w aplikacjach sesyjnych.
- **Zbyt długi czas życia tokena (TTL)** (większe ryzyko przejęcia).
- **Mieszanie sesji i tokenów bez potrzeby** (prowadzi do niepotrzebnej złożoności).

10 SZYBKIE PODSUMOWANIE

- Sesje przechowują stan na serwerze, tokeny są bezstanowe.
- Sesje są prostsze, tokeny lepiej skalują się w architekturze REST.
- Wybór zależy od typu aplikacji i wymagań.
- Zadbaj o bezpieczeństwo (CSRF, XSS, refresh tokeny, TTL).
- Nie ma jednego idealnego rozwiązania – wybierz świadomie!

> *Wybierz narzędzie dopasowane do swoich potrzeb, nie dlatego, że jest modne, ale dlatego, że rozwiązuje Twój problem.*



Lepsze aplikacje tworzysz dziś!

MAPA 47/100

CSRF

Czym jest Cross-Site Request Forgery i kiedy trzeba się nim realnie przejmować

Bezpieczne aplikacje to lepsze aplikacje na lata! ☺

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST CSRF?

DEFINICJA I NA CZYM POLEGA

CSRF (Cross-Site Request Forgery) to atak, w którym napastnik nakłania ofiarę do wykonania niezamierzonego żądania w zaufanej aplikacji, w której ofiara jest już uwierzytelniona (ma aktywną sesję).

Atak polega na tym, że przeglądarka ofiary automatycznie wysyła ciasteczka sesyjne do naszej aplikacji, a my nie jesteśmy w stanie łatwo odróżnić złośliwego żądania od prawdziwego.

Ofiara nie musi klikać niczego specjalnego – wystarczy, że odwiedzi złośliwą stronę.

2 JAK WYGLĄDA ATAK?

PROSTY SCHEMAT PRZEPŁYWU



Zawiera kod np. obrazek lub formularz, który wywołuje żądanie do naszej aplikacji.

3. Aplikacja wykonuje żądanie, bo widzi prawidłową sesję ofiary.

Ofiara myśli, że nadal przegląda zaufaną stronę, a w tle wykonywane są działania w naszej aplikacji.

3 KIEDY RYZYKO JEST WYSOKIE?

SYTUACJE, W KTÓRYCH TRZEBA UWAŻAĆ

- Operacje modyfikujące dane (np. zmiana hasła, e-maila, ustawień).
- Aplikacje korzystające z sesji i ciasteczek (JSESSIONID).
- Systemy bankowości elektronicznej i płatności online.
- Panele administracyjne i funkcje dla administratorów.
- Wszędzie tam, gdzie użytkownik jest już zalogowany, a żądanie może zmienić stan aplikacji.

Ryzyko jest niskie dla operacji tylko do odczytu (GET), ale wysokie dla operacji modyfikujących (POST, PUT, DELETE).

4 DLACZEGO DOTYCZY APLIKACJI SESYJNYCH?

PRZEGLĄDDRKA WYSYŁA CIASTECZKA AUTOMATYCZNIE

Przeglądarka automatycznie dołącza ciasteczka (np. JSESSIONID) do każdego żądania kierowanego do tej samej domeny, niezależnie od tego, skąd pochodzi żądanie (nasza strona czy złośliwa).

Dzięki temu aplikacja widzi, że żądanie pochodzi od zalogowanego użytkownika, nawet jeśli zostało wywołane przez zewnętrzną stronę.

To wygodne dla użytkownika, ale niebezpieczne, jeśli nie mamy dodatkowej ochrony (np. tokenów CSRF).

5 CSRF W REST API

KIEDY RYZYKO JEST MNIEJSZE?

JWT W nagłówku Authorization

- Jeśli używamy JWT przekazywanego w nagłówku Authorization: Bearer …
- przeglądarka nie wysyła go automatycznie na inne domeny.
- Ryzyko CSRF jest wtedy znacznie mniejsze.

Ciasteczka (cookie)

- Jeśli w REST API nadal używamy ciasteczek (np. z sesją lub tokenem JWT w cookie), ryzyko CSRF nadal istnieje.
- Trzeba stosować ochronę (tokeny CSRF) lub odpowiednie ustawienia ciasteczek (SameSite, HttpOnly, Secure).

Samo „to jest REST API" nie oznacza automatycznie, że CSRF nas nie dotyczy – liczy się sposób autoryzacji!

6 JAK SPRING SECURITY CHRONI PRZED CSRF?

DOMYŚLNA OCHRONA I TOKENY CSRF

- Spring Security domyślnie włącza ochronę przed CSRF w aplikacjach web (z sesją).
- Do każdego formularza HTML dodawany jest token CSRF (ukryte pole), który musi zostać przesłany razem z żądaniem.
- Spring Security weryfikuje token po stronie serwera – jeśli brakuje go lub jest nieprawidłowy, żądanie jest odrzucane (403 Forbidden).
- Ochrona działa automatycznie dla widoków Thymeleaf i formularzy HTML.

Dzięki tokenom CSRF złośliwa strona nie jest w stanie prawidłowo odtworzyć żądania, nawet jeśli ofiara ma aktywną sesję.

7 WYŁĄCZENIE VS POZOSTAWIENIE OCHRONY

KIEDY ZOSTAWIĆ, A KIEDY MOŻNA WYŁĄCZYĆ?

Zwykle zostaw ochronę (domyślnie)

- Aplikacje web z formularzami HTML.
- Aplikacje z sesją i widokami (np. Thymeleaf).
- Każda aplikacja, w której użytkownik wykonuje operacje modyfikujące dane.

Można wyłączyć (z rozwagą)

- Czyste REST API używające JWT w nagłówku Authorization.
- Brak sesji i brak ciasteczek uwierzytelniających.
- Klienci to aplikacje mobilne, SPA lub zewnętrzne serwisy, które same kontrolują nagłówki.

Wyłączaj tylko wtedy, gdy naprawdę rozumiesz konsekwencje i masz inny model bezpieczeństwa.

8 PRAKTYCZNY PRZYKŁAD KONFIGURACJI

DOMYŚLNA OCHRONA LUB WYŁĄCZENIE DLA REST API

```java
// Domyślna ochrona (zalecane dla aplikacji web)
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
    return http
        .csrf(withDefaults())
        .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated())
        .build();
}
```

```java
// Wyłączenie CSRF dla czystego REST API
@Bean
SecurityFilterChain apiSecurityFilterChain(HttpSecurity http)
        throws Exception {
    return http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated())
        .build();
}
```

W prawdziwym projekcie konfiguracja może być bardziej rozbudowana (np. wiele filtrów, różne ścieżki).

9 PUŁAPKI

NA CO UWAŻAĆ?

- Wyłączanie CSRF bez zrozumienia konsekwencji – może otworzyć aplikację na realne ataki.
- Mylenie CORS z CSRF – to różne mechanizmy. CORS kontroluje dostęp z innych domen, nie chroni przed CSRF.
- Brak odpowiednich ustawień ciasteczek (SameSite, HttpOnly, Secure).
- Przekonanie, że „to tylko wewnętrzna aplikacja" – atak może pochodzić z zewnętrznej strony.
- Używanie GET do operacji modyfikujących dane – zwiększa ryzyko i ułatwia ataki.

Bezpieczeństwo to nie tylko kod, ale także konfiguracja i świadome decyzje.

10 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- CSRF to atak wykorzystujący zaufanie przeglądarki.
- Dotyczy głównie aplikacji sesyjnych (ciasteczka).
- Ryzyko jest wysokie przy operacjach modyfikujących dane.
- Spring Security domyślnie chroni przed CSRF (tokeny).
- W REST API z JWT w nagłówku ryzyko jest mniejsze.
- Nie myl CSRF z CORS.
- Wyłączaj ochronę tylko świadomie i z uzasadnieniem.

MAPA 48/100

CORS

Dlaczego przeglądarka blokuje żądania między domenami i jak poprawnie to skonfigurować

Bezpieczne aplikacje działają lepiej w prawdziwym świecie!

MODUŁ 05 • SPRING SECURITY

100 MAP MYŚLI ZE SPRING BOOTA

1 CO TO JEST CORS?

Cross-Origin Resource Sharing

CORS to mechanizm, który pozwala (przy odpowiedniej konfiguracji) na wykonywanie żądań HTTP z jednej domeny do zasobów na innej domenie.

- To standard przeglądarki (nie Springa)
- Działa na podstawie specjalnych nagłówków HTTP
- Umożliwia kontrolowany dostęp do zasobów między domenami

2 SAME-ORIGIN POLICY

Przeglądarka domyślnie pozwala na żądania tylko do tej samej domeny (origin).

Origin = protokół + host + port

`https://example.com:8080`

- **Protokół** np. http, https
- **Host** np. example.com
- **Port** np. 80, 443, 8080

Dlaczego istnieje?

Chroni użytkownika przed złośliwymi stronami, które mogłyby wykonywać żądania bez jego wiedzy, np. odczytać dane z innych serwisów.

3 SIMPLE REQUEST VS PREFLIGHT

Simple Request (zwykłe żądanie)

Używane dla prostych żądań:

- metody: GET, POST, HEAD
- standardowe nagłówki (np. Content-Type: application/x-www-form-urlencoded)
- brak niestandardowych nagłówków

Preflight Request (żądanie OPTIONS)

Wysyłane automatycznie przez przeglądarkę dla „bardziej złożonych" żądań, np.

- metody: PUT, DELETE, PATCH
- niestandardowe nagłówki (np. Authorization)
- Content-Type: application/json

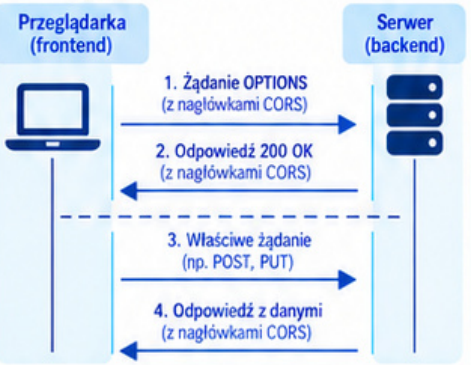
4 NAJWAŻNIEJSZE NAGŁÓWKI

Nagłówki HTTP używane w CORS:

Nagłówek	Opis
Access-Control-Allow-Origin	Określa, które domeny mogą wykonywać żądania. Np. https://example.com lub * (wszystkie).
Access-Control-Allow-Methods	Jakie metody HTTP są dozwolone. Np. GET, POST, PUT, DELETE.
Access-Control-Allow-Headers	Jakie nagłówki mogą być używane. Np. Content-Type, Authorization.
Access-Control-Allow-Credentials	Czy można wysyłać dane uwierzytelniające (cookies, tokeny). Wartość: true lub false.

5 TYPOWY PRZEPŁYW PREFLIGHT

Dla złożonych żądań przeglądarka najpierw wysyła żądanie OPTIONS.



Jeśli odpowiedź na OPTIONS zawiera odpowiednie nagłówki, przeglądarka wysyła właściwe żądanie.

6 SKĄD BIORĄ SIĘ BŁĘDY CORS?

Błędy CORS to ograniczenie przeglądarki, a nie samego backendu.

Jeśli brakuje odpowiednich nagłówków w odpowiedzi, przeglądarka blokuje dostęp do danych, nawet jeśli backend zwraca poprawną odpowiedź (np. 200 OK).

Najczęstsze przyczyny:

- Brak nagłówków CORS w odpowiedzi
- Nieprawidłowa konfiguracja originów
- Użycie niedozwolonych metod lub nagłówków
- Brak obsługi żądania OPTIONS
- Próba wysłania danych uwierzytelniających do nieautoryzowanej domeny

7 KONFIGURACJA W SPRING MVC

W aplikacji Spring MVC można skonfigurować CORS używając WebMvcConfigurer.

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
      .allowedOrigins("https://example.com")
      .allowedMethods("GET", "POST", "PUT", "DELETE")
      .allowedHeaders("*")
      .allowCredentials(true);
  }
}
```

8 KONFIGURACJA W SPRING SECURITY

W Spring Security wystarczy włączyć obsługę CORS i (opcjonalnie) dostarczyć CorsConfigurationSource.

```java
@Bean
public SecurityFilterChain securityFilterChain
      (HttpSecurity http) throws Exception {
  http
      .cors(withDefaults())
      .authorizeHttpRequests(auth -> auth
        .anyRequest().authenticated());
  return http.build();
}
```

Spring Security automatycznie użyje beana CorsConfigurationSource, jeśli jest zdefiniowany w kontekście aplikacji.

9 PRAKTYCZNY PRZYKŁAD

Konfiguracja CorsConfigurationSource w Spring Boot.

```java
@Bean
public CorsConfigurationSource corsConfigurationSource() {
  CorsConfiguration config = new CorsConfiguration();
  config.setAllowedOrigins(List.of("https://example.com"));
  config.setAllowedMethods(List.of("GET", "POST", "PUT",
                                   "DELETE"));
  config.setAllowedHeaders(List.of("Content-Type",
                                   "Authorization"));
  config.setAllowCredentials(true);

  UrlBasedCorsConfigurationSource source =
      new UrlBasedCorsConfigurationSource();
  source.registerCorsConfiguration("/api/**", config);
  return source;
}
```

10 PUŁAPKI

Najczęstsze błędy przy konfiguracji CORS:

- Użycie allowCredentials(true) z originem "*" (niedozwolone).
- Otwarcie wszystkich originów na produkcji.
- Mylenie CORS z uwierzytelnianiem i autoryzacją (CORS nie zastępuje bezpieczeństwa).
- Brak obsługi żądania OPTIONS (preflight).
- Zbyt szerokie reguły (np. wszystkie metody i nagłówki).
- Brak konfiguracji CORS w Spring Security (gdy używamy security filter chain).

11 SZYBKIE PODSUMOWANIE

- CORS pozwala na kontrolowany dostęp między domenami.
- Same-Origin Policy chroni użytkownika.
- Proste żądania nie wymagają preflight, złożone tak.
- Kluczowe nagłówki: Allow-Origin, Methods, Headers, Credentials.
- Błędy CORS to ochrona przeglądarki, nie błąd backendu.
- W Spring MVC użyj WebMvcConfigurer.
- W Spring Security włącz cors(withDefaults()).
- Możesz zdefiniować CorsConfigurationSource.
- Uważaj na pułapki, szczególnie z allowCredentials.
- Testuj konfigurację w różnych przeglądarkach.



MAPA 49/100

OAUTH2 RESOURCE SERVER

JWT, opaque token i jak Spring Security chroni endpointy z tokenem Bearer

Bezpieczne API to lepsze aplikacje i szczęśliwsi użytkownicy!

MODUŁ 05 • SPRING SECURITY | OCHRONA ZASOBÓW | TOKENY BEARER | PRAKTYCZNE PRZYKŁADY

1 CZYM JEST RESOURCE SERVER?

Resource Server to serwer API, który chroni swoje zasoby (endpointy) i akceptuje tokeny Bearer wystawione przez Authorization Server.

- Weryfikuje tokeny (JWT lub opaque)
- Kontroluje dostęp do endpointów
- Nie wystawia tokenów (to rola Authorization Server)
- Zwraca dane tylko dla uwierzytelnionych i uprawnionych użytkowników

2 BEARER TOKEN

Bearer token to token dostępu, który klient przesyła w nagłówku Authorization w każdym żądaniu do API.

```
GET /api/products HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

- Przesyłany w nagłówku Authorization
- Daje dostęp do chronionych zasobów
- Może być JWT lub opaque token
- Kto posiada token, ten może z niego skorzystać („bearer" = okaziciel)

3 JWT

JWT (JSON Web Token) to samodzielnie niosący informacje token, który zawiera tzw. claims, jest podpisany i ma określony czas ważności (exp).

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjMiLCJyb2xlcyI6Wy7JU0VSIl0s
ImVzcCi6MTcwMDAoMDAwfQ.Signature
```

- Zawiera informacje o użytkowniku (claims)
- Jest podpisany (np. RS256)
- Posiada czas ważności (exp)
- Może zawierać issuer (iss), audience (aud), role itp.

4 OPAQUE TOKEN

Opaque token to nieczytelny identyfikator, który sam w sobie nie zawiera informacji o użytkowniku. Resource Server musi skontaktować się z Authorization Server w celu jego introspekcji.

- Zwykle losowy, nieczytelny ciąg znaków
- Wymaga zapytania do endpointu introspekcji
- Pozwala na natychmiastowe unieważnienie
- Często używany w środowiskach korporacyjnych

5 WERYFIKACJA JWT

Resource Server weryfikuje JWT lokalnie, sprawdzając podpis i najważniejsze parametry tokenu.

- Sprawdza poprawność podpisu (klucz publiczny)
- Weryfikuje issuer (iss)
- Weryfikuje audience (aud)
- Sprawdza czas ważności (exp, nbf)
- Odczytuje claims (np. role, scope)

6 INTROSPEKCJA OPAQUE TOKEN

Resource Server wysyła zapytanie do Authorization Server na endpoint introspekcji, aby sprawdzić, czy token jest aktywny i jakie ma uprawnienia.

```
POST /oauth2/introspect HTTP/1.1
Authorization: Basic base64(clientId:secret)
Content-Type: application/x-www-form-urlencoded
token=opaque-token-value
```

- Zwraca informację, czy token jest aktywny
- Może zwracać dodatkowe informacje (scope, sub)
- Umożliwia natychmiastowe unieważnienie tokenu
- Wymaga clientId i clientSecret

7 KONFIGURACJA JWT W SPRING SECURITY

Minimalna konfiguracja Resource Servera dla JWT w Spring Security.

```
@Bean
public SecurityFilterChain securityFilterChain(
  HttpSecurity http) throws Exception {
 http
   .authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/public/**").permitAll()
    .anyRequest().authenticated()
  )
  .oauth2ResourceServer(oauth2 -> oauth2
   .jwt());

 return http.build();
}
```

8 KONFIGURACJA OPAQUE TOKEN

Konfiguracja Resource Servera dla tokenów opaque z introspekcją.

```
@Bean
public SecurityFilterChain securityFilterChain(
  HttpSecurity http) throws Exception {
 http
   .authorizeHttpRequests(auth -> auth
      .anyRequest().authenticated()
  )
  .oauth2ResourceServer(oauth2 -> oauth2
    .opaqueToken(opaque -> opaque
      .introspectionUri("https://auth-server/oauth2/introspect")
      .introspectionClientCredential(
        "client-id", "client-secret" ));
 return http.build();
}
```

9 JWT VS OPAQUE

Cecha	JWT	Opaque Token
Wydajność	Wysoka (weryfikacja lokalnie)	Niższa (wymaga zapytania)
Zawartość	Zawiera claims	Nie zawiera informacji (tylko identyfikator)
Prostota weryfikacji	Lokalna weryfikacja podpisu	Wymaga introspekcji
Unieważnienie	Trudniejsze (musi wygasnąć)	Możliwe natychmiast (po stronie AS)
Rozmiar	Większy	Mniejszy
Zastosowanie	Mikroserwisy, API publiczne	Środowiska korporacyjne, większa kontrola

10 PUŁAPKI

- **Brak walidacji issuer i audience** – akceptowanie tokenów z nieznanych źródeł.
- **Zbyt dużo danych w JWT** – nie umieszczaj wrażliwych informacji.
- **Brak obsługi odpowiedzi 401/403** – nieprawidłowa obsługa błędów uwierzytelniania i autoryzacji.
- **Mylenie Resource Server z Authorization Server** – Resource Server nie wystawia tokenów!

11 SZYBKIE PODSUMOWANIE

- Resource Server chroni zasoby i akceptuje tokeny.
- Tokeny Bearer są przesyłane w nagłówku Authorization.
- JWT zawiera claims i jest weryfikowany lokalnie.
- Opaque token wymaga introspekcji.
- Weryfikuj podpis, issuer, audience i ważność tokenów.
- W Spring Security używaj oauth2ResourceServer().
- Wybierz JWT lub opaque token zgodnie z wymaganiami wydajności i bezpieczeństwa.
- Unikaj typowych pułapek i poprawnie obsługuj błędy.

PRZYKŁADOWY PRZEPŁYW ŻĄDANIA



Secure Code Brighter Future!

MAPA 50/100

OAUTH2 CLIENT, LOGIN I AUTHORIZATION SERVER

Jak rozumieć role, przepływy i współpracę aplikacji podczas logowania i autoryzacji

Bezpieczne logowanie. Większe możliwości. Lepsze aplikacje!

MODUŁ 05 • SPRING SECURITY | AUTENTYKACJA • AUTORYZACJA • OAUTH2 | OPENID CONNECT

1 TRZY WAŻNE ROLE

GŁÓWNI GRACZE W OAUTH2 / OIDC

Użytkownik
Osoba, która chce skorzystać z aplikacji.

Authorization Server
Uwierzytelnia użytkownika i wydaje tokeny.

Client
Aplikacja, która loguje użytkownika i używa tokenów do dostępu do zasobów.

Resource Server
Chronione API z zasobami, które sprawdza tokeny.

2 AUTHORIZATION SERVER

CENTRUM UWIERZYTELNIANIA I TOKENÓW

Uwierzytelnia użytkownika
(np. formularz, social login, SSO).

Wydaje tokeny
(access token, id token, refresh token).

Udostępnia endpointy
/authorize – do logowania i zgody,
/token – do wymiany kodu na token.

Może przechowywać informacje o użytkownikach lub integrować się z innymi źródłami (LDAP, social, itp.).

Przykłady rozwiązań:
- Spring Authorization Server
- Keycloak
- Okta / Auth0
- Google / GitHub / Microsoft

3 OAUTH2 CLIENT

APLIKACJA KORZYSTAJĄCA Z ZEWNĘTRZNEGO LOGINU

Aplikacja kliencka, która przekierowuje użytkownika do zewnętrznego dostawcy (np. Google, GitHub, Microsoft).

Po zalogowaniu otrzymuje tokeny i tworzy sesję użytkownika.

Nie przechowuje haseł użytkowników (zajmuje się tym Authorization Server).

Popularne zastosowania:
– logowanie społecznościowe,
– integracja z firmowym SSO,
– dostęp do zewnętrznych API.

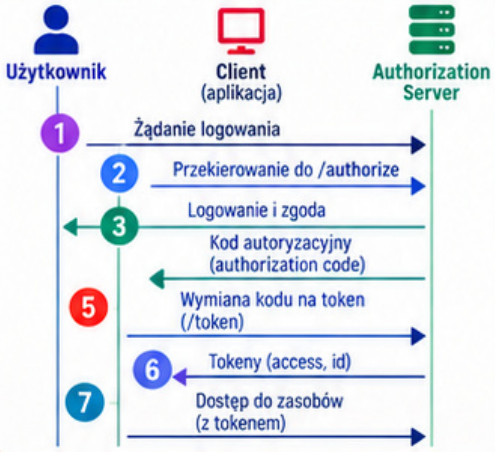
4 OAUTH2 LOGIN

JAK WYGLĄDA PROCES LOGOWANIA?

1. Użytkownik klika „Zaloguj przez ...”
2. Przekierowanie do Authorization Server.
3. Użytkownik loguje się i wyraża zgodę.
4. Server zwraca kod autoryzacyjny (authorization code).
5. Client wymienia kod na tokeny (access token, id token).
6. Tworzona jest sesja użytkownika w aplikacji.

5 AUTHORIZATION CODE FLOW

UPROSZCZONY PRZEBIEG KROK PO KROKU



6 OPENID CONNECT (OIDC)

WARSTWA TOŻSAMOŚCI NAD OAUTH2

OIDC to rozszerzenie OAuth2, które dodaje warstwę tożsamości (użytkownik + jego dane).

Zwraca id_token (JWT) z informacjami o użytkowniku (np. sub, email, name).

Umożliwia pobranie dodatkowych informacji przez endpoint /userinfo.

Używaj OIDC, gdy potrzebujesz potwierdzenia tożsamości użytkownika (np. logowanie, profil użytkownika).

OAuth2 = dostęp do zasobów
OIDC = dodatkowo tożsamość użytkownika

7 KIEDY UŻYĆ KTÓREGO ELEMENTU?

TYPOWE SCENARIUSZE

Login społecznościowy
Użyj OAuth2 Client + zewnętrzny dostawca (Google, GitHub, itp.).

Firmowe SSO
Użyj OIDC (np. Keycloak, Azure AD).

Własny serwer autoryzacji
Użyj Spring Authorization Server lub Keycloak.

Integracje B2B / API
Użyj OAuth2 (Client Credentials, Resource Server).

8 PRAKTYCZNA KONFIGURACJA CLIENTA

PRZYKŁAD W PLIK U APPLICATION.PROPERTIES

properties

```
spring.security.oauth2.client.registration.google
  .client-id=your-client-id
  .client-secret=your-client-secret
  .scope=openid,profile,email
  .redirect-uri={baseUrl}/login/oauth2/code/google
  .client-name=Google

spring.security.oauth2.client.provider.google
  .authorization-uri=https://accounts.google.com/oauth2/v2/auth
  .token-uri=https://oauth2.googleapis.com/token
  .user-info-uri=https://www.googleapis.com/oauth2/v3/userinfo
  .user-name-attribute=sub
```

Spring Boot automatycznie konfiguruje wiele ustawień na podstawie tych właściwości.

9 CLIENT VS RESOURCE SERVER VS AUTH SERVER

PORÓWNANIE RÓL I ODPOWIEDZIALNOŚCI

Cecha	Client	Resource Server	Authorization Server
Główna rola	Logowanie użytkownika	Udostępnia zasoby API	Uwierzytelnianie i wydawanie tokenów
Przechowuje użytkowników?	Nie	Nie	Tak (lub integruje się)
Weryfikuje tokeny?	Nie	Tak	Nie (wydaje)
Główne endpointy	/login /authorized	/api/**	/authorize /token /userinfo
Technologie	Spring Security OAuth2 Client	Spring Security Resource Server	Spring Authorization Server / Keycloak

10 PUŁAPKI

NA CO UWAŻAĆ?

- **Mylenie OAuth2 z autoryzacją lokalną**
  OAuth2 nie zastępuje własnego systemu użytkowników.
- **Błędne redirect URI**
  URI musi dokładnie się zgadzać z konfiguracją u dostawcy.
- **Brak walidacji scopes**
  Nadawaj tylko potrzebne uprawnienia.
- **Brak zrozumienia OIDC**
  OIDC dostarcza tożsamość (id_token), a nie tylko dostęp do API.

11 SZYBKIE PODSUMOWANIE

NAJWAŻNIEJSZE WNIOSKI

- Authorization Server uwierzytelnia i wydaje tokeny.
- Client odpowiada za logowanie użytkownika.
- Resource Server chroni zasoby i weryfikuje tokeny.
- Authorization Code Flow to najbezpieczniejszy przepływ.
- OIDC dodaje tożsamość użytkownika (id_token).
- Wybierz odpowiednie rozwiązanie do swojego przypadku.
- Poprawna konfiguracja i zrozumienie ról to klucz do bezpieczeństwa.
- Testuj i sprawdzaj uprawnienia (scopes).



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

STRATEGIA TESTÓW W SPRING BOOT

Jak łączyć szybkość, izolację i pewność działania aplikacji

MAPA 51/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Po co strategia testów?

Szybkie wykrywanie błędów
Wcześnie znajdujemy problemy.

Bezpieczny refactoring
Zmiany w kodzie bez obaw.

Większa pewność wdrożeń
Stabilna i niezawodna aplikacja na produkcji.

2 Piramida testów



Im niżej w piramidzie, tym testy są szybsze, tańsze w utrzymaniu i jest ich więcej.

3 Kiedy ładować Spring Context?

Ładuj Spring Context tylko, gdy:
- testujesz integrację ze Springiem
- potrzebujesz prawdziwych beanów
- sprawdzasz zachowanie całej aplikacji

Nie ładuj Spring Context, gdy:
- testujesz czystą logikę biznesową
- możesz użyć mocków
- zależy Ci na szybkości testów

Im mniej Spring Context, tym szybsze i bardziej niezawodne testy.

4 Test jednostkowy

- Izolowany test pojedynczej klasy
- Bez Spring Context
- Narzędzia: JUnit 5, Mockito, AssertJ
- Szybki i prosty w utrzymaniu

Przykład: Java

```java
@Test
void powinienZwrócicNazwe() {
  var serwis = new UserService();
  var wynik = serwis.getName(1L);
  assertThat(wynik).isEqualTo(
    "Jan");
}
```

5 Test slice

- Test konkretnej warstwy Spring
- Ładuje tylko potrzebne beany
- Przykłady: @WebMvcTest, @DataJpaTest, @JsonTest
- Szybszy niż pełny kontekst

Przykład (@WebMvcTest): Java

```java
@WebMvcTest(UserController.class)
class UserControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @Test
  void powinienZwrócic200() {
    mockMvc.perform(get("/users"))
      .andExpect(status().isOk());
  }
}
```

6 Test integracyjny

- Sprawdza współpracę komponentów
- Często używa @SpringBootTest
- Może korzystać z prawdziwej bazy lub Testcontainers
- Wolniejszy, ale daje większą pewność

Przykład: Java

```java
@SpringBootTest
class UserRepositoryIT {

  @Autowired
  private UserRepository repo;

  @Test
  void powinienZapisacUzytkownika() {
    var user = new User("Jan");
    repo.save(user);
    assertThat(repo.findById(user.getId()))
      .isPresent();
  }
}
```

7 Test systemowy / E2E

- Test całej aplikacji z perspektywy użytkownika / klienta
- Największa pewność działania
- Często @SpringBootTest z RANDOM_PORT i TestRestTemplate lub WebTestClient
- Najwolniejsze, uruchamiaj ich mniej

Przykład: Java

```java
@SpringBootTest(webEnvironment =
  SpringBootTest.WebEnvironment.
  RANDOM_PORT)
class UserE2ETest {
  @Autowired
  private TestRestTemplate rest;
  @Test
  void powinienZwrócicListe() {
    var resp = rest.getForEntity(
      "/users", String.class);
    assertThat(resp.getStatusCode())
  }   .isEqualTo(HttpStatus.OK);
```

8 Porównanie rodzajów testów

Rodzaj testów	Szybkość	Izolacja	Koszt	Pewność
Jednostkowe	Bardzo szybkie	Wysoka	Niski	Średnia
Slice	Szybkie	Wysoka	Niski	Średnia
Integracyjne	Wolniejsze	Średnia	Średni	Wysoka
Systemowe /E2E	Najwolniejsze	Niska	Wysoki	Najwyższa

9 Narzędzia i adnotacje

- **JUnit 5** – nowoczesny framework
- **Mockito** – mockowanie zależności
- **AssertJ** – czytelne asercje
- **@SpringBootTest** – pełny kontekst
- **@WebMvcTest** – testy kontrolerów
- **@DataJpaTest** – testy warstwy danych
- **MockMvc** – testy warstwy web (MVC)
- **Testcontainers** – prawdziwe usługi w testach (np. baza danych)

10 W praktyce

- Pisz dużo testów jednostkowych
- Dodaj wybrane testy slice
- Twórz ważne testy integracyjne
- Używaj kilku testów E2E
- Uruchamiaj testy automatycznie w CI (np. GitHub Actions)
- Dbaj o czas wykonania testów
- Regularnie refaktoryzuj testy

11 Pułapki

- Zbyt wiele ciężkich testów integracyjnych i E2E
- Testowanie frameworka zamiast własnej logiki
- Niestabilne (flaky) testy
- Zależność od kolejności wykonania testów
- Brak izolacji danych testowych
- Zbyt wolne testy utrudniające pracę zespołu

12 Szybkie podsumowanie

- Dobrze dobrana strategia testów zwiększa jakość kodu.
- Większość testów to testy jednostkowe.
- Testy slice dają szybkie potwierdzenie działania warstw.
- Testy integracyjne sprawdzają współpracę komponentów.
- Testy E2E dają najwyższą pewność, ale są najwolniejsze.
- Ładuj Spring Context tylko, gdy to konieczne.
- Automatyzuj testy w CI i dbaj o ich niezawodność.

Dobre testy to spokojniejszy developer i szczęśliwsi użytkownicy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

MAPA 52/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

TEST JEDNOSTKOWY BEZ SPRING CONTEXT

Jak szybko testować czystą logikę biznesową bez ładowania frameworka

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Co to jest test jednostkowy?

- **Testuje najmniejszy, izolowany fragment kodu** (np. jedną klasę lub metodę).
- **Sprawdza, czy kod działa** zgodnie z oczekiwaniami.
- **Szybki, prosty i niezależny od środowiska zewnętrznego** (np. bazy danych, Spring Context).

2 Dlaczego NIE ładować Spring Context?

- ✗ **Długi czas uruchamiania testów** (kilka sekund zamiast milisekund).
- ✗ **Większe zużycie zasobów** (pamięć, CPU).
- ✗ **Testujesz własną logikę, a nie framework.**
- ✗ **Trudniejsza diagnoza błędów** (więcej warstw, więcej szumu).
- ✓ **Bez Spring Context testy są szybsze, prostsze i bardziej stabilne.**

Używaj Spring Context tylko, gdy testujesz integrację komponentów Springa.

3 Co powinno być testowane?

- ✓ **Pojedyncza klasa** (np. serwis, util, walidator).
- ✓ **Logika biznesowa** (reguły, obliczenia, decyzje).
- ✓ **Różne scenariusze** (poprawne dane, wartości brzegowe, przypadki błędne).
- ✓ **Zachowanie w sytuacjach wyjątkowych** (np. błędy, puste dane, null).

Testuj publiczne zachowanie klasy, a nie szczegóły implementacji.

4 Izolacja i mockowanie

- ✓ Testuj klasę w izolacji.
- ✓ Zależności zastępuj mockami.
- ✓ Używaj Mockito do tworzenia atrap zależności (np. repozytoriów, klientów).
- ✓ Skup się na logice swojej klasy, nie na działaniu zewnętrznych komponentów.

5 JUnit 5 – podstawy

- ✓ **@Test** – oznacza metodę testową.
- ✓ **@BeforeEach** – kod wykonywany przed każdym testem.
- ✓ **@DisplayName** – czytelna nazwa testu.
- ✓ **Assertions** – wbudowane asercje (lub używaj AssertJ).

6 Mockito – najważniejsze elementy

- ✓ **@Mock** – tworzy mocka zależności.
- ✓ **@InjectMocks** – wstrzykuje mocki do testowanej klasy.
- ✓ **when(...).thenReturn(...)** – definiuje zachowanie mocka.
- ✓ **verify(...)** – sprawdza, czy metoda została wywołana.
- ✓ **ArgumentMatchers** – np. any(), eq().

7 AssertJ – czytelne asercje

- ✓ Bardziej czytelne niż standardowe asercje.
- ✓ Płynny, naturalny styl pisania.
- ✓ Lepsze komunikaty błędów.
- ✓ Przykłady:

```java
assertThat(result).isNotNull();
assertThat(lista).hasSize(3);
assertThat(wartosc).isEqualTo(10);
assertThat(tekst).contains("test");
assertThat(rzut).isInstanceOf(
                IllegalArgumentException.class);
```

8 Wzorzec Arrange – Act – Assert

1. **Arrange (Przygotuj)** – ustaw dane, mocki, stwórz obiekty.
2. **Act (Wykonaj)** – wywołaj testowaną metodę.
3. **Assert (Sprawdź)** – zweryfikuj wynik i zachowanie.

Ten wzorzec sprawia, że testy są czytelne, uporządkowane i łatwe do zrozumienia.

9 Przykład testu serwisu (bez Spring Context)

```java
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {
    @Mock
    private OrderRepository orderRepository;
    @InjectMocks
    private OrderService orderService;
    @Test
    @DisplayName("Powinien zwracać łączną kwotę zamówień")
    void shouldCalculateTotalAmount() {
        // Arrange
        List<Order> orders = List.of(
                new Order(1L, new BigDecimal("100.00")),
                new Order(2L, new BigDecimal("50.00"))
        );
        when(orderRepository.findAll()).thenReturn(orders);

        // Act
        BigDecimal total = orderService.calculateTotalAmount();

        // Assert
        assertThat(total).isEqualTo(new BigDecimal("150.00"));
    }   verify(orderRepository).findAll();
}
```

10 Kiedy testy jednostkowe są najlepsze?

- ✓ Gdy testujesz czystą logikę biznesową.
- ✓ Gdy chcesz szybko zweryfikować działanie kodu.
- ✓ Gdy zależności zewnętrzne są nieistotne.
- ✓ Gdy potrzebujesz dużej liczby szybkich testów.

11 Częste pułapki

- ✗ Testowanie szczegółów implementacji (zamiast publicznego zachowania).
- ✗ Zbyt wiele mocków (test staje się nienaturalny).
- ✗ Testowanie frameworka zamiast własnej logiki.
- ✗ Zbyt skomplikowane testy.
- ✗ Brak testów przypadków brzegowych.

12 Porównanie podejść

Kryterium	Test jednostkowy (bez Spring)	Test ze Spring Context
Szybkość	Bardzo szybki (milisekundy)	Wolny (sekundy)
Izolacja	Pełna	Częściowa
Koszt zasobów	Niski	Wysoki
Pewność wyniku	Wysoka (logika klasy)	Wyższa (integracja)

Szybkie podsumowanie

- ✓ Test jednostkowy sprawdza jedną klasę.
- ✓ Nie ładuj Spring Context, gdy nie jest potrzebny.
- ✓ Używaj Mockito do mockowania zależności.
- ✓ Stosuj czytelne asercje AssertJ.
- ✓ Pisz testy według wzorca Arrange–Act–Assert.
- ✓ Testuj scenariusze poprawne i błędne.
- ✓ Szybkie testy to szybszy rozwój i większa pewność.

W praktyce

- ✓ Pisz małe, skupione testy.
- ✓ Mockuj tylko to, co konieczne.
- ✓ Używaj czytelnych nazw testów.
- ✓ Testuj również przypadki brzegowe.
- ✓ Uruchamiaj testy często (np. po każdej zmianie).
- ✓ Trzymaj testy blisko kodu.



Małe testy duże możliwości – szybszy kod, szczęśliwszy deweloper!

Pisz testy
Buduj lepsze jutro!

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

@SPRINGBOOTTEST

Kiedy uruchamiać pełny kontekst aplikacji i jak robić to świadomie

MAPA 53/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Co robi @SpringBootTest?

Uruchamia pełny kontekst aplikacji Spring Boot.
Wczytuje całą konfigurację, beany, auto-konfigurację i środowisko.

Umożliwia testowanie aplikacji w sposób jak najbliższy produkcji.
Działa jak właściwa aplikacja, ale w środowisku testowym.

Automatycznie wyszukuje klasę główną @SpringBootApplication.
Można też wskazać ją jawnie przez classes = ...

2 Pełny kontekst aplikacji

Test z @SpringBootTest
↓
Auto-konfiguracja Spring Boot
Wszystkie beany (serwisy, repozytoria itp.)
Konfiguracja aplikacji (application.yml)
Profile i właściwości
Infrastruktura (baza danych, kolejki, web, ...)

To ta sama konfiguracja, co w uruchomionej aplikacji, więc testy są bardzo realistyczne.

3 Typowe zastosowania

- Testy end-to-end całej aplikacji.
- Testowanie warstwy web (kontrolery + filtry + security).
- Sprawdzenie integracji wielu komponentów.
- Testy z rzeczywistą bazą danych, kolejkami, usługami.
- Weryfikacja konfiguracji aplikacji.
- Testy scenariuszy biznesowych "od A do Z".

Używaj **@SpringBootTest**, gdy potrzebujesz pewności, że cała aplikacja działa razem, tak jak w środowisku produkcyjnym.

4 Tryby webEnvironment

MOCK — **Domyślny tryb.** Uruchamia kontekst web, ale bez prawdziwego serwera. Używa MockMvc.

RANDOM_PORT — **Uruchamia prawdziwy serwer na losowym porcie.** Dobre do testów HTTP z TestRestTemplate lub WebTestClient.

DEFINED_PORT — **Uruchamia prawdziwy serwer na wskazanym porcie.** Przydatne w specyficznych scenariuszach.

NONE — **Nie uruchamia warstwy web.** Tylko kontekst aplikacji bez serwera www. Szybszy start, gdy web nie jest potrzebny.

5 Kiedy używać @SpringBootTest?

- Gdy testujesz wiele warstw jednocześnie.
- Gdy potrzebujesz rzeczywistych beanów.
- Gdy testujesz integrację z bazą danych lub innymi usługami.
- Gdy zależy Ci na maksymalnej zgodności z produkcją.
- Gdy testy lżejsze (np. slice) są niewystarczające.

Używaj świadomie – tylko tam, gdzie wartości dodanej nie dają prostsze testy.

6 Zalety i wady

Zalety

- Realistyczne środowisko testów.
- Weryfikacja całej konfiguracji.
- Wykrywa błędy integracji.
- Testy bardzo zbliżone do środowiska produkcyjnego.
- Daje dużą pewność działania aplikacji.

Wady

- Długi czas uruchamiania.
- Większe zużycie zasobów (pamięć, CPU).
- Wymaga konfiguracji zewnętrznych zasobów (np. bazy danych).
- Bardziej kruche testy.
- Nie nadaje się do szybkiego feedbacku podczas codziennej pracy.

7 Koszt uruchomienia

Długi czas startu kontekstu.
Pełny kontekst może uruchamiać się od kilku do kilkunastu sekund.

Testy są cięższe.
Większe zużycie pamięci i CPU, szczególnie przy wielu testach.

Warto ograniczać liczbę takich testów.
Stosuj je głównie dla kluczowych scenariuszy integracyjnych.

8 Rzeczywiste beany i zasoby

Ładuje prawdziwe beany z kontekstu.
Serwisy, repozytoria, klienty HTTP, kolejki, konfiguracja bezpieczeństwa itd.

Może łączyć się z prawdziwymi zasobami.
Baza danych, RabbitMQ, Kafka, zewnętrzne API (w zależności od konfiguracji).

Można podmieniać beany na testowe.
Użyj @MockBean, @SpyBean lub własnej konfiguracji testowej.

9 Konfiguracja i profile testowe

Używaj profili testowych.
Dodaj plik application-test.yml i uruchamiaj testy z @ActiveProfiles("test").

Możesz nadpisać właściwości.
Użyj @TestPropertySource lub właściwości w adnotacji @SpringBootTest.

Dodatkowa konfiguracja testowa.
Twórz klasy @TestConfiguration, aby dodać lub zmienić beany na potrzeby testów.

10 Przykład testu

Java

```java
@SpringBootTest(webEnvironment =
    SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("test")
class UserControllerIT {
    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void shouldReturnUsers() {
        var response = restTemplate.getForEntity(
            "/api/users", String.class);
        assertThat(response.getStatusCode())
            .isEqualTo(HttpStatus.OK);
        assertThat(response.getBody())
            .contains("Jan");
}
```

11 Relacja do innych testów

Rodzaj testów	Zakres	Szybkość	Kiedy używać?
@SpringBootTest	Cała aplikacja	Wolne	Integracja, E2E, pełna pewność
@WebMvcTest	Warstwa web (kontrolery)	Szybkie	Testy kontrolerów, MockMvc
@DataJpaTest	Warstwa danych	Szybkie	Testy repozytoriów
@JdbcTest	Dostęp do DB (JDBC)	Szybkie	Testy zapytań JDBC
@Mockito / @Test (unit)	Pojedyncze klasy	Bardzo szybkie	Logika biznesowa, izolowane testy

Najpierw używaj testów lżejszych. Sięgaj po @SpringBootTest tylko wtedy, gdy naprawdę potrzebujesz pełnego kontekstu.

12 Najczęstsze pułapki

- Używanie @SpringBootTest do prostych testów jednostkowych.
- Brak profilu testowego (ryzyko połączenia z produkcją!).
- Zbyt wiele testów z pełnym kontekstem.
- Niepotrzebne uruchamianie serwera web.
- Brak izolacji danych testowych.
- Zależność od niestabilnych zasobów zewnętrznych.
- Długie i kruche testy utrudniające rozwój.

13 Najlepsze praktyki – szybkie podsumowanie

- Używaj @SpringBootTest tylko tam, gdzie jest to konieczne.
- Preferuj testy lżejsze (slice i jednostkowe).
- Korzystaj z profili testowych i oddzielnych konfiguracji.
- Ogranicz liczbę testów z pełnym kontekstem.
- W miarę możliwości używaj kontenerów testowych (Testcontainers).
- Podmieniaj zewnętrzne zasoby (@MockBean, @SpyBean).
- Dbaj o niezależność i powtarzalność testów.
- Regularnie refaktoryzuj testy, aby były szybkie i stabilne.

Świadome testy to stabilniejsza aplikacja i spokojniejszy developer!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

TEST SLICES W SPRING BOOT

Jak testować wycinek aplikacji szybciej niż pełny kontekst

MAPA 54/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym są testy slice?

Testy slice (wycinkowe)
ładują tylko część kontekstu Springa, potrzebną do testów.

To specjalne adnotacje testowe, które konfigurują minimalny kontekst aplikacji.

Dzięki temu testy są szybsze, prostsze w utrzymaniu i bardziej skupione na danym komponencie.

2 Po co używać testów slice?

- Szybsze uruchamianie testów.
- Mniejszy kontekst = mniejsze zużycie zasobów.
- Testy skupione na konkretnej warstwie.
- Łatwiejsze diagnozowanie problemów.
- Mniej konfiguracji, mniej przypadkowych błędów.
- Idealne do testów kontrolerów, repozytoriów, serializacji JSON itp.
- Uzupełniają testy jednostkowe i testy integracyjne (@SpringBootTest).

3 Najpopularniejsze testy slice

@WebMvcTest	Testy kontrolerów MVC (warstwa web).
@DataJpaTest	Testy repozytoriów JPA (warstwa danych).
@JsonTest	Testy serializacji i deserializacji JSON (ObjectMapper).
@RestClientTest	Testy klientów HTTP (RestTemplate, WebClient, RestClient).
@JdbcTest	Testy dostępu do bazy przez JDBC (JdbcTemplate) – opcjonalnie.

4 Co jest ładowane, a co nie?

Ładowane (zależnie od slice)

- Wybrane auto-konfiguracje Spring Boot.
- Komponenty specyficzne dla danej warstwy.
- Minimalny kontekst potrzebny do działania testów.

Nie jest ładowane

- Pełny kontekst aplikacji.
- Niepotrzebne beany z innych warstw (np. serwisy, kontrolery w testach JPA).
- Konfiguracje produkcyjne.
- Cała aplikacja jak w @SpringBootTest.

5 Mockowanie zależności

W testach slice często trzeba zamockować beany z innych warstw (np. serwisy w @WebMvcTest).

Używaj **@MockBean** do tworzenia mocków w kontekście Springa.

Dzięki temu testujesz tylko to, co Cię interesuje, bez zbędnych zależności.

6 Typowe scenariusze użycia

- Testy kontrolerów REST (bez warstwy serwisów).
- Testy repozytoriów JPA (z prawdziwą bazą w pamięci).
- Testy mapowania JSON (serialize/deserialize).
- Testy klientów HTTP (wywołania do zewnętrznych API).
- Testy zapytań SQL przez JdbcTemplate.
- Szybkie sprawdzenie konfiguracji konkretnej warstwy.

7 Przykłady adnotacji i kodu

@WebMvcTest — Test kontrolera MVC

```
// Test kontrolera
@WebMvcTest(UserController.class)
class UserControllerTest {
  @Autowired
  private MockMvc mockMvc;
  @MockBean
  private UserService userService;
  @Test
  void shouldReturnUser() throws
   Exception {
    when(userService.findById(1L))
      .thenReturn(new User(1L, "Jan"));
    mockMvc.perform(get("/users/1"))
     .andExpect(status().isOk());
  }
}
```

@DataJpaTest — Test repozytorium JPA

```
@DataJpaTest
class UserRepositoryTest {

  @Autowired
  private UserRepository repo;

  @Test
  void shouldSaveUser() {
    User user = new User(null, "Jan");
    User saved = repo.save(user);

    assertThat(saved.getId())
     .isNotNull();
    assertThat(repo.findById(saved
     .getId())).isPresent();
  }
}
```

@JsonTest — Test serializacji JSON

```
@JsonTest
class UserJsonTest {

  @Autowired
  private ObjectMapper objectMapper;

  @Test
  void shouldSerializeUser() throws
   Exception {
    User user = new User(1L, "Jan");

    String json =
     objectMapper.writeValueAsString(user);

    assertThat(json).contains("Jan");
  }
}
```

@RestClientTest — Test klienta HTTP

```
@RestClientTest(ExternalApiClient.class)
class ExternalApiClientTest {
  @Autowired
  private ExternalApiClient client;
  @Autowired
  private MockRestServiceServer server;
  @Test
  void shouldCallExternalApi() {
   server.expect(requestTo("/api/users/1"))
    .andRespond(withSuccess("{\"id\":1}",
     MediaType.APPLICATION_JSON));
   User user = client.getUser(1L);
   assertThat(user.getId()).isEqualTo(1L);
  }
```

8 Porównanie podejść

Cecha	Test jednostkowy	Test slice	@SpringBootTest
Ładowany kontekst	Brak (czysty JVM)	Częściowy	Pełny kontekst
Szybkość	Bardzo szybkie	Szybkie	Najwolniejsze
Zakres testów	Pojedyncza klasa	Wycinek warstwy	Cała aplikacja
Zależności	Mocki (np. Mockito)	@MockBean	Prawdziwe beany
Przeznaczenie	Logika jednostkowa	Testy warstw	Testy integracyjne
Użycie bazy danych	Nie	Tak (np. H2)	Tak
Przykład	UserServiceTest	@DataJpaTest	@SpringBootTest

9 Przegląd testów slice

Adnotacja	Ładuje (głównie)	Nie ładuje (np.)
@WebMvcTest	Kontrolery MVC, MockMvc, konwertery	Serwisów, repozytoriów, widoków Thymeleaf
@DataJpaTest	Repozytoria JPA, EntityManager, H2	Kontrolerów, serwisów, warstwy web
@JsonTest	ObjectMapper, moduły JSON	Kontrolerów, repozytoriów, serwisów
@RestClientTest	RestTemplate / WebClient / RestClient, MockServer	Całej aplikacji, serwisów, repozytoriów
@JdbcTest	JdbcTemplate, DataSource, H2	Kontrolerów, serwisów, JPA (EntityManager)

10 Typowe pułapki

- Oczekiwanie, że test slice załaduje całą aplikację.
- Brak mocków dla wymaganych zależności.
- Testowanie logiki serwisów w @WebMvcTest (zamiast mockowania).
- Niepoprawna konfiguracja bazy w @DataJpaTest.
- Zbyt wiele własnych konfiguracji, które niwelują zalety slice.
- Używanie @SpringBootTest, gdy wystarczy slice.

11 Dobre praktyki

- Używaj testów slice jako domyślnego wyboru dla testów warstw.
- Mockuj tylko to, co naprawdę jest potrzebne.
- Utrzymuj testy szybkie, proste i niezależne.
- Stosuj czytelne nazwy testów i metod.
- W razie potrzeby używaj @Import do dołączenia dodatkowej konfiguracji.
- Łącz testy slice z testami jednostkowymi i integracyjnymi.

12 Szybkie podsumowanie

- Testy slice = szybkie testy wycinka kontekstu.
- Do dyspozycji m.in. @WebMvcTest, @DataJpaTest, @JsonTest, @RestClientTest, @JdbcTest.
- Ładują tylko potrzebne komponenty.
- Wymagają często mockowania zależności.
- Idealne do testowania warstw aplikacji.
- Są szybsze i lżejsze niż @SpringBootTest.
- To kluczowy element skutecznej strategii testów.

Szybsze testy to szybszy rozwój – testuj mądrze, nie ciężej!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

TESTOWANIE REST Z MOCKMVC

Jak sprawdzać kontrolery i endpointy MVC bez uruchamiania prawdziwego serwera

MAPA 55/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest MockMvc?

- MockMvc to narzędzie testowe z Spring MVC.
- Umożliwia symulowanie żądań HTTP bez uruchamiania prawdziwego serwera.
- Pozwala testować kontrolery, filtry, obsługę błędów, walidację i serializację JSON.
- Działa szybko i niezależnie od warstwy sieciowej.

2 Dlaczego warto stosować?

- **Szybkie testy** (brak uruchamiania serwera)
- **Skupienie na warstwie web** (kontrolery, mapowania, walidacja)
- **Łatwe testowanie sytuacji brzegowych i błędów**
- **Większa pewność działania API**
- Dobrze współpracuje z JUnit 5 i ekosystemem Spring Boot

3 @WebMvcTest vs pełny kontekst

@WebMvcTest (zalecane)	Pełny kontekst (@SpringBootTest)
Ładuje tylko warstwę MVC	Ładuje całą aplikację
Szybkie uruchamianie testów	Wolniejsze testy
Automatycznie konfiguruje MockMvc	Może być użyteczne w testach integracyjnych
Wymaga @MockBean dla zależności (serwisy itp.)	Rzadziej potrzebne do samych kontrolerów
Idealne do testów kontrolerów	Większe zużycie zasobów

Do testów kontrolerów zwykle wystarczy **@WebMvcTest**. Używaj pełnego kontekstu tylko, gdy testujesz całą aplikację.

4 Wykonywanie żądań HTTP

GET	get("/api/users") – pobierz zasoby
POST	post("/api/users") – dodaj nowy zasób
PUT	put("/api/users/1") – aktualizuj zasób
DELETE	delete("/api/users/1") – usuń zasób

Możesz dodawać nagłówki, parametry, treść żądania i oczekiwania (assertions).

5 Przepływ żądania w teście

MockMvc (symulacja żądania) → DispatcherServlet (łańcuch filtrów) → Kontroler (@RestController)

Żądanie HTTP
- metoda (GET/POST itd.)
- ścieżka
- nagłówki
- parametry
- ciało (JSON)

Odpowiedź HTTP
- status
- nagłówki
- ciało (JSON)
- obsługa błędów

6 Parametry ścieżki i zapytania

Parametry ścieżki (Path)

```
get("/api/users/{id}", 1)
```

- Przekazywanie wartości do {id} w URL
- Weryfikacja poprawnego mapowania

Parametry zapytania (Query)

```
get("/api/users")
  .param("active", "true")
  .param("page", "0")
```

- Dodawanie wielu parametrów
- Weryfikacja obsługi filtrowania i paginacji

Używaj **.param()** dla parametrów zapytania i wartości w nawiasach klamrowych dla parametrów ścieżki.

7 Sprawdzanie odpowiedzi

- **Status HTTP** (isOk, isCreated, isBadRequest...)
- **Nagłówki** (header, contentType...)
- **Treść JSON** (jsonPath, content...)
- **Błędy walidacji** (pola i komunikaty)
- **Przekierowania i lokalizacje** (redirectedUrl)
- **Własne matchery** (hamcrest)

8 Mockowanie warstwy serwisu

W testach kontrolerów zwykle mockujemy zależności, np. serwisy, używając @MockBean.

Przykład: Java

```java
@WebMvcTest(UserController.class)
class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void shouldReturnUser() {
        when(userService.findById(1L))
            .thenReturn(new UserDto(1L, "Jan"));
        mockMvc.perform(get("/api/users/1"))
            .andExpect(status().isOk());
    }
}
```

9 StandaloneSetup vs z kontekstem

standaloneSetup()
- Nie wymaga kontekstu Springa
- Ręczna konfiguracja kontrolerów
- Szybsze i lżejsze testy
- Dobre do prostych przypadków

Przykład: Java

```java
MockMvc mockMvc =
  MockMvcBuilders.standaloneSetup(
      new UserController())
    .build();
```

Z kontekstem (np. @WebMvcTest)
- Automatyczna konfiguracja
- Obsługa filtrów, poradników itp.
- Bardziej zbliżone do rzeczywistej aplikacji
- Zalecane w większości przypadków

Przykład: Java

```java
@WebMvcTest(UserController.class)
class UserControllerTest {
  @Autowired
  private MockMvc mockMvc;
}
```

10 Przykładowy test kontrolera

Przykład pełnego testu: Java

```java
@WebMvcTest(UserController.class)
class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private UserService userService;
    @Test
    void shouldReturnUserJson() throws Exception {
        UserDto dto = new UserDto(1L, "Jan", "jan@test.pl");
        when(userService.findById(1L)).thenReturn(dto);
        mockMvc.perform(get("/api/users/1"))
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.name").value("Jan"))
        .andExpect(jsonPath("$.email").value("jan@test.pl"));
}
```

11 Najczęstsze problemy (pułapki)

- Brak @MockBean dla zależności (serwisy, repozytoria).
- Używanie pełnego kontekstu, gdy wystarczy @WebMvcTest.
- Błędne ścieżki URL lub metody HTTP.
- Niezgodny typ treści (Content-Type).
- Brak weryfikacji treści JSON.
- Pominięcie testów błędów walidacji.
- Zbyt ogólne asercje (np. tylko status 200).
- Niestabilne testy przez przypadkowe uzależnienie od danych.

12 Wskazówki praktyczne

- Używaj @WebMvcTest dla testów kontrolerów.
- Mockuj serwisy za pomocą @MockBean.
- Sprawdzaj status, nagłówki i treść JSON.
- Testuj zarówno scenariusze pozytywne, jak i błędne.
- Weryfikuj walidację (@Valid) i komunikaty błędów.
- Używaj jsonPath do sprawdzania pól w odpowiedzi.
- Twórz czytelne, niezależne testy.
- Korzystaj z fabryk danych (buildery, fixture'y).
- Utrzymuj testy szybkie i stabilne.

13 Szybkie podsumowanie

- MockMvc pozwala testować kontrolery bez serwera.
- Używaj @WebMvcTest i @MockBean.
- Sprawdzaj status, nagłówki, treść, walidację i parametry.
- Wybierz standaloneSetup do prostych testów.
- Testuj przypadki pozytywne i błędne.
- Utrzymuj testy czytelne i szybkie.
- To podstawa solidnego i niezawodnego API.
- Dobre testy to spokojniejszy zespół i szczęśliwsi użytkownicy ::)

Dobre testy to spokojniejszy developer i szczęśliwsi użytkownicy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

MAPA 56/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

TESTOWANIE APLIKACJI PRZEZ PRAWDZIWE HTTP

Kiedy używać TestRestTemplate, WebTestClient i uruchomionej aplikacji

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Na czym polega testowanie HTTP?

Testy wysyłają prawdziwe żądania **HTTP** do uruchomionej aplikacji (Spring Boot) i sprawdzają odpowiedzi.

Testujemy **całą aplikację** w sposób zbliżony do rzeczywistego użycia (endpointy, kontrolery, filtry, serializacja, konfiguracja).

Większa pewność, że aplikacja działa poprawnie w **realnym** środowisku.

2 Kiedy używać @SpringBootTest z RANDOM_PORT?

- Gdy chcesz uruchomić całą aplikację i testować ją przez prawdziwy HTTP.
- Gdy w testach potrzebujesz rzeczywistego kontekstu (kontrolery, filtery, bezpieczeństwo, konfiguracja).
- Idealne do testów API i scenariuszy end-to-end.
- Port jest losowy (RANDOM_PORT), więc testy mogą działać równolegle.
- Wolniejsze niż testy slice, ale dają największą pewność.

```
@SpringBootTest(webEnvironment =
   SpringBootTest.WebEnvironment.RANDOM_PORT)
class ApiHttpTest {
    // Testy przez prawdziwe HTTP
}
```

3 TestRestTemplate – kiedy używać?

- **Synchroniczny, prosty** w użyciu klient HTTP dla aplikacji servletowej (MVC).
- Świetny do klasycznych aplikacji Spring Boot (opartych na Spring MVC).
- Prosty, czytelny API, dobrze nadaje się do większości przypadków testowych.
- Obsługuje autoryzację, nagłówki, JSON.
- Polecany, gdy nie potrzebujesz reaktywności ani strumieniowania.

```
@Autowired
private TestRestTemplate restTemplate;

ResponseEntity<String> response =
 restTemplate.getForEntity("/api/users", String.class);
```

4 WebTestClient – kiedy używać?

- Nowoczesny, bardziej elastyczny klient HTTP.
- Działa z aplikacjami WebFlux (reaktywnymi) oraz z aplikacjami servletowymi (MVC) od Spring Boot 2.2+.
- Wspiera styl reaktywny, możliwość testowania strumieni (np. SSE).
- Płynne, czytelne API z asercjami.
- Wybierz, gdy używasz WebFlux lub chcesz nowocześniejszego stylu testów.

Java

```
@Autowired
private WebTestClient webTestClient;

webTestClient.get().uri("/api/health")
  .exchange()
  .expectStatus().isOk()
  .expectBody().jsonPath("$.status").isEqualTo("UP");
```

5 Żądania, odpowiedzi i asercje

- Testuj różne metody HTTP: GET, POST, PUT, DELETE, PATCH.
- Sprawdzaj kody statusu, nagłówki i treść odpowiedzi (JSON).
- Używaj asercji na konkretne pola JSON.
- Weryfikuj typ zawartości (Content-Type).
- Testuj scenariusze sukcesu i błędu.

Java

```
ResponseEntity<UserDto> response =
  restTemplate.postForEntity(
    "/api/users",
    new CreateUserRequest("Jan", "kowalski"),
    UserDto.class);

assertThat(response.getStatusCode())
    .isEqualTo(HttpStatus.CREATED);
assertThat(response.getBody().getName())
    .isEqualTo("Jan");
```

6 Nagłówki, JSON i autoryzacja

- Ustawiaj własne nagłówki (np. Accept, Content-Type, Authorization).
- Wysyłaj i sprawdzaj JSON.
- Testuj zabezpieczone endpointy (np. z JWT lub Basic Auth).
- Możesz używać tokenów wygenerowanych w testach lub testowej konfiguracji bezpieczeństwa.

Java

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.
    APPLICATION_JSON);
headers.setBearerAuth("test-token");

HttpEntity<CreateUserRequest> request =
    new HttpEntity<>(new CreateUserRequest(
      "Jan", "kowalski"), headers);

restTemplate.postForEntity("/api/users",
    request, UserDto.class);
```

7 Testy end-to-end API

- Testują pełny przepływ: HTTP → kontrolery → serwis → baza danych.
- Najlepsze do weryfikacji kluczowych scenariuszy użytkownika.
- Sprawdzają integrację wielu komponentów aplikacji.
- Używaj prawdziwej bazy testowej (np. H2, Testcontainers).
- Dłuższe w wykonaniu, ale dają największą pewność działania.



8 TestRestTemplate vs WebTestClient

Cecha	TestRestTemplate	WebTestClient
Styl	Synchroniczny	Reaktywny/Synchroniczny
Docelowa aplikacja	Spring MVC (servletowa)	WebFlux i MVC
Łatwość użycia	Bardzo prosta	Bardzo dobra
Streaming / SSE	Ograniczone wsparcie	Pełne wsparcie
Popularność	Bardzo wysoka	Rosnąca
Kiedy używać?	Klasyczne API (Spring MVC)	Aplikacje reaktywne lub nowoczesne testy

9 Dane testowe i środowisko

- Używaj oddzielnej bazy danych testowej (H2, Testcontainers, lub innej).
- Przygotuj dane testowe przed testem (np. @Sql, Testcontainers, @BeforeEach).
- Czyść dane po teście, aby testy były niezależne.
- Zadbaj o izolację testów (każdy test własne dane).
- Używaj profilu testowego (application-test.yml).

Java

```
@ActiveProfiles("test")
@SpringBootTest(webEnvironment =
    SpringBootTest.WebEnvironment.RANDOM_PORT)
class UserApiE2ETest {
    // ...
}
```

10 Pułapki i najczęstsze błędy

- Zapomnienie o RANDOM_PORT (i kolizjach portów).
- Używanie testów HTTP tam, gdzie wystarczą testy slice (są szybsze).
- Brak izolacji danych – testy zależne od siebie.
- Twardo zakodowane dane i ID.
- Ignorowanie błędów i brak asercji na odpowiedź.
- Zbyt duża liczba testów end-to-end (spowalnia build).
- Testowanie logiki biznesowej tylko przez HTTP (lepiej testy jednostkowe).

11 Dobre praktyki

- Używaj RANDOM_PORT.
- Testuj tylko kluczowe scenariusze end-to-end.
- Utrzymuj niezależne i powtarzalne testy.
- Sprawdzaj status, nagłówki i treść odpowiedzi.
- Korzystaj z profilu testowego i testowej bazy danych.
- Stosuj czytelne nazwy testów i metod.
- Łącz testy HTTP z testami slice i jednostkowymi.

12 Szybkie podsumowanie

- **TestRestTemplate** – proste testy HTTP dla aplikacji Spring MVC.
- **WebTestClient** – nowoczesny klient do WebFlux i MVC.
- Używaj **@SpringBootTest** z RANDOM_PORT, aby testować prawdziwe HTTP.
- Sprawdzaj statusy, nagłówki, JSON i autoryzację.
- Stosuj dane testowe i oddzielne środowisko.
- Unikaj nadmiernej liczby testów end-to-end.
- Wybierz narzędzie odpowiednie do typu aplikacji.
- Dobrze zaprojektowane testy HTTP dają dużą pewność działania.

Prawdziwe żądania HTTP to testy bliższe rzeczywistości i zadowoleni użytkownicy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

TESTOWANIE REPOZYTORIÓW I BAZY DANYCH

Jak wiarygodnie sprawdzać warstwę danych i zapytania

MAPA 57/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Po co testować repozytoria?

Weryfikacja poprawności dostępu do danych
Czy zapytania zwracają właściwe wyniki?

Wykrywanie błędów na wczesnym etapie
Błędy zapytań, mapowania, ograniczeń.

Pewność zmian
Refaktoryzacja encji i zapytań bez ryzyka regresji.

2 @DataJpaTest

@DataJpaTest

- Specjalna adnotacja do testów warstwy JPA (tylko komponenty związane z bazą danych).
- Szybki start kontekstu Spring.
- Domyślnie używa wbudowanej bazy (H2).
- Transakcje są automatycznie rollbackowane.
- Można importować dodatkowe konfiguracje (np. własne konwertery).

```java
@DataJpaTest
class ProductRepositoryTest {
    @Autowired
    private ProductRepository repository;
}
```

3 Wbudowana czy prawdziwa baza?

Cecha	Wbudowana (H2)	Prawdziwa baza (Testcontainers)
Szybkość	Bardzo szybka	Wolniejsza
Wierność działania	Może różnić się od produkcji	Taka sama jak na produkcji
Konfiguracja	Prosta (brak dodatkowych narzędzi)	Wymaga Testcontainers
Zastosowanie	Testy jednostkowe repozytoriów, szybkie iteracje	Testy integracyjne, weryfikacja zgodności z produkcją
Przykład	H2 (w pamięci)	PostgreSQL, MySQL, itp. (Docker)

4 Transakcje i rollback

Testy @DataJpaTest są domyślnie wykonywane w transakcji.

Po zakończeniu testu następuje automatyczny rollback.

Dane nie zostają w bazie, każdy test jest odizolowany.

W razie potrzeby użyj @Commit lub @Rollback(false) aby zachować dane (rzadko).

5 Co testować?

- Mapowania encji (relacje, kolumny).
- Zapytania metodami repozytoriów (np. findBy...).
- Własne zapytania @Query (JPQL/SQL).
- Ograniczenia i walidację (np. unikalność).
- Operacje zapisu, aktualizacji i usuwania.
- Zachowanie relacji (OneToMany, ManyToOne itp.).
- Obsługę sortowania, paginacji, projectionów.

6 TestEntityManager

TestEntityManager to uproszczony interfejs do operacji na encjach w testach.

- Ułatwia zapisywanie, wyszukiwanie i usuwanie encji.
- Pozwala wymusić flush() i clear().
- Pomaga testować zapytania w realistyczny sposób.

```java
@Autowired
private TestEntityManager em;

Product p = em.persistAndFlush(
  new Product("Laptop", new BigDecimal("2999.00"))
);
```

7 Weryfikacja zapytań

Testuj zapytania metodowe (np. findByNazwisko, findByStatus).

Testuj własne zapytania @Query (JPQL lub natywne SQL).

Sprawdzaj parametry, sortowanie i paginację.

Weryfikuj, że zwracane są dokładnie oczekiwane dane.

Sprawdzaj zapytania z joinami, warunkami i funkcjami agregującymi.

8 Przygotowanie schematu i danych

Użyj schema.sql i data.sql (lub import.sql dla H2).

Możesz też przygotowywać dane programatycznie (TestEntityManager).

Dbaj o czytelne i minimalne dane testowe.

Dla bardziej złożonych scenariuszy użyj @Sql lub bibliotek takich jak DBUnit / Database Rider.

9 Przykładowy test repozytorium

```java
@DataJpaTest
class ProductRepositoryTest {
    @Autowired
    private ProductRepository repository;
    @Autowired
    private TestEntityManager em;

    @Test
    void powinienZnalezcProduktPoNazwie() {
        em.persistAndFlush(new Product("Laptop", new BigDecimal("2999.00")));
        Optional<Product> wynik = repository.findByNazwa("Laptop");
        assertThat(wynik).isPresent();
        assertThat(wynik.get().getNazwa()).isEqualTo("Laptop");
    }
}
```

10 Pułapki

- Fałszywe poczucie pewności przy H2 – niektóre zapytania działają tylko w H2.
- Różnice w składni SQL, funkcjach i typach danych (np. JSON, UUID).
- Brak testów unikalnych ograniczeń.
- Zbyt proste dane testowe.
- Testowanie zbyt dużej logiki biznesowej w repozytoriach.
- Brak weryfikacji wydajności złożonych zapytań (N+1, joiny).

11 Najlepsze praktyki

- Stosuj @DataJpaTest do szybkich testów repozytoriów.
- Używaj Testcontainers dla kluczowych scenariuszy.
- Testuj zarówno zapytania metodowe, jak i @Query.
- Przygotowuj czytelne dane testowe.
- Weryfikuj ograniczenia i mapowania encji.
- Utrzymuj małe, niezależne i szybkie testy.
- Stosuj asercje na zawartość kolekcji (np. assertThat).

12 Szybkie podsumowanie

- Testy repozytoriów dają pewność poprawności warstwy danych.
- @DataJpaTest + H2 = szybkie testy.
- Testcontainers = zgodność z produkcją.
- Transakcje i rollback daje izolację.
- Sprawdzaj mapowania, zapytania, ograniczenia.
- TestEntityManager ułatwia pracę z danymi.
- Unikaj pułapek i stosuj dobre praktyki.



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

TESTCONTAINERS

Prawdziwe usługi w testach: bazy danych, brokery i więcej

MAPA 58/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest Testcontainers?

- Biblioteka Java do uruchamiania rzeczywistych usług w kontenerach Docker podczas testów.
- Pozwala na testy integracyjne z prawdziwymi zależnościami (np. bazami danych, brokerami).
- Automatyzuje pobieranie obrazów, konfigurację i zarządzanie cyklem życia kontenerów.
- Testy działają tak samo jak w środowisku produkcyjnym, ale są izolowane i powtarzalne.

2 Dlaczego warto używać?

Bardziej realistyczne testy
Prawdziwe usługi zamiast mocków.

Mniej problemów na produkcji
Testujesz rzeczywiste działanie.

Prosta konfiguracja
Mało kodu, duże możliwości.

Izolacja i powtarzalność
Każdy test ma własne, czyste środowisko.

Wsparcie dla wielu technologii
Bazy danych, brokery, cache, search i więcej.

3 Przykładowe wspierane usługi

PostgreSQL	MySQL	Redis	Kafka
postgres:16	mysql:8	redis:7	kafka:7

RabbitMQ	MongoDB	Elasticsearch	LocalStack
rabbitmq:3	mongo:7	elasticsearch:8	localstack:latest

Testcontainers wspiera setki modułów. Możesz też używać dowolnych obrazów Docker z Docker Hub.

4 Jak to działa?

Test uruchamia kontener
↓
Pobierany jest obraz Docker (jeśli nie istnieje lokalnie)
↓
Kontener startuje i jest konfigurowany
↓
Testy korzystają z prawdziwej usługi
↓
Po zakończeniu testów kontener jest usuwany

5 Deklaracja kontenera w testach

- Użyj gotowych klas z biblioteki (np. PostgreSQLContainer).
- Skonfiguruj wersję obrazu i ewentualne parametry (np. baza, użytkownik).
- Możesz używać kontenerów w pojedynczym tescie lub udostępniać je między testami.

Przykład: tworzenie kontenera PostgreSQL — Java

```java
PostgreSQLContainer<?> postgres =
  new PostgreSQLContainer<>("postgres:16")
    .withDatabaseName("testdb")
    .withUsername("test")
    .withPassword("test");
```

6 @Container i pola static

- Użyj adnotacji **@Container**, aby oznaczyć pole jako kontener.
- Najczęściej pole jest **static**, aby kontener był współdzielony dla wszystkich testów w klasie.
- Możesz też użyć pola niestatycznego – wtedy kontener będzie tworzony dla każdego testu.

Przykład z @Container — Java

```java
static @Container
static PostgreSQLContainer<?> postgres =
  new PostgreSQLContainer<>(
    "postgres:16")
  .withDatabaseName("testdb")
  .withUsername("test")
  .withPassword("test");
```

7 Dynamiczne właściwości

- Testcontainers dostarcza porty, adresy i dane po starcie kontenera.
- Użyj DynamicPropertySource, aby przekazać te wartości do konfiguracji Spring Boota.
- Dzięki temu aplikacja w testach automatycznie korzysta z uruchomionej usługi.

Przykład: DynamicPropertySource — Java

```java
@DynamicPropertySource
static void registerProperties(
  DynamicPropertyRegistry registry) {
  registry.add("spring.datasource.url",
    postgres::getJdbcUrl);
  registry.add("spring.datasource.username",
    postgres::getUsername);
  registry.add("spring.datasource.password",
    postgres::getPassword);
}
```

8 Integracja z JUnit 5

- Testcontainers świetnie integruje się z JUnit 5 (JUnit Jupiter).
- Wystarczy użyć adnotacji @Test, @Container i innych z biblioteki.
- Możesz korzystać z rozszerzeń, np. @Testcontainers (automatyczne zarządzanie cyklem).

Przykład klasy testowej — Java

```java
@Testcontainers
@SpringBootTest
class UserRepositoryTest {
  static @Container
  static PostgreSQLContainer<?> postgres =
    new PostgreSQLContainer<>("postgres:16");
  @Test
  void shouldSaveUser() { /* ... */ }
}
```

9 Zalety i wady

Zalety

- Testy z prawdziwymi usługami.
- Mniej błędów w środowisku produkcyjnym.
- Prosta i czytelna konfiguracja.
- Wsparcie dla wielu technologii.
- Izolacja i powtarzalność.

Wady / kompromisy

- Wymaga zainstalowanego Dockera.
- Dłuższy czas uruchamiania testów.
- Większe zużycie zasobów (CPU, pamięć).
- Pobieranie obrazów przy pierwszym uruchomieniu.
- Nie zawsze odpowiednie dla bardzo szybkich testów jednostkowych.

10 Najczęstsze pułapki

- Brak zainstalowanego Dockera (w CI lub lokalnie).
- Używanie kontenerów następczo (statycznych), gdy testy powinny być izolowane.
- Zbyt wolne testy przez nieoptymalną konfigurację obrazów.
- Nieczyszczenie danych między testami.
- Używanie niestabilnych tagów obrazów (np. latest).
- Problemy sieciowe przy pobieraniu obrazów w CI.

11 Dobre praktyki

- Używaj konkretnych wersji obrazów (np. postgres:16).
- Współdziel kontenery tam, gdzie to możliwe (pola static).
- Wykorzystuj DynamicPropertySource.
- Czyść dane między testami (lub używaj osobnych baz).
- Ogranicz liczbę uruchamianych kontenerów.
- Stosuj profile testowe (np. application-test.yml).
- W CI korzystaj z cache obrazów.
- Dokumentuj używane obrazy i konfigurację.

Prawdziwe usługi to prawdziwa jakość – testuj tak, jak działa Twoja aplikacja w świecie rzeczywistym! ☺



12 Szybkie podsumowanie

- Testcontainers = prawdziwe usługi w testach.
- Popularne usługi: PostgreSQL, MySQL, Redis, Kafka i wiele innych.
- Łatwa integracja z JUnit 5 i Spring Bootem.
- DynamicPropertySource pozwala na dynamiczną konfigurację.
- Zapewnia realistyczne, izolowane i powtarzalne testy.
- Wymaga Dockera i może być wolniejszy niż testy jednostkowe.

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

MAPA 59/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

@SERVICECONNECTION I KONFIGURACJA TESTCONTAINERS

Jak Spring Boot upraszcza podłączanie kontenerów testowych

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Co robi @ServiceConnection?

- **Automatycznie konfiguruje połączenie** do kontenerów testowych w testach Spring Boota.
- **Sam wykrywa uruchomione** kontenery i podłącza je do kontekstu aplikacji (DataSource, Redis, i inne).
- **Działa „od razu"** – bez ręcznego dodawania właściwości w application-test.properties.

2 Zamiast ręcznego konfigurowania

Przed (ręczne ustawianie)

- Ręczne pobieranie portów i ustawianie właściwości
- Dużo szablonowego kodu
- Łatwo o błędy i literówki
- Trudniejsze utrzymanie testów

```
@DynamicPropertySource
static void registerProps(...) {
  registry.add("spring.datasource.url",
    () -> postgres.getJdbcUrl());
  registry.add("spring.datasource.username",
    () -> postgres.getUsername());
  registry.add("spring.datasource.password",
    () -> postgres.getPassword());
}
```

Po (z @ServiceConnection)

- Brak ręcznych właściwości
- Automatyczna konfiguracja
- Krótszy i czytelniejszy kod
- Mniej potencjalnych błędów
- Lepsza integracja ze Spring Bootem

```
@ServiceConnection
static PostgreSQLContainer<?>
postgres = new PostgreSQLContainer<>(
        "postgres:16-alpine");
```

3 Jak to działa?

- Spring Boot wykrywa pola oznaczone **@ServiceConnection** w testach.
- Na podstawie typu kontenera konfiguruje odpowiednie właściwości aplikacji.
- Korzysta z wbudowanych "service connection factories" (mapujących kontenery na auto-konfigurację).
- Działa z Testcontainers i integruje się z kontekstem Spring.

4 Obsługiwane połączenia

Spring Boot wspiera wiele usług, m.in.:

- PostgreSQL (DataSource)
- MySQL / MariaDB (DataSource)
- Redis (RedisConnectionFactory)
- MongoDB (MongoClient)
- Kafka (KafkaTemplate)
- RabbitMQ (ConnectionFactory)
- Elasticsearch (RestClient)
- i inne... (lista stale rośnie)

Wspierane są najpopularniejsze usługi używane w aplikacjach Spring.

5 Przykład: PostgreSQL i Redis

PostgreSQL

```java
@SpringBootTest
class UserRepositoryTest {

  @ServiceConnection
  static PostgreSQLContainer<?>
    postgres = new PostgreSQLContainer<>(
        "postgres:16-alpine");

  @Autowired
  private UserRepository repo;

  @Test
  void shouldSaveUser() {
    repo.save(new User("Jan"));
    assertThat(repo.count()).isEqualTo(1);
  }
}
```

Redis

```java
@SpringBootTest
class OrderCacheTest {

  @ServiceConnection
  static GenericContainer<?> redis =
      new GenericContainer<>(
          "redis:7-alpine")
          .withExposedPorts(6379);

  @Autowired
  private StringRedisTemplate redisTemplate;

  @Test
  void shouldStoreInCache() {
    redisTemplate.opsForValue()
      .set("key", "value");
    assertThat(redisTemplate
      .opsForValue().get("key"))
      .isEqualTo("value");
  }
```

6 Kiedy użyć DynamicPropertySource?

- Gdy używasz niestandardowego kontenera (nieobsługiwanego przez @ServiceConnection).
- Gdy chcesz skonfigurować własne niestandardowe właściwości.
- Gdy potrzebujesz zaawansowanej logiki (np. dynamicznych portów, wielu kontenerów).

```java
@DynamicPropertySource
static void registerProps(
    DynamicPropertyRegistry registry) {
  registry.add("app.custom.url",
      () -> customContainer.getEndpoint());
}
```

7 Zalety

- Bardzo prosta konfiguracja
- Mniej kodu i mniej błędów
- Automatyczna integracja ze Spring Bootem
- Wsparcie dla wielu popularnych usług
- Czytelne i utrzymywalne testy
- Lepsze doświadczenie dewelopera

8 Ograniczenia

- Działa tylko z kontenerami wspieranymi przez Spring Boot.
- Wymaga użycia Testcontainers.
- Ograniczone możliwości niestandardowej konfiguracji (wtedy użyj DynamicPropertySource).
- Wymaga nowszej wersji Spring Boota (3.1+ — pierwsza wersja z @ServiceConnection).
- Nie zastąpi wszystkich scenariuszy niestandardowych.

9 Wskazówki praktyczne

- Używaj najnowszych wersji Spring Boot i Testcontainers.
- Trzymaj kontenery jako statyczne pola (static).
- Stosuj odpowiednie obrazy (np. alpine).
- Możesz dodać `.withReuse(true)` przy częstym uruchamianiu.
- Łącz z @Testcontainers (opcjonalnie).
- Przy wielu kontenerach stosuj opisowe nazwy pól.
- Dbaj o czysty kontekst między testami (@DirtiesContext jeśli potrzeba).

10 Migracja ze starszej konfiguracji

- Usuń ręczne ustawianie właściwości w testach.
- Zastąp DynamicPropertySource adnotacją @ServiceConnection.
- Uprość konfigurację kontenerów.
- Sprawdź czy dany kontener jest wspierany.
- Uruchom testy i usuń zbędny kod.
- Ciesz się prostszymi testami! 😀

11 Szybkie podsumowanie

- @ServiceConnection automatycznie konfiguruje połączenie do kontenerów Testcontainers.
- Zastępuje ręczne ustawianie właściwości.
- Obsługuje wiele popularnych usług (np. PostgreSQL, Redis, MongoDB).
- W trudniejszych przypadkach użyj DynamicPropertySource.
- To prostsze, czystsze i bardziej niezawodne testy w Spring Bootcie.

12 W praktyce

- Użyj @ServiceConnection.
- Uruchom kontener Testcontainers.
- Pozwól Spring Bootowi na konfigurację.
- Pisz testy jak zwykle.
- Skup się na logice aplikacji, nie na konfiguracji infrastruktury!
- Szybciej do działającej aplikacji! 🚀



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 06 • TESTOWANIE APLIKACJI

SPRING REST DOCS

Jak generować wiarygodną dokumentację API na podstawie testów

MAPA 60/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsza dokumentacja
Większa pewność!

1 Co to jest Spring REST Docs?

Spring REST Docs to biblioteka do automatycznego generowania dokumentacji API na podstawie testów.

Dokumentacja zawsze odzwierciedla rzeczywiste działanie aplikacji (bo powstaje z testów).

Tworzy tzw. snippet'y, które następnie przetwarzane są np. przez Asciidoctor do czytelnej dokumentacji (HTML, PDF).

2 Dlaczego generować dokumentację z testów?

Zawsze aktualna – dokumentacja powstaje na podstawie działających testów.

Wiarygodna – opisuje rzeczywiste zachowanie API, nie teoretyczne założenia.

Mniej pracy – nie musisz pisać i utrzymywać osobnych opisów.

Lepsza komunikacja – ułatwia współpracę zespołów (backend, frontend, QA).

Wyższa jakość API – testy wymuszają jasny kontrakt i przemyślane odpowiedzi.

3 Relacja z MockMvc / WebTestClient / REST Assured

MockMvc – Najczęściej używany w aplikacjach serwletowych (Spring MVC). Stosowany z *spring-restdocs-mockmvc*.

WebTestClient – Używany w aplikacjach reaktywnych (Spring WebFlux). Stosowany z spring-restdocs-webtestclient.

REST Assured – Może być używany z REST Assured do generowania snippetów. Stosowany z spring-restdocs-restassured.

Spring REST Docs integruje się z popularnymi narzędziami testowymi – wybierz to, którego już używasz w projekcie.

4 Czym są snippet'y?

Snippet'y to małe fragmenty plików generowane podczas testów opisujące różne elementy API (requesty, odpowiedzi itp.).

- Powstają w katalogu target/snippets
- Zawierają dane w formacie Asciidoctor (np. .adoc)
- Można je łatwo włączyć do dokumentacji
- Są generowane tylko dla tego, co przetestowano

5 Dokumentowanie żądania (request)

Pola żądania (body)
Opis pól w ciele żądania (JSON/XML)
`np. fieldWithPath("name").description(...)`

Parametry ścieżki (path params)
Opis parametrów w ścieżce URL
`np. parameterWithName("id").description(...)`

Parametry zapytania (query params)
Opis parametrów ?page=0&size=10
`np. parameterWithName("page").description(...)`

Nagłówki (headers)
Opis wymaganych nagłówków
`np. headerWithName("Authorization").description(...)`

6 Dokumentowanie odpowiedzi (response)

Pola odpowiedzi (body)
Opis struktury odpowiedzi
`np. fieldWithPath("id").description(...)`

Kody statusu
Dokumentowanie zwracanych kodów HTTP
`np. responseFields(...)`

Nagłówki odpowiedzi
Opis nagłówków w odpowiedzi
`np. headerWithName("Location").description(...)`

Dokumentuj tylko to, co jest istotne dla klienta API. Skup się na polach, które są częścią kontraktu.

7 Pełny przykład testu z REST Docs

Java

```
@Test
void powinienZwrócicUżytkownika() throws Exception {
  mockMvc.perform(get("/api/users/{id}", 1L)
        .accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.id").value(1))
    .andDo(document("user-get",
      pathParameters(
        parameterWithName("id").description("Identyfikator użytkownika")
      ),
      responseFields(
        fieldWithPath("id").description("Id użytkownika"),
        fieldWithPath("name").description("Nazwa użytkownika"),
        fieldWithPath("email").description("Adres e-mail")
      ));
}
```

8 Workflow: od testu do dokumentacji

1. Testy z REST Docs
Uruchom testy (MockMvc, WebTestClient lub REST Assured) z .andDo(document()).

↓

2. Generowanie snippetów
Snippet'y trafiają do katalogu target/snippets.

↓

3. Asciidoctor
Włącz snippet'y w plikach .adoc i zbuduj dokumentację.

↓

4. Gotowa dokumentacja
Wygenerowany HTML (np. target/generated-docs) gotowy do publikacji (np. na GitHub Pages).

9 Praktyczna konfiguracja

- Dodaj zależności w build.gradle / pom.xml:

```
testImplementation 'org.springframework.restdocs
    :spring-restdocs-mockmvc'
```

- Skonfiguruj MockMvc / WebTestClient z RestDocumentation.
- Użyj wtyczki Asciidoctor do generowania HTML.
- Przechowuj pliki .adoc w katalogu src/docs/asciidoc.
- Dodaj przykład pliku build.gradle:

```
tasks.named('test') {
    useJUnitPlatform()
}
asciidoctor {
    inputs.dir snippetsDir
}
```

10 Korzyści i ograniczenia

Korzyści

- Aktualna i wiarygodna dokumentacja
- Mniej duplikacji pracy
- Lepsza jakość i czytelność API
- Integracja z istniejącymi testami
- Wygodna publikacja (HTML, PDF)
- Wspiera różne narzędzia testowe

Ograniczenia

- Wymaga napisania testów
- Większy próg wejścia na początku
- Nie generuje diagramów (np. OpenAPI)
- Skupia się na opisie, nie na interakcji
- Wymaga konfiguracji Asciidoctor
- Nie zastępuje narzędzi typu Swagger

11 Najczęstsze pułapki

- Brak uruchamiania testów przed generacją.
- Nieudokumentowane pola odpowiedzi.
- Zbyt ogólne opisy (np. „pole" zamiast „Nazwa użytkownika").
- Pominięte kody błędów.
- Nieaktualne pliki .adoc.
- Problemy z konfiguracją ścieżek do snippetów.
- Traktowanie dokumentacji jako osobnego zadania (zamiast efektu testów).

12 Szybkie podsumowanie

- Spring REST Docs generuje dokumentację z testów.
- Działa z MockMvc, WebTestClient i REST Assured.
- Tworzy snippet'y opisujące request i response.
- Wykorzystaj Asciidoctor do budowy czytelnego HTML.
- Opisuj pola, parametry, nagłówki i kody statusu.
- Dokumentacja jest aktualna, wiarygodna i łatwa w utrzymaniu.

Dobra dokumentacja to zadowoleni użytkownicy.



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 — INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 61/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

RestClient

Synchroniczny klient HTTP w Spring Framework i Spring Boot

Lepsze integracje
Lepsze aplikacje
Większa pewność!

1 Czym jest RestClient?

RestClient to nowoczesny, **synchroniczny** klient **HTTP** dodany w **Spring Framework** 6.1 i dostępny w **Spring Boot 3.2+**.

Umożliwia wygodne wywoływanie zewnętrznych API (REST) przy użyciu fluent API.

Jest następcą **RestTemplate** i rekomendowany do nowych projektów.

2 Dlaczego powstał RestClient?

- Bardziej intuicyjne i czytelne fluent **API**.
- Prostszy i nowocześniejszy niż RestTemplate.
- Lepsza integracja z ekosystemem Spring (konfiguracją, observability, interceptory).
- Zachowuje model synchroniczny – idealny dla wielu zastosowań biznesowych.
- **Rekomendowany sposób** wywoływania API w nowych projektach.

3 Jak to działa? – przepływ żądania i odpowiedzi

1. Wybierz metodę HTTP (GET, POST, PUT, DELETE, ...), podaj URI, parametry, nagłówki i ewentualnie ciało żądania.
2. Wyślij żądanie za pomocą RestClient.
3. Odbierz odpowiedź (retrieve()).
4. Zmapuj ciało odpowiedzi do obiektu (np. klasę DTO, String lub **ResponseEntity**).
5. Obsłuż status odpowiedzi i ewentualne błędy (np. za pomocą **onStatus()**).

4 Tworzenie RestClienta (bean)

Java

```java
@Configuration
public class RestClientConfig {

  @Bean
  public RestClient restClient(
     @Value("${api.base-url}") String baseUrl) {
    return RestClient.bulder()
       .baseUrl(baseUrl)
       .defaultHeader(HttpHeaders.ACCEPT,
                MediaType.APPLICATION_JSON_VALUE)
       .defaultHeader(HttpHeaders.CONTENT_TYPE,
                MediaType.APPLICATION_JSON_VALUE)
       .build();
}
```

5 Przykład: żądanie GET

Java

```java
@Service
public class UserService {

  private final RestClient restClient;

  public UserService(RestClient restClient) {
    this.restClient = restClient;
  }
  public UserDto getUser(Long id) {
    return restClient.get()
       .uri("/users/{id}", id)
       .retrieve()
       .body(UserDto.class);
}
```

6 Przykład: żądanie POST

Java

```java
@Service
public class UserService {

  private final RestClient restClient;

  public UserService(RestClient restClient) {
    this.restClient = restClient;
  }
  public UserDto createUser(CreateUserRequest request) {
    return restClient.post()
       .uri("/users")
       .contentType(MediaType.APPLICATION_JSON)
       .body(request)
       .retrieve()
}      .body(UserDto.class);
```

7 URI, parametry, nagłówki i ciało

Element	Przykład użycia
Bazowy URI	`.baseUrl("https://api.example.com")`
Parametry ścieżki	`.uri("/users/{id}", id)`
Parametry zapytania	`.uri(uriBuilder -> uriBuilder .path("/users") .queryParam("active", true) .build())`
Nagłówki	`.header("X-API-KEY", "key123") defaultHeader(HttpHeaders.ACCEPT, MediaType.APPLICATION_JSON_VALUE)`
Ciało żądania	`.body(obiekt)` (automatycznie konwertowane do JSON przez HttpMessageConverter)

8 retrieve() i konwersja odpowiedzi

- retrieve() wysyła żądanie i zwraca specyfikację obsługi odpowiedzi.
- Odpowiedź jest automatycznie konwertowana przez HttpMessageConverters (np. z JSON do obiektu).
- Możesz mapować odpowiedź do klasy DTO, String lub ResponseEntity.

Java

```java
UserDto user = restClient.get()
      .uri("/users/{id}", id)
      .retrieve()
      .body(UserDto.class);
String json = restClient.get()
      .uri("/status")
      .retrieve() #- .body(String.class);
```

9 Obsługa błędów

- Użyj onStatus() do obsługi wybranych kodów statusu (4xx, 5xx).
- Możesz rzucać własne wyjątki z treścią błędu.
- Domyślnie błędne statusy zgłaszają wyjątki (np. RestClientResponseException).
- Warto skonfigurować timeouts, retries oraz logowanie (np. poprzez interceptory).

Java

```java
restClient.get()
    .uri("/users/{id}", id)
    .retrieve()
    .onStatus(HttpStatus.NOT_FOUND::equals,
        (request, response) ->
        Mono.error(new UserNotFoundException(id)))
    .body(UserDto.class);
```

10 RestClient vs RestTemplate vs WebClient

Cecha	RestTemplate	RestClient	WebClient
Wprowadzony	Spring 3.0	Spring 6.1 (Boot 3.2+)	Spring 5.0
Status	W trybie utrzymania	Rekomendowany	Aktywnie rozwijany
Model	Synchroniczny	Synchroniczny	Reaktywny (nieblokujący)
Styl API	Klasyczny	Fluent API	Fluent API (Reaktywny)
Zastosowanie	Starsze projekty	Nowe projekty (zalecane)	Aplikacje reaktywne

11 Dobre praktyki

- Definiuj **RestClient** jako bean i centralizuj konfigurację (base URL, nagłówki, interceptory).
- Używaj DTO do mapowania danych.
- Waliduj statusy odpowiedzi i obsługuj błędy.
- Konfiguruj **timeouts** i ewentualne retry.
- Dodaj logowanie i metryki (np. Micrometer).
- Wykorzystuj interceptory do dodawania nagłówków (np. autoryzacja).
- Unikaj duplikacji kodu – twórz wspólne komponenty.

12 Szybkie podsumowanie

- RestClient to nowoczesny, synchroniczny klient HTTP.
- Używa prostego i czytelnego **fluent** API.
- Świetnie integruje się ze Spring **Bootem**.
- Umożliwia łatwe mapowanie odpowiedzi do obiektów.
- Obsługuje błędy przez **onStatus()** i wyjątki.
- To rekomendowany sposób wywoływania REST API w nowych projektach.

Integracje otwierają Twoją aplikację na świat! ☺

- Pisz czytelny i utrzymywalny kod.
- Testuj integracje z zewnętrznymi API.
- Dbaj o bezpieczeństwo i obsługę błędów.



Dobre integracje to nowe możliwości! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 – INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 62/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

WebClient

Klient HTTP i model reaktywny

Lepsze testy
Lepsza dokumentacja
Większa pewność!

1 Czym jest WebClient?

WebClient to nowoczesny, **reaktywny** klient HTTP w Springu.

Należy do modułu **Spring WebFlux** i działa w modelu nieblokującym (opartym o reactive streams).

Obsługuje protokoły HTTP/HTTPS, oferuje elastyczny, funkcyjny API i dobrze integruje się z aplikacjami reaktywnymi.

2 Podstawa reaktywna i WebFlux

WebClient opiera się na **Project Reactor** (Mono i Flux) i implementuje nieblokujący model I/O.

Jest częścią **Spring WebFlux**, ale może być używany również w aplikacjach opartych o Spring MVC.

Wykorzystuje reaktory **Netty** jako domyślny silnik HTTP (brak blokujących wątków).

WebClient idealnie pasuje do architektury reaktywnej, ale działa także jako klient w aplikacjach tradycyjnych.

3 Mono i Flux – w skrócie

Mono<T>
- reprezentuje 0 lub 1 element
- używany dla pojedynczego wyniku
- np. odpowiedź JSON (obiekt)

Flux<T>
- reprezentuje 0..N elementów
- strumień wielu danych
- np. lista obiektów, strumień SSE

Mono i Flux są „leniwe" – wykonanie następuje dopiero po subskrypcji (np. poprzez .subscribe() lub zwrot z metody kontrolera).

4 Podstawowe użycie – GET

- Wysyłanie żądań GET.
- Pobieranie danych do Mono lub Flux.
- Obsługa parametrów URI.

Przykład: Java

```
Mono<User> user = webClient
  .get()
  .uri("https://api.example.com/users/{id}", 123)
  .retrieve()
  .bodyToMono(User.class);
```

5 Wysyłanie danych – POST

- Wysyłanie danych w body (np. JSON).
- Automatyczna serializacja obiektów.
- Zwracanie odpowiedzi jako Mono/Flux.

Przykład: Java

```
Mono<User> created = webClient
  .post()
  .uri("https://api.example.com/users")
  .contentType(MediaType.APPLICATION_JSON)
  .bodyValue(new CreateUserRequest("Jan"))
  .retrieve()
  .bodyToMono(User.class);
```

6 Body, nagłówki i budowanie URI

- Łatwe dodawanie nagłówków.
- Różne sposoby przekazania body.
- Wygodne budowanie dynamicznych URI.

Przykład: Java

```
URI uri = UriComponentsBuilder
  .fromUriString("https://api.example.com")
  .path("/search")
  .queryParam("q", "spring boot")
  .queryParam("page", 1)
  .build()
  .toUri();

webClient.get()
  .uri(uri)
  .header("Authorization", "Bearer token")
  .retrieve()
  .bodyToFlux(Item.class);
```

7 retrieve() czy exchangeTo...?

- **retrieve()** – proste API, automatyczna obsługa błędów HTTP (status 4xx/5xx).
- **exchangeToMono()/exchangeToFlux()** – pełna kontrola nad odpowiedzią (ClientResponse).

Przykład (exchangeToMono): Java

```
Mono<String> result = webClient.get()
  .uri("/resource")
  .exchangeToMono(response -> {
  if (response.statusCode().is2xxSuccessful()) {
    return response.bodyToMono(String.class);
  } else {
    return response.bodyToMono(ErrorDto.class)
      .flatMap(err -> Mono.error(
        new RuntimeException(err.message())));
  }
  }));
```

8 Obsługa błędów

- Obsługa błędów HTTP i wyjątków sieciowych.
- Metoda onStatus() do mapowania błędów.
- Możliwość ponawiania (retry) i fallbacków.

Przykład: Java

```
webClient.get()
  .uri("/users/{id}", 1)
  .retrieve()
  .onStatus(HttpStatus::is4xxClientError,
     resp -> resp.bodyToMono(ErrorDto.class)
       .map(err -> new BadRequestException(err.message())))
  .bodyToMono(User.class)
  .retry(3)
  .onErrorResume(ex -> Mono.empty());
```

9 Blokujący czy nieblokujący?

Nieblokujący (zalecany)
- Zwracaj Mono lub Flux
- Nie używaj .block() w aplikacji reaktywnej
- Lepsza wydajność i skalowalność
- Wykorzystanie niewielkiej liczby wątków

Blokujący (tylko w specyficznych przypadkach, np. aplikacje MVC)
- .block() zatrzymuje wątek
- Może prowadzić do problemów z wydajnością
- Używaj świadomie, np. w batchach lub aplikacjach nie reaktywnych

10 Kiedy wybrać WebClient?

- Potrzebujesz nieblokującego, reaktywnego klienta.
- Pracujesz z dużą liczbą równoległych wywołań.
- Chcesz korzystać ze strumieniowania danych (np. SSE, duże zbiory danych).
- Masz aplikację opartą o WebFlux.
- Potrzebujesz większej elastyczności niż w RestClient.

WebClient zamiast RestClient, gdy:
- ważna jest skalowalność i wydajność,
- potrzebujesz strumieniowania (SSE, Flux),
- chcesz uniknąć blokujących operacji,
- budujesz aplikację reaktywną lub hybrydową.

11 Strumieniowanie i SSE

- WebClient obsługuje Server-Sent Events (SSE).
- Możesz odbierać strumień danych jako Flux.
- Przydatne np. w powiadomieniach, postępie zadań, logach na żywo.

Przykład SSE: Java

```
Flux<ServerSentEvent<String>> events = webClient.get()
  .uri("/events")
  .accept(MediaType.TEXT_EVENT_STREAM)
  .retrieve()
  .bodyToFlux(new ParameterizedTypeReference<
     ServerSentEvent<String>>() {});
```

12 Dobre praktyki

- Utwórz i konfiguruj WebClient jako bean (z użyciem WebClient.builder()).
- Stosuj timeouty i strategię retry.
- Używaj filtrów (logowanie, autoryzacja, metryki).
- Zwracaj Mono/Flux zamiast blokować.
- Poprawnie obsługuj błędy (onStatus, fallback).
- Stosuj mapowanie DTO zamiast zwracania surowych map.
- Pisz testy z użyciem MockWebServer lub WireMock.

13 Najczęstsze pułapki

- Używanie .block() w aplikacji reaktywnej.
- Brak obsługi błędów HTTP.
- Nieuwzględnienie timeoutów (np. connectionTimeout).
- Niepoprawne zarządzanie zasobami (tworzenie WebClient w każdej metodzie).
- Zwracanie typów blokujących (np. List zamiast Flux).
- Ignorowanie kodów statusu odpowiedzi.
- Brak testów integracyjnych (MockWebServer/WireMock).
- Nadmiarowe mapowanie obiektów (brak DTO).

Nieblokujące aplikacje to bardziej skalowalne rozwiązania!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 63/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

APPLICATION EVENTS i @EVENTLISTENER

Luźne powiązania i komunikacja wewnątrz aplikacji

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym są Application Events?

Application Events to mechanizm zdarzeń w Springu, który pozwala na luźne powiązania między komponentami aplikacji.

Umożliwiają komunikację typu „jeden do wielu" bez bezpośrednich zależności między nadawcą a odbiorcami.

Sprawdzają się w sytuacjach, gdy chcemy reagować na istotne zdarzenia w aplikacji, np. utworzenie użytkownika, zmiana stanu zamówienia itp.

2 Role: publisher i listener

Publisher (nadawca)
- publikuje zdarzenie w odpowiednim momencie
- nie wie, kto je obsługuje

Listener (odbiorca)
- nasłuchuje wybranych zdarzeń
- wykonuje określoną logikę
- może być wielu listenerów na to samo zdarzenie
- działają niezależnie od nadawcy

Dzięki temu komponenty są luźno powiązane, co ułatwia rozwój, testowanie i utrzymanie aplikacji.

3 Wbudowane zdarzenia Springa

ContextRefreshedEvent
– kontekst aplikacji został odświeżony.

ContextStartedEvent
– kontekst aplikacji został uruchomiony.

ContextStoppedEvent
– kontekst aplikacji został zatrzymany.

ContextClosedEvent
– kontekst aplikacji został zamknięty.

RequestHandledEvent
– obsłużono żądanie HTTP (w aplikacjach webowych).

Możesz nasłuchiwać na te zdarzenia tak samo, jak na własne zdarzenia aplikacyjne.

4 Własne zdarzenia

- Własne zdarzenia reprezentują istotne sytuacje w Twojej domenie biznesowej, np. UserRegisteredEvent czy OrderCreatedEvent.

- Najczęściej dziedziczą po klasie **ApplicationEvent** (można też użyć rekordu lub zwykłej klasy).
- Zdarzenie może zawierać dane, które są potrzebne listenerom.

5 ApplicationEventPublisher

ApplicationEventPublisher to interfejs pozwalający na publikowanie zdarzeń w aplikacji.

- Wstrzyknij go do swojego komponentu.
- Użyj metody **publishEvent(...)**, aby opublikować zdarzenie.

Spring automatycznie dostarcza implementację tego interfejsu (jako bean).

6 @EventListener

@EventListener oznacza metodę, która ma być wywołana po opublikowaniu określonego zdarzenia.

- Możesz nasłuchiwać na konkretne typy zdarzeń (np. UserRegisteredEvent).
- Metoda może przyjmować zdarzenie jako parametr.
- Obsługujesz zdarzenia tylko, które Cię interesują.

7 Przykład: publikowanie zdarzenia

Java

```
@Service
public class UserService {

    private final ApplicationEventPublisher
        eventPublisher;

    public UserService(ApplicationEventPublisher
        eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public void registerUser(String username) {
        // logika rejestracji użytkownika
        UserRegisteredEvent event =
            new UserRegisteredEvent(this, username);
        eventPublisher.publishEvent(event);
    }
}
```

8 Przykład: nasłuchiwanie zdarzenia

Java

```
@Component
public class UserEventListener {

    @EventListener
    public void handleUserRegistered(
        UserRegisteredEvent event) {
        System.out.println("Nowy użytkownik: "
            + event.getUsername());
        // dodatkowa logika, np. wysłanie e-maila
    }
}
```

```
public class UserRegisteredEvent
            extends ApplicationEvent {
    private final String username;
    public UserRegisteredEvent(Object source,
                String username) {
        super(source);
        this.username = username;
    }
    public String getUsername() {
        return username;
    }
}
```

9 Zachowanie synchroniczne (domyślnie)

- Domyślnie listenery są wywoływane **synchronicznie** – w tym samym wątku, który publikuje zdarzenie.
- Każdy listener wykonywany jest kolejno.
- Długi czas przetwarzania listenera może spowolnić działanie całej aplikacji.

Jeśli obsługa zdarzenia jest czasochłonna, rozważ zastosowanie przetwarzania asynchronicznego.

10 Asynchroniczne listenery z @Async

- Użyj adnotacji **@Async** na metodzie listenera.
- Wymaga włączenia obsługi asynchroniczności przez **@EnableAsync**.
- Listener będzie wykonywany w osobnym wątku.
- Przydatne dla długich operacji, np. wysyłki e-maili.

11 Zdarzenia powiązane z transakcjami

- **@TransactionalEventListener** pozwala nasłuchiwać zdarzeń w konkretnych fazach transakcji.
- Dostępne fazy: BEFORE_COMMIT, AFTER_COMMIT (domyślna), AFTER_ROLLBACK, AFTER_COMPLETION.
- Przydatne, gdy reakcja na zdarzenie ma być wykonana dopiero po udanym zatwierdzeniu transakcji.

12 Przykład: @TransactionalEventListener

Java

```
@Component
public class OrderEventListener {

    @TransactionalEventListener(
        phase = TransactionPhase.AFTER_COMMIT)
    public void onOrderCreated(OrderCreatedEvent event) {
        // wyślij potwierdzenie zamówienia
        emailService.sendOrderConfirmation(
            event.getOrderId());
    }
}
```

13 Praktyczne zastosowania

- Rejestracja i obsługa nowych użytkowników
- Obsługa utworzonych zamówień
- Wysyłka e-maili i powiadomień
- Czyszczenie cache po zmianach danych
- Integracja z innymi modułami aplikacji
- Publikacja zdarzeń domenowych (DDD)
- Audyt i logowanie istotnych akcji

14 Korzyści

- Luźne powiązania komponentów
- Łatwiejsze testowanie
- Skalowalność i elastyczność
- Obsługa wielu listenerów
- Integracja z transakcjami
- Spójny, prosty w użyciu model programowania

15 Pułapki i na co uważać

- Długie operacje w synchronii blokują aplikację.
- Zbyt wiele listenerów może utrudniać śledzenie przepływu zdarzeń.
- Brak obsługi wyjątków w listenerach (warto logować błędy).
- Nieprzewidywalna kolejność wykonania wielu listenerów.
- Nie używaj eventów do krytycznej logiki biznesowej, gdzie potrzebna jest ścisła kontrola przepływu.

16 Szybkie podsumowanie

- ApplicationEvents umożliwiają luźną komunikację w aplikacji.
- Używaj ApplicationEventPublisher do publikowania zdarzeń.
- Nasłuchuj zdarzeń za pomocą @EventListener.
- Domyślnie listenery działają synchronicznie.
- Użyj **@Async** dla asynchronicznego przetwarzania.
- @TransactionalEventListener pozwala reagować w fazach transakcji.
- To prosty i potężny mechanizm do budowania czystej, modułowej aplikacji.

Luźne powiązania to bardziej elastyczna i przyszłościowa aplikacja!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 64/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

@Async, Executor i pule wątków

Asynchroniczne wykonywanie zadań w Springu

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Co robi @Async?

- Wykonuje metodę w osobnym wątku, dzięki czemu nie blokuje wątku wywołującego.
- Zwraca wynik natychmiast (w przypadku void) lub obiekt reprezentujący wynik (Future / CompletableFuture).
- Umożliwia obsługę długotrwałych operacji bez blokowania użytkownika/klienta aplikacji.
- Idealny do zadań I/O, wysyłania e-maili, integracji z systemami zewnętrznymi czy przetwarzania danych w tle.

2 Włączanie asynchroniczności

- Dodaj adnotację **@EnableAsync** w klasie konfiguracyjnej.
- Oznacz metody adnotacją **@Async** (w razie potrzeby wskaż nazwę executora).

Przykład: Java

```java
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean(name = "taskExecutor")
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor =
            new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("async-");
        executor.initialize();
        return executor;
}
```

3 Jak działają proxy?

- **@Async** jest obsługiwane przez proxy Springa (AOP).
- Wywołanie metody z zewnątrz (przez proxy) trafia do interceptora, który deleguje wykonanie do executora.
- Wywołanie metody wewnątrz tej samej klasy (self-invocation) omija proxy – **@Async** nie zadziała!
- Tylko metody **publiczne** są domyślnie obsługiwane (przy użyciu proxy JDK). Dla metod niepublicznych wymagany jest proxy CGLIB.

Aby wywołać metodę @Async z tej samej klasy, wydziel ją do osobnego beana.

4 Typy zwracane

- **void** – metoda działa asynchronicznie, brak wyniku.
- **Future<T>** – starsze API, umożliwia sprawdzenie wyniku i statusu.
- **CompletableFuture<T>** – nowoczesne API z metodami do łączenia, kompozycji i obsługi błędów.

Przykład: Java

```java
@Async
public CompletableFuture<String>
   pobierzDane() {
    return CompletableFuture
           .supplyAsync(() -> "wynik");
}
```

5 TaskExecutor i pule wątków

- Najczęściej używany: **ThreadPoolTaskExecutor** (oparty o ThreadPoolExecutor).
- Spring udostępnia też **SimpleAsyncTaskExecutor** (bez puli – tworzy nowy wątek dla każdego zadania).
- Można zdefiniować własny executor jako bean i wskazać go w @Async("nazwaBean").

Przykład użycia w serwisie: Java

```java
@Service
public class RaportService {

    @Async("taskExecutor")
    public void generujRaport() {
        // długotrwałe przetwarzanie...
    }
}
```

6 Najważniejsze parametry puli

- **corePoolSize** – liczba wątków utrzymywanych zawsze (nawet gdy brak zadań).
- **maxPoolSize** – maksymalna liczba wątków (paralelne przetwarzanie).
- **queueCapacity** – rozmiar kolejki zadań, które czekają na wykonanie.
- **keepAliveSeconds** – czas życia nieaktywnych wątków (powyżej corePoolSize).
- **threadNamePrefix** – prefiks nazw wątków (ułatwia logowanie i diagnostykę).

Dobrze dobrane parametry puli zapobiegają przeciążeniu systemu i problemom z pamięcią.

7 Obsługa wyjątków

- W metodach zwracających void wyjątki nie są przekazywane do wywołującego.
- Użyj **AsyncUncaughtExceptionHandler** do globalnej obsługi wyjątków.
- Dla **Future/CompletableFuture** obsłuż wyjątki przez exceptionally(), handle() lub get().

Przykład konfiguracji: Java

```java
@Bean
public AsyncUncaughtExceptionHandler
   asyncExceptionHandler() {
   return (ex, method, params) -> {
      log.error("Błąd w metodzie async: {}",
         method.getName(), ex);
}    };
```

8 Kiedy asynchroniczność pomaga?

- Długie operacje I/O (np. wywołania HTTP, wysyłanie e-maili, zapisywanie plików).
- Przetwarzanie danych w tle (raporty, eksporty).
- Zadania, które nie muszą zwrócić wyniku od razu.
- Poprawa responsywności aplikacji (szybsza odpowiedź dla użytkownika).

9 Typowe zastosowania

- Wysyłanie e-maili i powiadomień.
- Integracja z zewnętrznymi systemami (HTTP, API).
- Generowanie raportów i eksportów plików.
- Przetwarzanie plików (np. CSV, obrazy).
- Operacje na zdarzeniach i logach.
- Wstępne przygotowanie danych (cache, indeksy).

Kiedy NIE pomaga?

- Zadania CPU-bound (intensywne obliczenia) – lepiej użyć innego podejścia (np. batch, kolejka).
- Gdy i tak musisz czekać na wynik.
- Bardzo krótkie zadania – narzut wątku może być większy niż korzyść.
- Gdy wywołujesz metodę asynchroniczną wewnątrz tej samej klasy (self-invocation).

10 Najczęstsze pułapki

- Wywołanie metody @Async z tej samej klasy.
- Zbyt mała lub zbyt duża pula wątków.
- Blokujące operacje w wątkach asynchronicznych.
- Brak obsługi wyjątków.
- Niepoprawne oczekiwanie na wynik (np. get() bez timeoutu).
- Współdzielenie stanu między wątkami (problemy z wątkowością).

11 Najlepsze praktyki

- Używaj dedykowanego TaskExecutora.
- Nadaj czytelne nazwy wątkom (threadNamePrefix).
- Monitoruj wykorzystanie puli wątków (logi, metryki).
- Obsługuj wyjątki (AsyncUncaughtExceptionHandler, exceptionally()).
- Dobieraj parametry puli do charakteru zadań.
- Nie blokuj wątków asynchronicznych (np. długie wywołania z timeoutem).
- Testuj scenariusze obciążeniowe.
- Dokumentuj, które operacje są wykonywane asynchronicznie.

12 Szybkie podsumowanie

- @Async uruchamia metodę w osobnym wątku.
- Włącz asynchroniczność przez **@EnableAsync**.
- Używaj **TaskExecutora** z poprawnie dobraną pulą.
- Obsługuj wyjątki i monitoruj działanie.
- Idealny do zadań I/O i operacji w tle.
- Pamiętaj o ograniczeniach i pułapkach.



Asynchroniczność to więcej czasu dla użytkowników i lepsze aplikacje!

Pisz testy
Buduj lepsze jutro!

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

@Scheduled

Wykonywanie zadań cyklicznych

MAPA 65/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest @Scheduled?

Umożliwia wykonywanie zadań cyklicznych w aplikacji.
Wykonuje metody automatycznie w określonych odstępach czasu.

Przydatny do zadań typu:
czyszczenie danych, raporty, integracje, powiadomienia.

Proste, ale bardzo użyteczne narzędzie w Spring Boot.
Działa w tle, bez potrzeby zewnętrznych harmonogramów.

2 Włączanie planowania zadań

- Dodaj adnotację **@EnableScheduling** w klasie konfiguracyjnej lub głównej klasie aplikacji.
- Metody z adnotacją **@Scheduled** będą wtedy automatycznie uruchamiane przez Springa.
- Możesz umieścić je w dowolnym beanie (np. w serwisie).

Przykład: Java

```java
@SpringBootApplication
@EnableScheduling
public class MojaAplikacja {
    public static void main(String[] args) {
        SpringApplication.run(MojaAplikacja.class, args);
    }
}
```

3 Rodzaje harmonogramów

fixedRate	Wykonuje zadanie w stałych odstępach czasu, licząc od początku poprzedniego wykonania.
fixedDelay	Wykonuje zadanie w stałych odstępach czasu, licząc od końca poprzedniego wykonania.
initialDelay	Opóźnia pierwsze wykonanie zadania o określony czas.
cron	Wykonuje zadanie zgodnie z wyrażeniem CRON (bardziej elastyczny harmonogram).

4 Przykłady użycia

fixedRate – co 5 sekund Java

```java
@Scheduled(fixedRate = 5000)
public void raportujStatystyki() {
    log.info("Generuję statystyki...");
}
```

fixedDelay + initialDelay Java

```java
@Scheduled(fixedDelay = 10000,
           initialDelay = 3000)
public void czyscTymczasowePliki() {
    log.info("Czyszczę pliki tymczasowe...");
}
```

cron – codziennie o 2:00 w nocy Java

```java
@Scheduled(cron = "0 0 2 * * ?")
public void generujRaportDzienny() {
    log.info("Generuję dzienny raport...");
}
```

5 Wyrażenia CRON – podstawy

Element	Zakres	Przykład	Opis
Sekundy	0–59	0	W 0 sekundzie
Minuty	0–59	0	W 0 minucie
Godziny	0–23	2	O 2 w nocy
Dzień miesiąca	1–31	*	Każdego dnia
Miesiąc	1–12	*	Każdego miesiąca
Dzień tygodnia	0–7	1	W poniedziałek (1 = poniedziałek)

Przykład: 0 0 2 * * ?
= codziennie o 2:00 w nocy

6 Strefa czasowa

- Domyślnie używana jest strefa czasowa serwera (JVM).
- Możesz jawnie określić strefę za pomocą atrybutu zone.
- Ważne w aplikacjach działających w chmurze i na różnych serwerach.
- Używaj nazw stref IANA, np. "Europe/Warsaw".

Przykład: Java

```java
@Scheduled(cron = "0 0 2 * * ?",
           zone = "Europe/Warsaw")
public void zadanie() {
    log.info("Uruchamiam o 2:00 czasu PL");
}
```

7 Praktyczne zastosowania

- Generowanie raportów (np. dziennych).
- Czyszczenie starych danych i plików.
- Synchronizacja danych z zewnętrznymi systemami.
- Wysyłka powiadomień (e-mail, SMS).
- Sprawdzanie statusów zamówień.
- Odświeżanie cache.
- Wykonywanie zadań konserwacyjnych.
- Integracje z API (np. pobieranie danych).

8 Unikanie nakładania się zadań

- Jeśli zadanie trwa dłużej niż kolejny harmonogram, może dojść do nakładania się wykonań.
- Domyślnie Spring nie blokuje kolejnych uruchomień.
- Rozważ użycie mechanizmu blokady, np. ShedLock (dla wielu instancji).
- Możesz też samodzielnie kontrolować wykonywanie np. za pomocą AtomicBoolean.

9 Pula wątków (Thread Pool)

- Domyślnie Spring używa pojedynczego wątku (jeden wątek dla wszystkich zadań).
- Dla wielu zadań warto skonfigurować własny ThreadPoolTaskScheduler.
- Pozwala to na równoległe wykonywanie zadań.
- Można ustawić liczebność puli wątków.

Przykład konfiguracji: Java

```java
@Bean
public ThreadPoolTaskScheduler taskScheduler() {
    ThreadPoolTaskScheduler scheduler =
        new ThreadPoolTaskScheduler();
    scheduler.setPoolSize(5);
    scheduler.setThreadNamePrefix("scheduler-");
    return scheduler;
}
```

10 Monitorowanie i logowanie

- Loguj początek i koniec wykonywania zadań.
- Rejestruj czas trwania zadania (np. za pomocą Micrometer).
- Monitoruj błędy – nieprzechwycone wyjątki są tylko logowane.
- Możesz użyć AOP do centralnego logowania zadań.
- Warto dodać metryki (liczba wykonań, czas trwania, sukcesy/błędy).

Przykład logowania: Java

```java
log.info("[ZADANIE] Start: {}", nazwa);
try {
    // logika zadania
    log.info("[ZADANIE] Koniec: {}", nazwa);
} catch (Exception e) {
    log.error("[ZADANIE] Błąd: {}", e.getMessage());
}
```

11 Pułapki

- ✗ Długie zadania blokujące wątek.
- ✗ Brak obsługi wyjątków.
- ✗ Nakładanie się wykonań tego samego zadania.
- ✗ Niepoprawne wyrażenia CRON.
- ✗ Różnice stref czasowych między środowiskami.
- ✗ Zbyt wiele zadań bez konfiguracji puli wątków.
- ✗ Brak monitorowania i logowania.
- ✗ Zależność od czasu systemowego (np. zmiana czasu letniego).

12 Szybkie podsumowanie

- Używaj @Scheduled do prostych zadań cyklicznych.
- Włącz planowanie przez @EnableScheduling.
- Wybierz odpowiedni typ harmonogramu (fixedRate, fixedDelay, initialDelay, cron).
- Pamiętaj o strefie czasowej.
- Dbaj o to, aby zadania się nie nakładały.
- Skonfiguruj pulę wątków dla wielu zadań.
- Loguj i monitoruj wykonania.
- Stosuj dobre praktyki i obsługuj wyjątki.
- Planuj zadania świadomie – prosto, ale niezawodnie.

Małe zadania
wielkie możliwości
– automatyzuj
mądrze!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 66/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

SPRING CACHE

@Cacheable, @CachePut i @CacheEvict w praktyce

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest cache i dlaczego pomaga?

Cache to tymczasowe przechowywanie danych, aby szybciej na nie odpowiadać.

Zmniejsza liczbę kosztownych operacji (np. zapytań do bazy, wywołań zewnętrznych API).

Poprawia wydajność, skraca czas odpowiedzi i odciąża systemy zewnętrzne.

Cache jest szczególnie przydatny, gdy dane zmieniają się rzadziej niż są odczytywane.

2 Abstrakcja cache w Spring

- Spring Cache to abstrakcja, niezależna od technologii (np. Caffeine, Redis, Ehcache).
- Jednolite API i adnotacje – łatwa zmiana implementacji.
- Wspiera różne dostawcy cache przez CacheManager.
- Działa przez AOP (proksy), więc adnotacje umieszcza się na publicznych metodach.
- Skup się na logice biznesowej, a szczegóły cache zostaw Springowi.

Ta sama aplikacja może działać z cache w pamięci (np. Caffeine) lub rozproszonym (np. Redis), bez zmiany kodu.

3 Włączanie cache w aplikacji

- Dodaj zależność do wybranego dostawcy (np. spring-boot-starter-cache + caching provider).
- Włącz obsługę cache w konfiguracji adnotacją @EnableCaching.
- Skonfiguruj CacheManager (opcjonalnie).

Java

```java
@SpringBootApplication
@EnableCaching
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

4 @Cacheable – cachowanie odczytów

- Zwraca dane z cache, jeśli istnieją.
- W przeciwnym razie wykonuje metodę i zapisuje wynik w cache.
- Najczęściej używana adnotacja.

Java

```java
@Service
public class UserService {

    @Cacheable(cacheNames = "users", key = "#id")
    public User findById(Long id) {
        // kosztowne zapytanie do bazy
        return userRepository.findById(id);
    }
}
```

5 @CachePut – zawsze aktualizuj

- Zawsze wykonuje metodę i zapisuje wynik w cache.
- Stosuj przy operacjach aktualizacji danych, gdy chcesz natychmiast odświeżyć cache.

Java

```java
@Service
public class UserService {

    @CachePut(cacheNames = "users", key = "#user.id")
    public User updateUser(User user) {
        // zapis do bazy
        return userRepository.save(user);
    }
}
```

6 @CacheEvict – usuwanie danych

- Usuwa wpis(y) z cache.
- Stosuj przy operacjach usuwania lub gdy dane stają się nieaktualne.
- Możesz usunąć konkretny klucz lub cały cache.

Java

```java
@Service
public class UserService {

    @CacheEvict(cacheNames = "users", key = "#id")
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }

    @CacheEvict(cacheNames = "users", allEntries = true)
    public void deleteAllUsers() {
        userRepository.deleteAll();
    }
}
```

7 @Caching i @CacheConfig

- **@Caching** pozwala użyć wielu operacji cache na jednej metodzie.
- **@CacheConfig** pozwala ustawić wspólne ustawienia dla całej klasy.

Java

```java
@Service
@CacheConfig(cacheNames = "products")
public class ProductService {

    @Caching(
        put = @CachePut(key = "#product.id"),
        evict = @CacheEvict(key = "#product.category.id")
    )
    public Product update(Product product) {
        return productRepository.save(product);
    }
}
```

8 Klucze cache (key)

- Klucz domyślnie tworzony jest z parametrów metody.
- Możesz użyć SpEL do definiowania własnych kluczy.
- Dla złożonych obiektów rozważ prostszy klucz (np. ID, unikalne pole).

Przykłady kluczy:

- `key = "#id"` – pojedynczy parametr
- `key = "#user.id"` – pole obiektu
- `key = "T(java.util.Objects).hash(#param)"`
- `key = "#root.methodName + ":" + #id"`

9 Warunki cache: condition i unless

- **condition** – określa, czy metoda ma być cache'owana (przed wykonaniem).
- **unless** – pozwala zrezygnować z zapisu do cache (po wykonaniu, na podstawie wyniku).

Java

```java
@Cacheable(
    cacheNames = "users",
    key = "#id",
    condition = "#id > 0",
    unless = "#result == null"
)
public User findById(Long id) {
    return userRepository.findById(id);
}
```

10 Wybór dostawcy cache

- **Caffeine** – szybki, w pamięci, idealny do pojedynczej aplikacji (domyślny wybór).
- **Redis** – rozproszony cache, dobry dla wielu instancji.
- **Ehcache** – sprawdzony, prosty w konfiguracji.
- **Inne** – np. Hazelcast, Infinispan.

Wybór zależy od wymagań: czy cache ma być w pamięci aplikacji, czy współdzielony między instancjami.

11 Unieważnianie cache – podstawy

- Usuwaj cache przy zmianie danych (write-through).
- Stosuj @CacheEvict lub @CachePut.
- Rozważ strategię TTL (czas życia wpisów).
- Dla powiązanych danych usuń wiele kluczy lub cały cache (allEntries = true).
- Przy złożonych zależnościach rozważ zdarzenia domenowe.

Dobre unieważnianie cache zapobiega problemom z nieaktualnymi danymi.

12 Najczęstsze zastosowania

- Dane referencyjne (np. słowniki, konfiguracja).
- Wyniki zapytań do bazy danych.
- Odpowiedzi z zewnętrznych API.
- Agregacje i raporty.
- Listy produktów, użytkowników, uprawnienia.

13 Pułapki i problemy

- Nieaktualne dane (stale data) – brak właściwego unieważniania.
- Samowywołanie metod – adnotacje nie zadziałają (obiekt nie przechodzi przez proxy).
- Cache'owanie zbyt wielu danych – rosnące zużycie pamięci.
- Zbyt złożone klucze lub niestabilne obiekty jako klucz.
- Brak obsługi wyjątków – nie cache'uj błędów.
- Niewłaściwy TTL lub brak strategii czyszczenia.

14 Dobre praktyki

- Cache'uj tylko to, co ma sens i zmienia się rzadko.
- Używaj prostych, stabilnych kluczy.
- Zadbaj o unieważnianie po zmianach danych.
- Ustal rozsądny TTL i monitoruj rozmiar cache.
- Testuj scenariusze cache (hit/miss/evict).
- Unikaj cache'owania obiektów z wrażliwymi danymi.
- Dokumentuj strategię cache w projekcie.

Szybsze aplikacje to szczęśliwsi użytkownicy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

MAPA 67/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

REDIS

Cache, dane i typowe zastosowania

Lepsze testy
Lepsza dokumentacja
Większa pewność!

1 Czym jest Redis?

- Redis (Remote Dictionary Server) to szybka, otwartoźródłowa baza danych typu key-value.
- Działa w pamięci RAM (in-memory), co zapewnia bardzo niskie opóźnienia.
- Wspiera trwałość danych, replikację, klastrowanie i bogaty zestaw struktur danych.
- Popularny do cache'owania, przechowywania sesji, kolejek, limiterów i wielu innych zastosowań.

2 Podstawowe struktury danych

String	Prosta wartość tekstowa, np. liczba, napis, JSON. Najprostszy typ danych.
Hash	Mapa pól i wartości, np. dane użytkownika: `{"id": "1", "name": "Jan" }`.
List	Uporządkowana lista elementów. Obsługa jak kolejka (FIFO/LIFO).
Set	Nieuporządkowany zbiór unikalnych elementów, np. tagi użytkownika.
Sorted Set (ZSet)	Zbiór z przypisaną oceną (score). Elementy są uporządkowane, np. ranking wyników.

3 Redis jako cache czy baza danych?

Cache

- Szybki dostęp do danych.
- Dane mogą być odtwarzane z innego źródła.
- Typowe zastosowania: wyniki zapytań, dane słownikowe, sesje, tokeny.
- Krótszy czas życia danych (TTL).

Baza danych (datastore)

- Przechowywanie danych biznesowych.
- Wyższe wymagania dotyczące trwałości i spójności.
- Używany, gdy potrzebna jest bardzo wysoka wydajność.
- Dane często o dłuższym czasie życia.

Redis świetnie sprawdza się jako **cache**, ale może być również używany jako pełnoprawna baza danych – w zależności od potrzeb.

4 Redis w Spring Boot

- Spring Boot zapewnia świetne wsparcie dla integracji z Redisem.
- Używamy Spring Data Redis.
- Możemy korzystać z RedisTemplate (niższy poziom) lub gotowych abstrakcji (np. @Cacheable, Spring Session).
- Łatwa konfiguracja przez properties i auto-konfigurację.

5 Konfiguracja w Spring Boot

dependencies (pom.xml): XML

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

application.yml: YAML

```yaml
spring:
  data:
    redis:
      host: localhost
      port: 6379
      password:
      database: 0
```

6 Przykład użycia RedisTemplate

Java

```java
@Service
public class UserCacheService {

  private final RedisTemplate<String, Object> redisTemplate;

  public UserCacheService(RedisTemplate<String, Object> redisTemplate) {
    this.redisTemplate = redisTemplate;
  }

  public void saveUser(String id, User user) {
    redisTemplate.opsForValue().set("user:" + id, user, Duration.ofMinutes(30));
  }

  public User getUser(String id) {
    return (User) redisTemplate.opsForValue().get("user:" + id);
  }
}
```

7 Typowe zastosowania

Cache danych	Wyniki zapytań, dane słownikowe, konfiguracja.
Sesje użytkowników	Przechowywanie sesji (np. Spring Session).
Rate limiting	Ograniczanie liczby żądań (np. per użytkownik / IP).
Kolejki i messaging	Proste kolejki, obsługa zadań asynchronicznych.
Rankingi / statystyki	Tablice wyników (Sorted Set), liczniki, analityka w czasie rzeczywistym.

8 TTL i wygąśanie danych

- Redis pozwala ustawić czas życia klucza (TTL – Time To Live).
- Po upływie TTL klucz jest automatycznie usuwany.
- Można ustawić TTL przy zapisie lub osobno (EXPIRE).
- Przydatne w cache, sesjach, limiterach.

Przykład: Java

```java
redisTemplate.opsForValue()
   .set("token:123", "abc", Duration.ofMinutes(10));

// lub osobno
redisTemplate.expire("token:123", Duration.ofMinutes(10));
```

9 Serializacja danych

- Redis przechowuje dane jako bajty – musimy zadbać o odpowiednią serializację i deserializację.
- Najczęściej używane formaty:
  - **JSON** (np. Jackson).
  - **binarny** (JDK serialization).
  - prostsze formaty (String, Long, itp.).
- W Spring Boot możemy skonfigurować serializer w RedisTemplate.
- W przypadku obiektów niestandardowych warto używać JSON (lepsza interoperacyjność).

Przykład (konfiguracja JSON): Java

```java
redisTemplate.setValueSerializer(
        new GenericJackson2JsonRedisSerializer());
```

10 Trwałość danych (persistence)

Redis może zapisywać dane na dysku, dzięki czemu przetrwają restart serwera.

RDB (snapshoty)

- Okresowe zapisywanie całego zrzutu bazy do pliku.
- Prostsza konfiguracja.
- Wyższa wydajność.
- Ryzyko utraty ostatnich zmian (od ostatniego zrzutu).

AOF (Append Only File)

- Zapisuje każdą operację w formie logu.
- Większe bezpieczeństwo danych.
- Może być większy rozmiar pliku.
- Można łączyć RDB i AOF.

11 Dobre praktyki

- Używaj TTL dla danych cache i sesji.
- Wybierz odpowiednie struktury danych.
- Zadbaj o właściwą serializację (np. JSON).
- Monitoruj wykorzystanie pamięci.
- Stosuj nazewnictwo kluczy (np. prefixy: "user:", "cache:").
- Używaj Spring Cache (np. @Cacheable) dla prostych scenariuszy.
- Rozważ replikację i klastrowanie w produkcji.
- Testuj integrację (np. Testcontainers + Redis).

12 Najczęstsze pułapki

- Brak TTL – niekontrolowany wzrost danych.
- Nieodpowiednia serializacja obiektów.
- Zbyt duże obiekty w cache.
- Używanie Redis jako jedynego źródła prawdy bez backupu.
- Brak monitorowania (pamięć, klucze, latency).
- Nieoptymalne klucze (brak prefixów, trudne do zarządzania).
- Brak obsługi błędów (np. niedostępny Redis).
- Nadmierne używanie Redis tam, gdzie wystarczy cache lokalny.

13 Szybkie podsumowanie

- Redis to szybka baza danych in-memory.
- Wspiera wiele struktur danych (String, Hash, List, Set, Sorted Set).
- Świetnie sprawdza się jako cache, sesja, limiter czy kolejka.
- Łatwa integracja w Spring Boot.
- Możemy ustawić TTL dla kluczy.
- Ważny jest wybór odpowiedniej serializacji.
- Obsługuje trwałość danych (RDB/AOF).
- Stosuj dobre praktyki i unikaj typowych błędów.

Szybkie dane to szczęśliwsi użytkownicy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

KAFKA W SPRING BOOT

Producer, consumer i consumer groups

MAPA 68/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest Apache Kafka?

Apache Kafka to rozproszona platforma do strumieniowania danych (event streaming), umożliwiająca niezawodne, skalowalne i szybkie przesyłanie komunikatów między systemami.

- Projekt z ekosystemu Apache.
- Wysoka przepustowość i niskie opóźnienia.
- Trwałe przechowywanie komunikatów.
- Idealna do systemów zdarzeniowych i przetwarzania w czasie rzeczywistym.

2 Podstawowe pojęcia

Broker – serwer Kafka, który przechowuje dane i obsługuje żądania klientów.

Topic – logiczny kanał (kolejka) komunikatów.

Partition – podział topica na niezależne, uporządkowane fragmenty danych.

Offset – unikalny, rosnący identyfikator komunikatu w partycji.

Komunikaty w partycji są uporządkowane i identyfikowane przez offset.

3 Rola producenta i konsumenta

Producer (producent)

- Wysyła komunikaty do wskazanego topica.
- Wybiera partycję (automatycznie lub ręcznie).
- Może działać synchronicznie lub asynchronicznie.

Consumer (konsument)

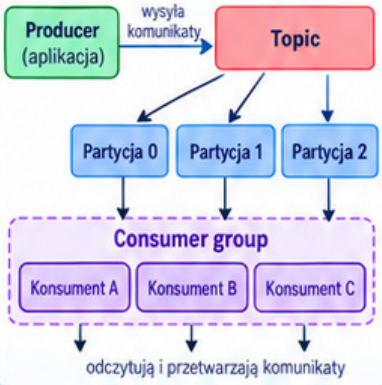
- Odczytuje komunikaty z topica.
- Przetwarza dane w aplikacji.
- Zapisuje swój postęp za pomocą offsetów.

Kafka decoupluje producentów i konsumentów – mogą działać niezależnie, w różnym tempie.

4 Consumer groups

- Grupa konsumentów pozwala skalować przetwarzanie danych.
- Każda partycja jest czytana przez dokładnie jednego konsumenta w grupie.
- Wiele instancji tej samej aplikacji może należeć do jednej grupy.
- Jeśli konsument przestanie działać, partycje zostaną przekazane innemu członkowi grupy.

5 Przepływ komunikatów



6 Spring Kafka – podstawy

Spring Kafka to oficjalny moduł Springa do integracji z Apache Kafka.

- Uproszcza konfigurację producentów i konsumentów.
- Wykorzystuje auto-konfigurację Spring Boot.
- Działa z KafkaTemplate, @KafkaListener i innymi narzędziami.
- Obsługuje serializację, deserializację, retry, kontrakt błędów, transakcje i więcej.

7 Wysyłanie komunikatów

KafkaTemplate

```java
@Service
public class OrderProducer {
  private final KafkaTemplate<
    String, OrderEvent> kafkaTemplate;

  public OrderProducer(
    KafkaTemplate<String, OrderEvent>
    kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
  }

  public void sendOrder(OrderEvent event) {
    kafkaTemplate.send("orders", event);
  }
}
```

8 Odbieranie komunikatów

@KafkaListener

```java
@Service
public class OrderConsumer {

  @KafkaListener(topics = "orders",
      groupId = "order-group")
  public void handleOrder(OrderEvent event) {
    System.out.println("Odebrano: " + event);
    // przetwarzanie komunikatu
  }
}
```

9 Serializacja danych

Najczęściej używane formaty:

- **JSON** – prosty i czytelny (np. Jackson)
- **Avro** – wydajny, ze schematem (Schema Registry)
- **Protobuf** – kompaktowy i szybki
- **String / byte[]** – proste przypadki

- W Spring Kafka konfigurujemy Serializery/Deserializery.
- Można użyć własnych (np. JSON + Jackson).
- Dla Avro warto użyć Confluent Schema Registry.

10 Kolejność i skalowanie

Kolejność komunikatów jest gwarantowana tylko wewnątrz jednej partycji.

- Aby skalować przetwarzanie, zwiększ liczbę partycji.
- Liczba konsumentów w grupie nie może być większa niż liczba partycji.
- Do kluczowania komunikatów użyj klucza (np. userId), aby zawsze trafiały do tej samej partycji.

11 Retry i obsługa błędów

- Automatyczne ponowienia (retry) przy przejściowych błędach.
- Możliwość konfiguracji backoff (np. ExponentialBackOff).
- Obsługa błędów przez DefaultErrorHandler.
- Nieobsługiwane komunikaty można kierować do tzw. **DLQ** (Dead Letter Queue).
- Kontrola liczby prób i strategii (postprocessing, pomijanie, DLQ).

12 Najczęstsze przypadki użycia

- Obsługa zdarzeń biznesowych (np. zamówienia, płatności).
- Integracja między mikroserwisami.
- Logi i metryki w czasie rzeczywistym.
- Przetwarzanie strumieni danych.
- Systemy powiadomień.
- Event sourcing i CQRS.
- Integracja z systemami zewnętrznymi (np. analityka, hurtownie danych).

13 Częste pułapki

- Brak klucza – problemy z kolejnością.
- Zbyt mała liczba partycji.
- Nieobsługa błędów – utrata komunikatów.
- Nieodpowiednia serializacja danych.
- Zbyt duże komunikaty.
- Zbyt długie przetwarzanie w konsumentach.
- Niewłaściwe zarządzanie offsetami.
- Brak monitoringu (lag konsumentów).

14 Dobre praktyki

- Używaj kluczy do partycjonowania.
- Planuj liczbę partycji z zapasem.
- Obsługuj błędy i stosuj DLQ.
- Monitoruj lag i metryki aplikacji.
- Stosuj sensowny rozmiar komunikatów.
- Używaj idempotentnych konsumentów.
- Dokumentuj kontrakty komunikatów (np. schematy).
- Testuj integracje (np. Testcontainers).

15 Szybkie podsumowanie

- Kafka to niezawodny, skalowalny system strumieniowania danych.
- Producer wysyła komunikaty, consumer je przetwarza.
- Consumer groups umożliwiają skalowanie.
- Spring Kafka upraszcza integrację.
- Właściwa serializacja, obsługa błędów i monitorowanie to klucz do stabilnej aplikacji.

Wielkie systemy tworzą małe eventy! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 – INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

RabbitMQ i AMQP

Exchange, queue, binding i routing

MAPA 69/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Czym jest RabbitMQ?

- RabbitMQ to popularny broker wiadomości typu open source.
- Umożliwia asynchroniczną komunikację między aplikacjami.
- Opiera się na protokole AMQP (Advanced Message Queuing Protocol).

Pozwala na niezawodną, skalowalną i elastyczną wymianę wiadomości między usługami.

2 Najważniejsze pojęcia AMQP

Wiadomość (message)
– dane przesyłane między aplikacjami.

Exchange
– przyjmuje wiadomości od producenta i decyduje, do której kolejki je przekazać.

Kolejka (queue)
– przechowuje wiadomości, które czekają na przetworzenie przez konsumentów.

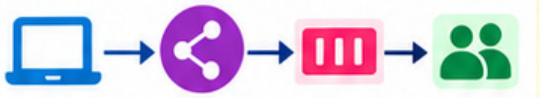
Routing key
– klucz routingu określający, dokąd ma trafić wiadomość.

Binding
– powiązanie kolejki z exchange'em, z opcjonalnym kluczem routingu.

3 Rodzaje exchange'ów

Direct	• Przekazuje wiadomości do kolejek, których binding ma dokładnie pasujący routing key. • Prosty i przewidywalny.
Topic	• Wykorzystuje wzorce w routing key (np. user.*, order.created). • Elastyczny, obsługuje wiele kolejek.
Fanout	• Przekazuje wiadomość do wszystkich powiązanych kolejek. • Ignoruje routing key. • Dobry do powiadomień (broadcast).
Headers	• Routowanie na podstawie nagłówków wiadomości (zamiast routing key). • Przydatny, gdy reguły routingu są bardziej złożone.

4 Jak to działa? – przepływ wiadomości



Producent wysyła wiadomość z routing key

Exchange kieruje wiadomość na podstawie bindingów i typu

Kolejka przechowuje wiadomości

Konsument pobiera i przetwarza wiadomości

Producent nie wie, który konsument przetworzy wiadomość. Exchange decyduje, do której kolejki trafi wiadomość.

5 Potwierdzenia (ack)

- Po przetworzeniu wiadomości konsument wysyła potwierdzenie (ack).
- Jeśli ack nie zostanie wysłany, wiadomość może zostać ponownie dostarczona.
- Możliwe tryby potwierdzania:
  - automatyczne (auto-ack) – mniej bezpieczne,
  - ręczne (manual ack) – większa kontrola,
  - transakcyjne – dla krytycznych scenariuszy.

Dzięki ack unikamy utraty wiadomości i możemy zapewnić przetwarzanie co najmniej raz.

6 Dead-Letter Queue (DLQ)

- Jeśli wiadomość nie może być przetworzona (np. błąd, przekroczona liczba prób), może trafić do specjalnej kolejki DLQ.
- Korzysta z mechanizmu dead-letter exchange oraz ustawień w kolejce (x-dead-letter-exchange, x-dead-letter-routing-key).
- Pozwala na analizę problematycznych wiadomości i ich późniejsze ponowne przetworzenie.

DLQ zwiększa niezawodność systemu i ułatwia obsługę błędów.

7 Spring AMQP – podstawowa konfiguracja

- Spring AMQP to moduł Springa ułatwiający pracę z RabbitMQ.
- Zapewnia integrację przez **RabbitTemplate**, **@RabbitListener** oraz prostą konfigurację beans.
- Obsługuje konwersję wiadomości (np. na JSON).
- Wspiera potwierdzenia, transakcje i retry.

8 Przykład konfiguracji (exchange, queue, binding)

Java

```java
@Configuration
public class RabbitConfig {
    @Bean
    public DirectExchange exchange() {
        return new DirectExchange("app.exchange");
    }
    @Bean
    public Queue queue() {
        return QueueBuilder.durable("app.queue").build();
    }
    @Bean
    public Binding binding(Queue queue, DirectExchange exchange) {
        return BindingBuilder.bind(queue)
                .to(exchange)
                .with("app.key");
    }
}
```

9 Producent – wysyłanie wiadomości

Java

```java
@Service
public class MessageProducer {

    private final RabbitTemplate rabbitTemplate;

    public MessageProducer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }
    public void wyslijWiadomosc(String tresc) {
        rabbitTemplate.convertAndSend(
            "app.exchange",
            "app.key",
            tresc
        );
    }
}
```

10 Konsument – @RabbitListener

Java

```java
@Service
public class MessageConsumer {

    @RabbitListener(queues = "app.queue")
    public void odbierzWiadomosc(String tresc) {
        System.out.println("Otrzymano: " + tresc);
        // tutaj logika przetwarzania wiadomości
    }
}
```

11 Przykłady zastosowań

- **Zamówienia** – przetwarzanie w tle
- **Powiadomienia** – e-mail, SMS, push
- Generowanie raportów
- Integracja z systemami zewnętrznymi
- Przetwarzanie dużych zbiorów danych
- Zdarzenia domenowe (event-driven)
- Komunikacja między mikroserwisami
- Buforowanie żądań i kolejkowanie zadań

12 Pułapki, dobre praktyki i podsumowanie

Najczęstsze pułapki

- Brak obsługi błędów i DLQ.
- Niepotwierdzane wiadomości (auto-ack).
- Zbyt duże wiadomości.
- Nieodpowiedni typ exchange'a.
- Brak monitoringu i logowania.
- Niewłaściwa konfiguracja retry (pętle błędów).

Dobre praktyki

- Używaj DLQ i retry z limitem.
- Stosuj ręczne potwierdzenia.
- Projektuj małe, niezależne wiadomości.
- Wybieraj odpowiedni typ exchange'a.
- Monitoruj kolejki i metryki.
- Testuj scenariusze awaryjne.
- Dokumentuj kontrakty wiadomości (format, klucze).

13 Szybkie podsumowanie

- RabbitMQ to niezawodny broker wiadomości oparty na AMQP.
- Exchange, queue, binding i routing key to kluczowe pojęcia.
- Direct, topic, fanout i headers to różne strategie routingu.
- Spring AMQP ułatwia integrację (RabbitTemplate, @RabbitListener).
- Ack zapewnia niezawodność przetwarzania.
- DLQ pozwala obsługiwać nieudane wiadomości.
- RabbitMQ sprawdza się w wielu scenariuszach biznesowych.
- Dobre praktyki i monitoring zwiększają stabilność systemu.

Asynchroniczność daje skalowalność, odporność i swobodę rozwoju Twojej aplikacji!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 07 • INTEGRACJE, ASYNCHRONICZNOŚĆ I MESSAGING

NIEZAWODNA KOMUNIKACJA

Idempotencja, retry, DLQ, potwierdzenia i transactional outbox

MAPA 70/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze testy
Lepsze aplikacje
Większa pewność!

1 Dlaczego niezawodność jest ważna?

Systemy rozproszone zawodzą.
To normalne, nie wyjątek!

Sieć, serwery, bazy, komunikaty mogą być niedostępne, opóźnione lub dostarczone wielokrotnie.

Musimy **projektować** aplikacje na takie sytuacje, aby dane nie zginęły, nie były przetwarzane wielokrotnie i system pozostał spójny.

Niezawodność to podstawa skalowalnych, odpornych i dobrze działających systemów rozproszonych.

2 Idempotencja – ten sam efekt za każdym razem

- Idempotentna operacja daje ten sam rezultat niezależnie od liczby wywołań.
- Chroni przed duplikatami komunikatów i retry.
- Klucz idempotencyjny (np. orderId, messageId) pozwala rozpoznać, czy operacja była już wykonana.
- Stosuj unikalne klucze i sprawdzanie istnienia rekordu.

Przykład (sprawdzanie po kluczu): Java

```java
@Transactional
public void processOrder(String orderId, OrderDto dto) {
    if (orderRepository.existsById(orderId)) {
        log.info("Zamówienie {} już przetworzone", orderId);
        return; // idempotentne – nic nie robimy
    }
    // wykonaj operację
    orderRepository.save(mapToOrder(dto));
}
```

3 Retry i backoff – spróbuj ponownie

- Używaj retry dla tymczasowych błędów (np. problemy sieciowe, 5xx).
- Stosuj wykładniczy backoff (coraz dłuższe odstępy między próbami).
- Ustal maksymalną liczbę prób.
- Unikaj retry dla błędów biznesowych (4xx).
- Dodaj jitter, aby uniknąć efektu „thundering herd".



4 Dead-Letter Queue (DLQ)

- Nie każdy komunikat da się przetworzyć od razu (lub wcale).
- Po przekroczeniu liczby prób, komunikat trafia do DLQ.
- Pozwala to odseparować problemy i nie blokować przetwarzania kolejki.
- Wymaga monitorowania i procesu obsługi komunikatów z DLQ.



5 Potwierdzenia i potwierdzanie odbioru

- Zawsze używaj potwierdzeń (ack) po pomyślnym przetworzeniu komunikatu.
- W przypadku błędu nie potwierdzaj – komunikat będzie ponownie dostarczony.
- Wybierz odpowiedni tryb ack (auto / manual).
- Przetwarzaj komunikat w transakcji i potwierdzaj dopiero po zapisaniu danych.

Przykład (Kafka – ręczne potwierdzenie): Java

```java
@KafkaListener(topics = "orders", containerFactory = "kafkaListener")
public void handle(OrderCreated event, Acknowledgment ack) {
    try {
        orderService.process(event); // idempotentnie
        ack.acknowledge(); // potwierdzamy po sukcesie
    } catch (Exception e) {
        log.error("Błąd przetwarzania", e);
        // brak ack – komunikat zostanie dostarczony ponownie
        throw e;
    }
}
```

6 Exactly-once vs At-least-once

Exactly-once (idealnie)	At-least-once (praktycznie)
Komunikat przetwarzany dokładnie raz.	Komunikat przetwarzany co najmniej raz.
Trudne do osiągnięcia w praktyce.	Możliwe duplikaty (należy stosować idempotencję).
Wymaga specjalnego wsparcia infrastruktury (np. Kafka EOS).	Prostsze i powszechnie stosowane podejście.
Zwykle droższe i bardziej złożone.	W większości przypadków wystarczające.

Zasada: zakładaj, że komunikat może być dostarczony więcej niż raz i projektuj aplikację z idempotencją.

7 Transactional Outbox – spójność danych

- Rozwiązuje problem braku transakcji między bazą danych a brokerem komunikatów.
- Zapisujemy zdarzenie w tabeli outbox w tej samej transakcji co dane biznesowe.
- Osobny proces (publisher) odczytuje outbox i publikuje komunikaty do brokera.
- Po udanej publikacji oznaczamy komunikat jako wysłany.



8 Deduplikacja komunikatów

- Broker może dostarczyć ten sam komunikat wielokrotnie (retry, failover).
- Stosuj unikalne klucze (messageId, eventId).
- Przechowuj już przetworzone komunikaty (np. w bazie lub cache, z TTL).
- Deduplikacja powinna być szybka i skalowalna.

Przykład (prosta deduplikacja): Java

```java
if (processedMessageRepository.existsById(eventId)) {
    log.info("Już przetworzono komunikat {}", eventId);
    return;
}
processedMessageRepository.save(
    new ProcessedMessage(eventId, Instant.now())
);
// dalej przetwarzamy komunikat
```

9 Monitoring i obserwowalność

- Monitoruj opóźnienia, liczbę retry i komunikaty w DLQ.
- Śledź wskaźniki: sukcesy, błędy, czas przetwarzania.
- Używaj logów diagnostycznych z identyfikatorem komunikatu (correlationId / messageId).
- Dodaj metryki (np. Micrometer + Prometheus).
- Twórz alerty na wzrost błędów, rozmiar DLQ i opóźnienia.

Metryki | Logi (correlationId) | Tracing (rozproszony ślad) | Alerty

10 Dobre praktyki

- Projektuj operacje w sposób idempotentny.
- Stosuj retry z wykładniczym backoffem i jitterem.
- Używaj DLQ i regularnie ją obsługuj.
- Zastosuj wzorzec transactional outbox.
- Monitoruj i ustawiaj sensowne alerty.
- Testuj scenariusze awarii (chaos, integracyjne).

11 Najczęstsze pułapki

- Brak idempotencji – duplikaty w danych.
- Nieograniczone retry – przeciążenie systemu.
- Brak obsługi DLQ – utracone komunikaty.
- Potwierdzanie przed zapisem danych.
- Brak outboxa – niespójność danych i zdarzeń.
- Ignorowanie duplikatów komunikatów.
- Słaby monitoring – problemy wykrywane za późno.
- Założenie, że „to się nie zdarzy".

12 Szybkie podsumowanie

- Systemy rozproszone zawodzą – projektuj na awarie.
- Idempotencja chroni przed duplikatami.
- Stosuj retry z backoffem i jitterem.
- Używaj DLQ dla trudnych komunikatów.
- Potwierdzaj przetworzenie po sukcesie.
- Transactional outbox zapewnia spójność danych i zdarzeń.
- Monitoruj, mierz i reaguj na problemy.
- Zakładaj dostarczenie co najmniej raz.



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

SPRING BOOT ACTUATOR

Mapa możliwości — endpointy, konfiguracja i diagnostyka

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

MAPA 71/100

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Czym jest Actuator?

Spring Boot Actuator to zestaw wbudowanych funkcji do monitorowania i zarządzania aplikacją w czasie rzeczywistym.

Udostępnia informacje o stanie, metrykach, środowisku i konfiguracji.

Ułatwia diagnozowanie problemów i analizę działania aplikacji.

Jest nieoceniony w środowiskach produkcyjnych, testowych i developerskich.

2 Starter i konfiguracja

Aby użyć Actuator, dodaj zależność **spring-boot-starter-actuator** do pliku build (np. Maven lub Gradle).

Maven (pom.xml):

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Gradle (build.gradle):

```
implementation 'org.springframework.boot:
 spring-boot-starter-actuator'
```

Actuator działa od razu po dodaniu zależności – wystarczy konfiguracja ekspozycji endpointów.

3 Najważniejsze endpointy

/actuator/health	Informacja o stanie aplikacji i jej zależności (np. baza danych).
/actuator/info	Dodatkowe informacje o aplikacji (np. wersja, opis, dane niestandardowe).
/actuator/metrics	Dostęp do metryk aplikacji (np. użycie pamięci, liczba żądań).
/actuator/env	Informacje o zmiennych środowiskowych i właściwościach konfiguracji.
/actuator/loggers	Podgląd i zmiana poziomów logowania w czasie działania.
/actuator/beans	Lista wszystkich beanów w kontekście aplikacji.
/actuator/mappings	Informacje o mapowanych endpointach (żądania HTTP).
/actuator/threaddump	Zrzut wątków aplikacji (do analizy problemów z wydajnością).

4 Ekspozycja endpointów

Domyślnie w Spring Boot dostępne są tylko wybrane endpointy, Możesz jawnie określić, które endpointy udostępnić przez HTTP.

Użyj właściwości: **management.endpoints.web.exposure.include** aby wskazać listę dozwolonych endpointów.

Unikaj ekspozycji wszystkich endpointów (*) w środowisku produkcyjnym.

5 HTTP i JMX

Dostęp przez HTTP

- Najczęściej używana forma dostępu.
- Endpointy dostępne pod ścieżką /actuator.
- Można je ograniczyć i zabezpieczyć.

Dostęp przez JMX

- Alternatywny sposób dostępu.
- Przydatny w środowiskach korporacyjnych i narzędziach JConsole.
- Można włączyć niezależnie od HTTP.
- Dobrze sprawdza się w środowiskach, gdzie nie ma dostępu HTTP.

6 Przykład application.yml

YAML

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,env,loggers
    endpoint:
      health:
        show-details: when-authorized
      info:
        enabled: true
  info:
    app:
      name: moja-aplikacja
      description: Przykładowa aplikacja
      version: 1.0.0
```

7 Bezpieczeństwo podstawowe

Endpointy Actuatora mogą zawierać wrażliwe informacje, dlatego należy je odpowiednio zabezpieczyć.

- Nie udostępniaj wszystkich endpointów publicznie.
- Stosuj autoryzację (np. Spring Security).
- Ogranicz dostęp do endpointów wrażliwych (np. /env, /beans, /threaddump).
- Używaj sieci wewnętrznych, VPN lub firewall.
- W środowisku produkcyjnym stosuj zasadę minimalnych uprawnień.

8 W PRAKTYCE

Zespoły wykorzystują Actuator do:

- Monitorowania stanu aplikacji w systemach produkcyjnych.
- Integracji z narzędziami do monitoringu (np. Prometheus, Grafana).
- Zbierania metryk i analizy wydajności.
- Szybkiej diagnozy problemów (np. z wątkami, pamięcią, konfiguracją).
- Sprawdzania gotowości aplikacji (np. health checks w orkiestratorach).
- Zarządzania logowaniem w locie (bez restartu).
- Wspierania zespołów operacyjnych i DevOps.

9 Najczęstsze pułapki

- Ekspozycja wszystkich endpointów (*) w środowisku produkcyjnym.
- Brak zabezpieczeń (brak autoryzacji).
- Ignorowanie szczegółów health (show-details: never).
- Brak analizy i wykorzystania metryk.
- Udostępnianie endpointów wrażliwych publicznie (np. /env, /beans).
- Traktowanie Actuatora jako zbędnego dodatku.

10 SZYBKIE PODSUMOWANIE

- Actuator dostarcza endpointy do monitorowania i diagnostyki.
- Dodaj spring-boot-starter-actuator.
- Udostępniaj tylko wybrane endpointy.
- Zabezpieczaj dostęp do endpointów.
- Korzystaj z HTTP lub JMX.
- Wykorzystuj metryki i informacje w praktyce.
- Unikaj najczęstszych pułapek.



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

HEALTH, LIVENESS I READINESS

Stan aplikacji, gotowość i sondy w praktyce

MAPA 72/100

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Czym jest /actuator/health

/actuator/health to endpoint w Spring Boot Actuator, który zwraca informacje o stanie aplikacji i jej zależnościach.

Umożliwia automatyczne sprawdzanie czy aplikacja działa i może obsługiwać ruch.

Wykorzystywany przez mechanizmy monitorowania, load balancery i orchestratory (np. Kubernetes).

Nie zwraca tylko stanu aplikacji, ale także stan zależności (np. baza danych, kolejki, zewnętrzne systemy).

2 Kody/statusy

Endpoint /actuator/health zwraca ogólny status aplikacji. Możliwe statusy to m.in.:

UP	Aplikacja działa prawidłowo, wszystkie sprawdzone komponenty są dostępne.
DOWN	Aplikacja nie działa, co najmniej jeden obowiązkowy komponent jest niedostępny.
OUT_OF_SERVICE	Aplikacja działa, ale nie powinna obsługiwać ruchu (np. tryb serwisowy, brak zależności).
UNKNOWN	Nie udało się określić stanu komponentu (np. błąd sprawdzania lub brak implementacji).

3 Liveness vs Readiness

Liveness (żywotność)

Czy aplikacja nadal działa?

- Informuje, czy proces aplikacji jest zdrowy i nie zawiesił się.
- Jeśli zwraca błąd – kontener zostanie zrestartowany.
- Nie powinien zależeć od zewnętrznych systemów (np. bazy danych).
- Służy do wykrywania zawieszeń i deadlocków.

Readiness (gotowość)

Czy aplikacja jest gotowa do obsługi ruchu?

- Sprawdza dostępność kluczowych zależności (np. bazy danych, kolejek).
- Jeśli zwraca błąd – ruch nie będzie kierowany do podu.
- Może zależeć od zewnętrznych systemów.
- Służy do kontrolowania ruchu i stopniowego startu.

4 HealthIndicator i własne wskaźniki

Spring Boot automatycznie dostarcza wiele wskaźników (DataSource, DiskSpace, Ping, Redis, itp.). Możemy też tworzyć własne sprawdzenia, implementując interfejs HealthIndicator.

```java
@Component
public class ZewnętrznySystemHealth
        implements HealthIndicator {

    @Override
    public Health health() {
        boolean ok = sprawdzPolaczenie();
        if (ok) {
            return Health.up()
                    .withDetail("system", "dostępny")
                    .build();
        } else {
            return Health.down()
                    .withDetail("system", "niedostępny")
                    .build();
        }
    }
}
```

5 Grupy health i detale odpowiedzi

Można definiować grupy health, aby oddzielić sprawdzenia dla różnych scenariuszy (np. liveness, readiness). Endpoint może zwracać szczegółowe informacje o komponentach.

Przykładowa odpowiedź JSON:

```json
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "PostgreSQL",
        "validationQuery": "SELECT 1"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 107374182400,
        "free": 53687091200,
        "threshold": 10485760
      }
    }
  }
}
```

6 Użycie w Kubernetes probes

Kubernetes wykorzystuje endpointy health do konfiguracji sond (probes):

livenessProbe	Sprawdza czy aplikacja działa. W razie błędu – restartuje kontener.
readinessProbe	Sprawdza czy aplikacja jest gotowa do obsługi ruchu. W razie błędu – pod zostaje usunięty z load balancera.
startupProbe	Opcjonalna sonda podczas uruchamiania aplikacji. Daje więcej czasu na start.

W Spring Boot najlepiej używać odpowiednich grup health (np. liveness i readiness), aby oddzielić sprawdzenia i uniknąć fałszywych restartów.

7 Przykładowa konfiguracja application.yml

Konfiguracja grup health i komponentów:

```yaml
management:
  endpoints:
    health:
      show-details: when-authorized
      probes:
        enabled: true
  health:
    group:
      liveness:
        include: ping
      readiness:
        include: db,redis,diskSpace
    defaults:
      enabled: true
```

8 Kiedy liveness, a kiedy readiness

Liveness powinno failować, gdy:

- Aplikacja przestała odpowiadać (np. deadlock).
- Główne wątki są zablokowane.
- Występuje błąd krytyczny uniemożliwiający działanie aplikacji.
- Proces jest w stanie nienaprawialnym.

Readiness powinno failować, gdy:

- Niedostępna jest baza danych lub inna kluczowa zależność.
- Aplikacja jest w trakcie inicjalizacji.
- Trwa migracja danych / rozruch systemu.
- Aplikacja działa, ale nie powinna obsługiwać ruchu (np. tryb konserwacji).

9 Najczęstsze pułapki

- Używanie tych samych sprawdzeń dla liveness i readiness.
- Sprawdzanie zewnętrznych systemów w liveness (powoduje niepotrzebne restarty).
- Brak konfiguracji probes.enabled w nowszych wersjach Spring Boot.
- Zbyt krótki czas początkowy (initialDelaySeconds).
- Ignorowanie szczegółów w odpowiedzi health (show-details).
- Niewłaściwe kody statusu (np. zawsze UP).
- Brak monitorowania i alertów dla stanu aplikacji.
- Zbyt wiele ciężkich sprawdzeń (spowalniają endpoint).

10 W PRAKTYCE

Jak wykorzystać health w codziennej pracy:

- [x] Skonfiguruj osobne grupy liveness i readiness.
- [x] Dodaj własne HealthIndicator dla kluczowych zależności.
- [x] Przetestuj zachowanie pod obciążeniem i w scenariuszach awaryjnych.
- [x] Obserwuj odpowiedzi /actuator/health w monitoringu.
- [x] Używaj sond w środowiskach kontenerowych (Kubernetes, OpenShift itp.).

11 Dobre praktyki

- [x] Oddziel liveness i readiness.
- [x] Używaj lekkich i szybkich sprawdzeń.
- [x] Uwzględniaj tylko krytyczne zależności w readiness.
- [x] Zwracaj szczegóły w środowiskach nieprodukcyjnych.
- [x] Testuj scenariusze awaryjne.
- [x] Monitoruj metryki i alertuj na zmiany stanu.
- [x] Regularnie weryfikuj konfigurację i progi czasowe.
- [x] Dokumentuj własne wskaźniki zdrowia.

12 Szybkie podsumowanie

- [x] Endpoint /actuator/health pokazuje stan aplikacji.
- [x] Możliwe statusy: UP, DOWN, OUT_OF_SERVICE, UNKNOWN.
- [x] Liveness sprawdza czy aplikacja działa (i restartuje w razie błędu).
- [x] Readiness sprawdza czy jest gotowa do ruchu.
- [x] Można tworzyć własne HealthIndicator i grupy health.
- [x] W Kubernetes używamy livenessProbe i readinessProbe.
- [x] Kluczem jest właściwa konfiguracja i testowanie.
- [x] Dobre praktyki zapewniają stabilność i niezawodność aplikacji.

Monitoruj
Diagnozuj
Reaguj szybciej
Buduj lepsze aplikacje!

SPRING BOOT

Obserwuj
to, co ważne
Twórz stabilne
systemy!

MONITORING
METRYKI
OBAGNOSTYKA
OBSERVABILITY
NIEZAWODNOŚĆ
ROZWÓJ

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

MICROMETER METRICS

Counter, Gauge, Timer i Distribution Summary

MAPA 73/100

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Czym jest Micrometer?

Micrometer to biblioteka do zbierania metryk w aplikacjach JVM (np. Spring Boot).

Zapewnia prosty, zunifikowany interfejs do instrumentacji kodu.

Jest niezależny od systemu monitorowania – te same metryki można wysłać do Prometheusa, Graphite, Datadoga i innych.

W Spring Bootcie Micrometer jest zintegrowany z Actuator i działa od razu po dodaniu zależności.

2 Registry – co to jest?

Registry to komponent, który przechowuje metryki i odpowiada za ich eksport do konkretnego systemu monitorowania.

W Spring Boot domyślnie dostępny jest SimpleMeterRegistry (do testów), a po dodaniu zależności – np. PrometheusMeterRegistry.

Tworzysz metryki przez Micrometer API, a registry zajmuje się ich publikacją w odpowiednim formacie (np. Prometheus).

Przykładowe registry:

- SimpleMeterRegistry (pamięć aplikacji)
- PrometheusMeterRegistry
- GraphiteMeterRegistry
- AtlasMeterRegistry, DatadogMeterRegistry

3 Counter – zliczanie zdarzeń

Counter służy do zliczania rosnącej liczby zdarzeń (np. liczby zapytań, błędów, operacji).

```
Counter counter = Counter.builder(
  "moj.aplikacja.zapytania")
  .description("Liczba otrzymanych
     zapytań HTTP")
  .tag("metoda", "GET")
  .register(meterRegistry);

counter.increment();
```

Używaj Counter tylko do wartości rosnących – nie dekrementuj i nie ustawiaj wartości ręcznie.

4 Gauge – wartość bieżąca

Gauge reprezentuje wartość bieżącą, która może rosnąć i maleć (np. liczba aktywnych połączeń, rozmiar kolejki).

```
AtomicInteger aktywne =
      new AtomicInteger(0);
Gauge.builder("moj.aplikacja.
   aktywne.polaczenia", aktywne,
   AtomicInteger::get)
   .description("Liczba aktywnych
     połączeń")
   .register(meterRegistry);

// później w kodzie
aktywne.incrementAndGet();
```

5 Timer – czas trwania operacji

Timer służy do pomiaru czasu trwania operacji i dodatkowo zlicza ich liczbę oraz udostępnia statystyki (średnia, percentyle, maksimum).

```
Timer timer = Timer.builder(
  "moj.aplikacja.obsluga.zapytania")
   .description("Czas obsługi zapytania")
   .tag("endpoint", "/api/produkty")
   .register(meterRegistry);

try (Timer.Sample sample = Timer.start(
     meterRegistry)) {
   // kod obsługi zapytania
   sample.stop(timer);
}
```

6 DistributionSummary – rozkład

DistributionSummary służy do pomiaru rozkładu wartości liczbowych (np. rozmiar żądań/odpowiedzi, liczba elementów w kolekcji).

```
DistributionSummary summary =
  DistributionSummary.builder(
    "moj.aplikacja.rozmiar.odpowiedzi")
   .description("Rozmiar odpowiedzi
    w bajtach")
   .baseUnit("bytes")
   .tag("endpoint", "/api/produkty")
   .register(meterRegistry);

summary.record(liczbaBajtow);
```

7 Tagi i ich znaczenie

Tagi to klucz-wartość, które dodają kontekst do wettryk (np. endpoint, metoda HTTP, status). Dzięki nim można filtrować i analizować dane.

- Używaj stabilnych, niskokardynalnych wartości tagów (np. nazwy endpointów).
- Nie używaj unikalnych wartości, np. identyfikatorów użytkowników czy UUID – powodują eksplozję metryk.
- Stosuj spójne nazewnictwo tagów w całej aplikacji.
- Najczęstsze tagi: method, uri, status, exception, outcome, application, instance.

8 Integracja z Actuator

Spring Boot automatycznie udostępnia zebrane metryki przez endpoint:

/actuator/metrics

Możesz też sprawdzić szczegóły danej metryki:

/actuator/metrics/{nazwa}

- Endpoint zwraca listę dostępnych metryk i ich wartości.
- Można filtrować metryki po nazwie i tagach.
- Działa od razu po dodaniu dependencji micrometer.

9 Eksport do Prometheusa

Aby eksportować metryki do Prometheusa, dodaj zależność i włącz endponit Prometheus.

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-
    prometheus</artifactId>
</dependency>
```

Po uruchomieniu metryki dostępne są pod:

/actuator/prometheus

- Prometheus zaczytuje metryki w formacie tekstowym (Prometheus exposition format).
- Możesz następnie skonfigurować scraping w pliku prometheus.yml.

10 Przykłady metryk JVM/HTTP

Spring Boot i Micrometer dostarczają domowych metryk JVM, systemowych i HTTP. Przykłady:

Metryki JVM:

- jvm.memory.used – użycie pamięci
- jvm.memory.max – maksymalna pamięć
- jvm.gc.pause – czas przerw GC
- jvm.threads.live – liczba wątków

Metryki HTTP (Spring Boot):

- http.server.requests – liczba i czas żądań
- http.server.requests.active – aktywne żądania
- http.server.requests.duration – czas obsługi
- http.server.requests.size – rozmiar żądań/odpowiedzi

Metryki systemowe:

- system.cpu.usage – użycie CPU
- system.load.average.1m – średnie obciążenie
- process.uptime – czas działania aplikacji

11 Dobre praktyki

- Nadaj czytelne, biznesowe nazwy metryk.
- Używaj odpowiedniego typu metryki (Counter, Gauge, Timer, DistributionSummary).
- Dodawaj sensowne tagi (o niskiej kardynalności).
- Monitoruj kluczowe ścieżki aplikacji (np. endpointy, operacje na bazie).
- Ustal alerty na istotne wskaźniki (np. błędy, czas odpowiedzi).
- Testuj metryki w środowiskach nieprodukcyjnych.
- Regularnie przeglądaj i usuwaj nieużywanie lub nadmiarowe metryki.

12 Najczęstsze pułapki

- Używanie Counter do wartości, które mogą maleć.
- Wysoka kardynalność tagów (np. ID użytkownika, UUID).
- Brak opisów (description) i jednostek (baseUnit).
- Ignorowanie metryk JVM i systemowych.
- Duża liczba nieużywanych metryk – obciążenie systemu monitoringu.
- Nieodpowiednie nazewnictwo – trudna analiza.
- Brak testów i weryfikacji metryk w różnych środowiskach.

13 Szybkie podsumowanie

- Micrometer umożliwia zbieranie metryk w aplikacji Spring Boot.
- Registry odpowiada za ich eksport do systemu monitorowania.
- Do wyboru: Counter, Gauge, Timer, DistributionSummary.
- Tagi dodają kontekst – używaj ich rozsądnie.
- Metryki są dostępne przez Actuator i mogą być eksportowane do Prometheusa.
- Wykorzystuj wbudowane metryki JVM/HTTP.
- Stosuj dobre praktyki i unikaj najczęstszych pułapek.

Monitoruj
Diagnozuj
Reaguj szybciej
Buduj lepsze aplikacje!

SPRING BOOT

Obserwuj
to, co ważne
Twórz stabilne
systemy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

WŁASNE METRYKI I CARDINALITY

Jak mierzyć to, co ważne — bez psucia monitoringu

MAPA 74/100

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Po co tworzyć własne metryki?

Standardowe metryki Actuatora są przydatne, ale nie wystarczą, aby mierzyć to, co naprawdę ważne dla Twojej aplikacji i biznesu.

Własne metryki pozwalają:

- mierzyć kluczowe procesy domenowe
- lepiej rozumieć zachowanie użytkowników
- wykrywać problemy szybciej
- podejmować decyzje oparte na danych
- łączyć perspektywę techniczną i biznesową

Mierz to, co pomaga w działaniu, a nie wszystko, co możliwe.

2 Czym jest cardinality?

Cardinality (liczba unikalnych wartości) to liczba różnych kombinacji tagów dla danej metryki.

Przykład:

Metryka http.requests z tagami:
method=[GET,POST] (2)
status=[200,500] (2)
uri=[/api/users,/api/orders] (2)

Daje 2 × 2 × 2 = **8** unikalnych serii.

Im więcej unikalnych wartości tagów, tym większa cardinality i więcej danych do przechowywania w systemie metryk.

3 Dlaczego wysoka cardinality jest problemem?

- Duże zużycie pamięci i miejsca na dysku w systemie monitoringu (np. Prometheus, Grafana).
- Wolniejsze zapytania i gorsza wydajność.
- Wyższe koszty przechowywania danych (w chmurze i on-premise).
- Trudniejsza analiza i większy szum w metrykach.
- Może doprowadzić do utraty danych lub awarii systemu monitoringu.

Wysoka cardinality często wynika z tagów o dużej liczbie unikalnych wartości (np. userId, sessionId, IP, dynamiczne ścieżki URL, wyjątki, pełne nazwy plików).

4 Dobre i złe przykłady tagów

Dobre tagi (niska cardinality)	Złe tagi (wysoka cardinality)
method = GET/POST	userId = 12345
status = 200/500	sessionId = abc-123...
outcome = SUCCESS/ERROR	ip = 192.168.1.10
region = eu-central-1	uri = /api/users/123
service = payment	exception = pełny komunikat
type = internal/external	file = /tmp/upload_abc.pdf

5 Tworzenie własnych liczników i timerów

Licznik (Counter) – zlicza zdarzenia

```
private final Counter orderCounter;

public PaymentService(MeterRegistry registry) {
  this.orderCounter = Counter.builder("orders_total")
    .description("Liczba złożonych zamówień")
    .tag("servioe", "payment")
    .register(registry);
}
orderCounter.increment();
```

Timer – mierzy czas trwania operacji

```
Timer timer = Timer.builder("order.processing.time")
    .description("Czas przetwarzania zamówienia")
    .tag("service", "payment")
    .register(registry);

return timer.record(() -> processOrder(order));
```

6 Metryki biznesowe vs techniczne

Metryki biznesowe

- Mówią o wartości dla biznesu (np. liczba zamówień, rejestracji).
- Pomagają mierzyć cele i efektywność produktu.
- Są zrozumiałe dla interesariuszy biznesowych.

Przykłady:

- liczba złożonych zamówień
- liczba aktywnych użytkowników
- wartość koszyka
- liczba rejestracji

Metryki techniczne

- Opisują działanie systemu (np. czas odpowiedzi, błędy).
- Pomagają w diagnozowaniu problemów.
- Są kluczowe dla utrzymania stabilności aplikacji.

Przykłady:

- czas obsługi żądania
- liczba błędów 5xx
- wykorzystanie zasobów
- długość kolejek
- dostępność endpointów

7 Nazewnictwo metryk

Stosuj spójną, zrozumiałą i hierarchiczną konwencję nazewnictwa.

Zalecenia:

- używaj małych liter i kropek (np. orders.total)
- grupuj metryki wg domeny lub funkcji
- stosuj opis (description) i tagi zamiast wielu nazw
- unikaj specjalnych znaków i spacji
- nazwy powinny być stabilne i zrozumiałe

Unikaj:

- zbyt ogólnych nazw (np. counter1)
- nazw związanych z implementacją
- zmiennych nazw (np. z ID użytkownika)

8 Przykłady w Micrometer

Micrometer ułatwia tworzenie własnych metryk. Działa z Prometheus, Grafana, InfluxDB i innymi systemami.

```
@Service
public class UserService {

   private final MeterRegistry registry;

   public UserService(MeterRegistry registry) {
      this.registry = registry;
   }
   public void userRegistered(String source) {
      Counter.builder("users.registreed")
         .description("Liczba rejestracji")
         .tag("source", source)
         .register(registry)
         .increment();
   }
```

9 Najczęstsze błędy

- Używanie tagów o wysokiej cardinality (np. userId, sessionId, dynamiczne URI).
- Tworzenie zbyt wielu metryk bez potrzeby.
- Brak opisu (description) i dokumentacji.
- Mieszanie metryk biznesowych i technicznych.
- Ignorowanie kosztów przechowywania danych.
- Zmiana nazw metryk (utrata historii).
- Brak przeglądu i czyszczenia nieużywanych metryk.
- Monitorowanie wszystkiego zamiast tego, co ważne.

10 W PRAKTYCE

- Zidentyfikuj kluczowe procesy do mierzenia.
- Zaprojektuj metryki i tagi (uwzględniając cardinality).
- Dodaj metryki do kodu (Micrometer).
- Zweryfikuj dane w Prometheus / Grafana.
- Regularnie przeglądaj i usuwaj zbędne metryki.
- Współpracuj z zespołem (Dev, QA, Biznes).

11 Dobre praktyki

- Mierz to, co naprawdę ważne.
- Utrzymuj niską cardinality.
- Używaj spójnego nazewnictwa.
- Dodawaj opisy i sensowne tagi.
- Regularnie przeglądaj metryki.
- Łącz metryki z alertami i dashboardami.
- Myśl o kosztach przechowywania danych.
- Dokumentuj metryki (np. w README).

12 Szybkie podsumowanie

- Własne metryki pozwalają mierzyć to, co ważne.
- Cardinality to liczba unikalnych kombinacji tagów.
- Wysoka cardinality powoduje problemy i koszty.
- Używaj prostych, stabilnych tagów.
- Twórz liczniki i timery z Micrometer.
- Rozróżniaj metryki biznesowe i techniczne.
- Stosuj spójne nazewnictwo i dobre praktyki.
- Unikaj najczęstszych błędów.
- Regularnie przeglądaj i optymalizuj metryki.

Mierz to, co ważne, nie wszystko!

Małe tagi
Wielka dostrzegalność!

SPRING BOOT

Dobre metryki
= lepsze decyzje
= lepsze aplikacje!

MONITORING
METRYKI
DIAGNOSTYKA
OBSERVABILITY
NIEZAWODNOŚĆ
ROZWÓJ

100 MAP MYŚLI ZE SPRING BOOTA

MAPA

MAPA 75/100

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

TRZY FILARY OBSERVABILITY

Logi, metryki i ślady rozproszone

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

1 Czym jest observability?

Observability to zdolność do zrozumienia, co dzieje się w systemie na podstawie danych, które sam system generuje: logów, metryk i śladów rozproszonych.

- Pozwala szybko wykrywać i diagnozować problemy.
- Daje wgląd w działanie aplikacji w produkcji.
- Ułatwia analizę wydajności, dostępności i zachowania użytkowników.
- Jest szczególnie ważna w systemach rozproszonych i mikroserwisach.

2 Logi – do czego służą?

Logi to szczegółowe, tekstowe informacje o zdarzeniach i działaniach w aplikacji.

Dzięki logom możemy:

- poznać co się wydarzyło (krok po kroku), na poziomie szczegółów,
- diagnozować błędy i wyjątki,
- śledzić przepływ żądań,
- analizować zachowanie użytkowników,
- mieć historyczny zapis zdarzeń.

Logi odpowiadają na pytanie: **„CO się wydarzyło?"**

3 Metryki – do czego służą?

Metryki to numeryczne dane o stanie i działaniu aplikacji, zbierane w regularnych odstępach czasu.

Dzięki metrykom możemy:

- monitorować wydajność (np. czas odpowiedzi, liczba żądań),
- śledzić wykorzystanie zasobów (CPU, pamięć),
- ustawiać alerty i progi,
- analizować trendy i anomalie,
- szybko wykrywać problemy z dostępnością.

Metryki odpowiadają na pytanie: **„JAK JEST teraz?"**

4 Tracing – do czego służy?

Tracing (śledzenie rozproszone) pozwala zobaczyć pełną ścieżkę żądania w systemach rozproszonych.

Dzięki tracingowi możemy:

- śledzić przepływ żądania przez wiele serwisów,
- mierzyć czas trwania poszczególnych kroków,
- zidentyfikować wąskie gardła,
- zobaczyć zależności między serwisami,
- łatwiej diagnozować problemy w mikroserwisach.

Tracing odpowiadają na pytanie: **„GDZIE i DLACZEGO to trwa tak długo?"**

5 Jak filary się uzupełniają?

Logi, metryki i ślady rozproszone dają różne perspektywy na ten sam system. Razem tworzą pełny obraz i pozwalają szybciej znaleźć przyczynę problemu.

- Metryki pokazują, że jest problem.
- Tracing pokazuje, gdzie w systemie występuje problem.
- Logi pokazują, co dokładnie się stało.

Najlepsze efekty daje ich wspólne wykorzystanie!

TRZY FILARY OBSERVABILITY



6 Przykład incydentu i jak każdy filar pomaga

Sytuacja: Użytkownicy zgłaszają, że logowanie do aplikacji działa bardzo wolno, a czasami kończy się błędem 500.

Metryki:
- pokazują wzrost czasu odpowiedzi na endpoint logowania,
- widać wzrost liczby błędów 500,
- alert informuje o przekroczeniu progu.

Tracing:
- pokazuje, że największe opóźnienie jest w serwisie użytkowników,
- widać, który dokładnie krok (np. zapytanie do bazy) trwa najdłużej.

Logi:
- zawierają szczegółowy błąd (np. timeout do bazy danych),
- pokazują komunikaty wyjątków i dane kontekstowe,
- pozwalają znaleźć dokładną przyczynę (np. przeciążenie bazy).

Dzięki połączeniu wszystkich trzech filarów szybko znajdujemy przyczynę i rozwiązujemy problem!

7 Correlation IDs

Correlation ID to unikalny identyfikator żądania, który pozwala powiązać logi, metryki i ślady z tym samym żądaniem w wielu serwisach.

- Ułatwia śledzenie przepływu żądania.
- Jest automatycznie propagowany w nagłówkach (np. traceparent).
- Pojawia się w logach, metrykach i tracingu.
- Jest standardem w systemach rozproszonych.

Jeden ID – pełny obraz żądania w całym systemie!

8 Narzędzia typowe

Popularne narzędzia ekosystemu Spring i obserwowalności:

- **Micrometer** — Zbieranie metryk w Spring Boot.
- **Prometheus** — Przechowywanie i analiza metryk.
- **Grafana** — Wizualizacją metryk, logów i tracingu.
- **OpenTelemetry** — Standard zbierania logów, metryk i śladów rozproszonych.

9 Dobre praktyki

- Instrumentuj aplikację od początku (Micrometer, OpenTelemetry).
- Używaj spójnych formatów logów (np. JSON).
- Dodawaj i propaguj Correlation ID.
- Definiuj sensowne metryki i alerty.
- Zadbaj o meaningful nazwy spanów i tagi.
- Ustaw odpowiednie poziomy logowania.
- Testuj i weryfikuj obserwowalność w środowisku zbliżonym do produkcji.

10 Najczęstsze pułapki

- Tylko logi albo tylko metryki (brak pełnego obrazu).
- Brak Correlation ID.
- Zbyt mało kontekstu w logach.
- Zbyt wiele nieistotnych logów (szum informacyjny).
- Brak alertów lub źle ustawione progi.
- Ignorowanie śladów rozproszonych w systemach mikroserwisowych.
- Niezgodne formaty i brak standaryzacji danych.

11 W PRAKTYCE

Zespoły wykorzystują observability do:

- Szybkiego wykrywania i diagnozowania incydentów.
- Monitorowania SLA i SLO.
- Analizy wydajności i optymalizacji aplikacji.
- Lepszego zrozumienia zachowania użytkowników.
- Budowania bardziej niezawodnych i odpornych systemów.
- Wsparcia kultury DevOps i ciągłego doskonalenia.

12 SZYBKIE PODSUMOWANIE

- Observability = logi + metryki + tracing.
- Logi: co się wydarzyło?
- Metryki: jak jest teraz?
- Tracing: gdzie i dlaczego?
- Razem dają pełny obraz systemu.
- Używaj Correlation ID.
- Wykorzystuj narzędzia: Micrometer, Prometheus, Grafana, OpenTelemetry.
- Stosuj dobre praktyki i unikaj pułapek.



Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

DISTRIBUTED TRACING I OTLP

Śledzenie żądań w systemach rozproszonych

MAPA 76/100

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Czym jest trace i span?

Trace to pełna ścieżka jednego żądania w systemie rozproszonym – od momentu wejścia do systemu aż do uzyskania odpowiedzi.

Span to pojedynczy krok (operacja) w ramach trace, np. wywołanie innej usługi, zapytanie do bazy danych lub przetwarzanie logiki.

Trace składa się z wielu powiązanych spanów tworzących drzewo zależności.

Tracing pozwala zobaczyć, co dzieje się w całym systemie – a nie tylko w jednej aplikacji.

2 Trace ID i Span ID

Trace ID – unikalny identyfikator całego żądania. Ten sam we wszystkich spanach danego żądania.

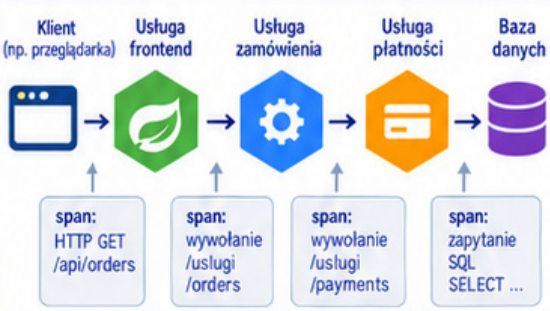
```
traceId: 4f9e7c2d8a4b1f3e9c0d6a7b8e1f2345
```

Span ID – unikalny identyfikator pojedynczego spana (kroku). Każdy span ma inny Span ID. ale należy do tego samego Trace ID.

```
spanId: 9c0d6a7b8e1f2345
```

Dzięki tym identyfikatorom możemy powiązać wszystkie logi, metryki i zdarzenia z jednym żądaniem.

3 Jak żądanie przechodzi przez kilka usług



Jedno żądanie tworzy jeden **trace**, który przechodzi przez wiele usług, a każda usługa dodaje swoje spany do tego samego trace ID.

4 Propagacja kontekstu

Kontekst trace (Trace ID, Span ID i inne dane) jest przekazywany między usługami w nagłówkach HTTP.

```
traceparent: 00-4f9e7c2d8a4b1f3e-
9c0d6a7b8e1f2345-9c0d6a7b8e1f2345-01
tracestate: vendor=xyz
```

Dzięki propagacji wszystkie usługi wiedzą, że obsługują to samo żądanie.

Propagacja działa również dla innych protokołów: gRPC, messaging (np. Kafka), R2DBC itp.

5 Czym jest OTLP

OTLP (OpenTelemetry Protocol) to otwarty, standardowy protokół do przesyłania śladów (traces), metryk i logów z aplikacji do systemów obserwowalności.

OTLP wykorzystuje **gRPC** lub **HTTP** i jest domyślnym formatem danych w ekosystemie OpenTelemetry.

- Uproszcza integrację z narzędziami.
- Wspiera wiele backendów (np. Jaeger, Grafana Tempo, SigNoz, Elastic).
- Pozwala wysyłać dane w jednym spójnym formacie.

6 OpenTelemetry i eksport śladów

OpenTelemetry

OpenTelemetry to otwarty projekt dostarczający biblioteki, SDK i narzędzia do zbierania, przetwarzania i eksportowania śladów, metryk i logów.

- Automatyczne instrumentacje dla popularnych bibliotek (np. Spring Boot, HTTP, bazy danych).
- Eksport śladów przez OTLP do wybranego backendu.
- Minimalna konfiguracja, bogate możliwości.
- W Spring Boot można użyć startera OpenTelemetry i skonfigurować eksport do Jaeger, Tempo lub innego systemu.

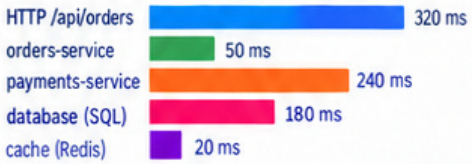
7 Sampling (próbkowanie)

Sampling to mechanizm ograniczania liczby zbieranych śladów, aby zmniejszyć koszty i narzut na system.

- Można próbkować losowo (np. 1% żądań).
- Można stosować reguły (np. zawsze próbkuj błędy lub wolne żądania).
- Istnieje sampling po stronie klienta (w aplikacji) i po stronie serwera (w backendzie).
- Dobrze dobrany sampling pozwala zachować ważne dane bez przeciążania systemu.

8 Jak tracing pomaga diagnozować opóźnienia

Tracing pokazuje dokładny czas trwania każdego spana, dzięki czemu łatwo zidentyfikować, gdzie powstają opóźnienia.



Dzięki temu szybko widzimy, która usługa lub operacja jest wąskim gardłem.

9 Dobre praktyki

- Dodawaj sensowne nazwy spanów (np. HTTP, DB, nazwa operacji).
- Wzbogacaj spany o atrybuty (np. userId, orderId, endpoint).
- Propaguj kontekst we wszystkich kanałach komunikacji (HTTP, gRPC, messaging).
- Stosuj sampling, ale zawsze zapisuj ślady błędów.
- Łącz tracing z logami i metrykami (korelacja po Trace ID).
- Używaj standardu OTLP i OpenTelemetry.
- Monitoruj i analizuj najważniejsze ścieżki w systemie.

10 Najczęstsze pułapki

- Brak propagacji kontekstu między usługami.
- Nieprawidłowe nazwy spanów i brak atrybutów.
- Zbyt duży lub zbyt mały sampling.
- Brak instrumentacji ważnych komponentów (np. baz danych, kolejek).
- Brak korelacji logów z trace ID.
- Ignorowanie kosztów (zbyt duża ilość danych).
- Traktowanie tracingu jako zbędnego dodatku.

11 W PRAKTYCE

Przykład powiązanych logów i trace'a

```
# Logi z aplikacji (Spring Boot)
2024-06-01T10:15:23.123 INFO [http-nio-8080]
[traceId=4f9e7c2d8a4b1f3e9c0d6a7b8e1f2345]
[spanId=9c0d6a7b8e1f2345] o.s.web: Przyjęto żądanie
GET /api/orders

2024-06-01T10:15:33.456 INFO [http-nio-8080]
[traceId=4f9e7c2d8a4b1f3e9c0d6a7b8e1f2345]
[spanId=3b2c1d0e9f8a7b6c] c.e.payment: Wywołanie
usługi płatności zakończone sukcesem (240 ms)
```

Ten sam Trace ID w logach, trace'ach i metrykach pozwala łatwo powiązać wszystkie dane.

12 Szybkie podsumowanie

- Trace to pełna ścieżka żądania, span to pojedynczy krok.
- Trace ID i Span ID pozwalają powiązać wszystkie dane.
- Kontekst propagowany jest w nagłówkach (np. traceparent).
- OTLP to standardowy protokół przesyłania śladów, metryk i logów.
- OpenTelemetry ułatwia instrumentację i eksport.
- Sampling ogranicza liczbę zbieranych śladów.
- Tracing pomaga szybko diagnozować opóźnienia.
- Stosuj dobre praktyki i unikaj najczęstszych pułapek.
- W praktyce łącz tracing z logami i metrykami.
- Distributed tracing to kluczowy element obserwowalności w systemach rozproszonych.

Monitoruj
Diagnozuj
Reaguj szybciej
Buduj lepsze aplikacje!

SPRING BOOT

Obserwuj
to, co ważne
Twórz stabilne
systemy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

MAPA 77/100

LOG CORRELATION I STRUCTURED LOGGING

Czytelne logi, które naprawdę pomagają

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Po co strukturalne logowanie?

- Łatwiejsze wyszukiwanie i filtrowanie logów (np. po userId, traceId, statusie).
- Lepsza analiza i metryki (agregacje, trendy, alerty).
- Czytelne i spójne logi niezależnie od środowiska.
- Szybsze rozwiązywanie problemów i mniejszy czas przestoju.
- Logi stają się danymi, które można automatycznie przetwarzać.

2 Log correlation – czym jest?

Log correlation to łączenie powiązanych logów z różnych komponentów i usług w jedną historię żądania (np. HTTP). Dzięki wspólnemu identyfikatorowi (trace ID / request ID) możemy śledzić przepływ żądania przez całą architekturę – od bramy API, przez mikroserwisy, aż po bazy danych i kolejki.

Efekt: widzimy pełny kontekst, a nie pojedyncze, odizolowane wpisy logów.

3 Trace ID / Request ID w logach

- **Trace ID** – identyfikator całego śladu (ang. trace) w systemie rozproszonym.
- **Request ID** – identyfikator pojedynczego żądania (często taki sam jak trace ID w prostszych systemach).
- Dodawany automatycznie przez Spring Boot (np. z Micrometer Tracing).
- Przekazywany między usługami w nagłówkach (np. traceparent, X-Request-ID).
- Powinien być obecny w każdym wpisie loga.

4 Format JSON vs tekstowy

Cecha	JSON (strukturalny)	Tekstowy (tradycyjny)
Czytelność dla ludzi	Dobra	Dobra
Łatwość analizy maszynowej	Świetna	Trudna
Filtrowanie i agregacje	Bardzo łatwe	Ograniczone (regex)
Rozmiar	Nieco większy	Mniejszy
Standard w observability	Tak (ELK, Loki, itp.)	Rzadziej
Rekomendacja	✅ **Zalecany**	Tylko do prostych przypadków

5 Poziomy logowania

TRACE	Najbardziej szczegółowe informacje (np. przepływ metody).
DEBUG	Informacje pomocne przy debugowaniu.
INFO	Normalne działanie aplikacji (ważne zdarzenia biznesowe).
WARN	Sytuacje niepokojące, ale aplikacja działa dalej.
ERROR	Błędy, które wymagają uwagi.
FATAL	Krytyczne błędy (rzadko używane).

6 Przykładowe pola loga

Pole	Opis
timestamp	Czas zdarzenia (ISO 8601)
level	Poziom logowania (INFO, ERROR, ...)
traceId	Identyfikator całego śladu
requestId	Identyfikator żądania
service	Nazwa aplikacji / serwisu
version	Wersja aplikacji
environment	Środowisko (dev, test, prod)
userId	Identyfikator użytkownika (jeśli dotyczy)
message	Czytelny komunikat
details	Dodatkowe dane kontekstowe (np. błędy)

7 Jak szukać problemów po korelacji

1. Znajdź traceId lub requestId z loga błędu.
2. Wyszukaj wszystkie logi z tym samym identyfikatorem w systemie (np. Grafana Loki, ELK, Datadog).
3. Przeanalizuj kolejność zdarzeń w czasie – zobacz, w którym miejscu wystąpił problem.
4. Sprawdź powiązane metryki, trasy i span'y (observability).
5. Znajdź przyczynę (np. błąd walidacji, timeout, błąd zewnętrznego serwisu).
6. Podejmij działania i zweryfikuj poprawkę.

8 Przykładowy log w formacie JSON

```json
{
  "timestamp": "2024-05-10T10:15:30.123Z",
  "level": "INFO",
  "traceId": "0af7651916cd43dd8448eb211c80319c",
  "requestId": "b7e3f2a9c1d4",
  "service": "moja-aplikacja",
  "version": "1.0.0",
  "environment": "prod",
  "userId": "u-12345",
  "message": "Zrealizowano zamówienie",
  "orderId": "o-98765",
  "status": "SUCCESS",
  "durationMs": 152
}
```

9 Integracja z observability stack

- Wysyłaj logi do centralnego systemu (np. Grafana Loki, ELK/Opensearch, Datadog).
- Używaj Micrometer Tracing + OpenTelemetry do propagacji trace ID.
- Koreluj logi z metrykami (Prometheus, Grafana) i śladami (Jaeger, Tempo).
- Skonfiguruj eksport logów (np. via Logback encoder, Fluent Bit, Agent).
- Używaj tych samych identyfikatorów (traceId) we wszystkich narzędziach.

10 Dobre praktyki

- ✅ Loguj w formacie JSON.
- ✅ Zawsze dodawaj traceId / requestId.
- ✅ Używaj sensownych poziomów logowania.
- ✅ Loguj kontekst (np. userId, orderId, operation).
- ✅ Nie loguj danych wrażliwych (hasła, tokeny, PESEL).
- ✅ Ustal spójny format logów w całym zespole.
- ✅ Rotuj i archiwizuj logi.
- ✅ Testuj logowanie (np. w testach integracyjnych).

11 Najczęstsze pułapki

- ❌ Logowanie danych wrażliwych (np. haseł, tokenów).
- ❌ Brak traceId / requestId w logach.
- ❌ Zbyt ogólne lub nieczytelne komunikaty.
- ❌ Nadmierna ilość logów (szum).
- ❌ Tylko logi tekstowe bez struktury.
- ❌ Różne formaty logów w różnych serwisach.
- ❌ Brak korelacji z metrykami i śladami.
- ❌ Ignorowanie logów WARN (ważne sygnały!).

12 W PRAKTYCE

- Dodaj zależności: spring-boot-starter-actuator i Micrometer Tracing (np. z OpenTelemetry).
- Skonfiguruj JSON logowanie (np. logstash-logback).
- Upewnij się, że traceId i requestId są w każdym logu.
- Wyślij logi do wybranego systemu (Loki / ELK).
- Odtwórz scenariusz błędu i znajdź logi po traceId.
- Skoreluj logi z metrykami i śladami w Grafanie.
- Zweryfikuj działanie w środowisku produkcyjnym.
- Dodaj test, który sprawdza obecność traceId w logach.

13 SZYBKIE PODSUMOWANIE

- ✅ Strukturalne logi = lepsza widoczność i szybsze diagnozowanie.
- ✅ Korelacja po traceId łączy logi z całego systemu.
- ✅ JSON to standard w nowoczesnej observability.
- ✅ Używaj odpowiednich poziomów logowania.
- ✅ Integruj logi z metrykami i śladami.
- ✅ Stosuj dobre praktyki i unikaj pułapek.
- ✅ Czytelne logi to podstawa niezawodnych aplikacji.

Czytelne logi
to krótszy czas awarii
i szczęśliwszy zespół!
Buduj lepsze aplikacje!



Logi łączą kropki między usługami!
Obserwuj, Diagnozuj, Rozwijaj!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

MAPA 78/100

GRACEFUL SHUTDOWN I APPLICATION AVAILABILITY

Bezpieczne zatrzymywanie aplikacji i gotowość usług

Pisz testy Buduj pewniej Rozwijaj się w Spring Bootcie!

Lepszy monitoring Lepsza widoczność Lepsze aplikacje!

1 Czym jest graceful shutdown?

Graceful shutdown to kontrolowane i uporządkowane zatrzymanie aplikacji, w którym aplikacja kończy obsługę bieżących żądań, odrzuca nowe i bezpiecznie zwalnia zasoby.

- Aplikacja kończy rozpoczęte operacje (np. HTTP, komunikaty).
- Zamyka połączenia, wątki i zasoby.
- Trwa określony czas (timeout).

Dzięki temu nie tracimy danych, nie przerywamy operacji i zwiększamy dostępność usług.

2 Dlaczego jest ważny?

- Zapobiega utracie danych i niespójnym stanom.
- Pozwala dokończyć trwające żądania i operacje.
- Umożliwia łagodne wyłączenie instancji podczas wdrożeń.
- Wspiera wysoką dostępność i ciągłość działania.
- Zmniejsza liczbę błędów 5xx podczas restartów.
- Jest niezbędny w środowiskach produkcyjnych (Kubernetes, load balancer, autoskalowanie).

3 ApplicationAvailability

Spring Boot udostępnia mechanizm **ApplicationAvailability**, który monitoruje stan gotowości i żywotności aplikacji.

Dzięki niemu Actuator udostępnia endpoiny: **/actuator/health, /actuator/health/liveness** i **/actuator/health/readiness**, a my możemy reagować na zmiany stanu w kodzie.

- Automatycznie zarządza stanami podczas startu i zamykania.
- Umożliwia integrację z Kubernetes i innymi platformami.
- Można nasłuchiwać zdarzeń zmiany stanu (AvailabilityChangeEvent).

4 LivenessState i ReadinessState

LivenessState (żywotność)

- Określa, czy aplikacja działa.
- Jeśli jest nieprawidłowy (BROKEN), instancja powinna zostać rzestartowana.
- Chroni przed zawieszeniem się aplikacji.

ReadinessState (gotowość)

- Określa, czy aplikacja jest gotowa do obsługi ruchu.
- Podczas zamykania zmienia się na REFUSING_TRAFFIC.
- Gdy nie jest gotowa, load balancer nie kieruje do niej żądań.

5 Co dzieje się podczas zamykania?

1. Otrzymanie sygnału zamknięcia (np. SIGTERM).
2. Ustawienie ReadinessState na REFUSING_TRAFFIC (odrzucanie nowych żądań).
3. Load balancer przestaje kierować ruch do tej instancji.
4. Aplikacja kończy obsługę bieżących żądań i operacji (np. z bazy, kolejek).
5. Zamykane są serwery i zasoby (np. wątki, połączenia, schedulery).
6. Po upływie czasu (timeout) aplikacja zatrzymuje się całkowicie.

6 Jak ustawić graceful shutdown w Spring Boot?

W Spring Boot wystarczy włączyć graceful shutdown i opcjonalnie dostosować czasy.

application.yml:

```yaml
server:
  shutdown: graceful
spring:
  lifecycle:
    timeout-per-shutdown-phase: 30s
```

- Spring Boot ustawia ReadinessState i odrzuca nowe żądania.
- Aplikacja ma czas na dokończenie bieżących operacji.
- Można dostosować timeout do potrzeb aplikacji.

7 Wpływ na Kubernetes / load balancer

- Po ustawieniu ReadinessState na REFUSING_TRAFFIC, endpoint readiness zwraca niegotowość (HTTP 503).
- Kubernetes usuwa pod z puli Service (Endpointy).
- Load balancer przestaje kierować nowe żądania do tej instancji.
- Istniejące połączenia są kończone zgodnie z timeoutami.
- Dzięki temu wdrożenia (rolling update) i skalowanie są bezpieczne i bez przestojów.

8 Najczęstsze pułapki

- Brak włączonego graceful shutdown.
- Zbyt krótki timeout na zamknięcie aplikacji.
- Długie operacje blokujące zamknięcie (np. niekończące się taski, połączenia).
- Niewłaściwa konfiguracja probe'ów w Kubernetes.
- Brak obsługi przerwania w niestandardowych komponentach.
- Odrzucanie żądań za wcześnie (lub zbyt późno).
- Ignorowanie zdarzeń zamykania własnych zasobów (kolejki, schedulery, zewnętrzne klienty).

9 Dobre praktyki

- Zawsze włączaj graceful shutdown.
- Ustaw realistyczny timeout (np. 30–60 sekund).
- Obsługuj zdarzenia *AvailabilityChangeEvent*.
- Upewnij się, że wszystkie zasoby zamykają się poprawnie.
- Testuj proces zamykania w środowisku zbliżonym do produkcji.
- Monitoruj metryki i logi podczas wdrożeń.
- Dbaj o krótkie i idempotentne operacje.

10 W PRAKTYCE

Zespoły wykorzystują graceful shutdown do:

- Bezpiecznych wdrożeń (rolling update).
- Skalowania w górę i w dół.
- Zarządzania obciążeniem w klastrze.
- Unikania błędów 5xx podczas restartów.
- Kończenia operacji biznesowych (np. płatności, zamówienia).
- Obsługi długich zapytań i procesów w tle.
- Integracji z kolejkami (np. Kafka, RabbitMQ).
- Zwiększenia niezawodności i dostępności usług.

11 Przykład application.yml

```yaml
# Konfiguracja graceful shutdown
server:
  shutdown: graceful

# Czas na zakończenie operacji
spring:
  lifecycle:
    timeout-per-shutdown-phase: 30s

management:
  endpoints:
    web:
      exposure:
        include: health,info
  endpoint:
    health:
      probes:
        enabled: true
```

12 Cykl życia aplikacji

Aplikacja działa (READY)

↓

Otrzymanie sygnału zamknięcia (SIGTERM)

↓

ReadinessState = REFUSING_TRAFFIC (odrzucanie nowych żądań)

↓

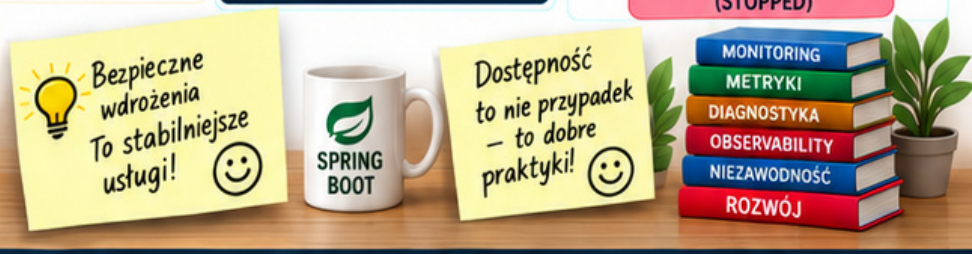
Kończenie bieżących operacji i zamykanie zasobów

↓

Aplikacja zatrzymana (STOPPED)

13 SZYBKIE PODSUMOWANIE

- Graceful shutdown = bezpieczne i uporządkowane zatrzymanie aplikacji.
- Używa mechanizmu ApplicationAvailability.
- Wykorzystuje stany LivenessState i ReadinessState.
- Pozwala dokończyć bieżące operacje i odrzuca nowe żądania.
- Jest kluczowy dla Kubernetes i load balancerów.
- Włączaj graceful shutdown i dostosuj timeouty.
- Stosuj dobre praktyki i testuj proces zamykania.
- Zwiększa niezawodność, dostępność i stabilność aplikacji.



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

MAPA 79/100

BEZPIECZNY ACTUATOR W PRODUKCJI

Jak monitorować i nie ujawniać za dużo

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Dlaczego zabezpieczenie Actuatora jest ważne?

Actuator udostępnia wrażliwe informacje o aplikacji, środowisku i infrastrukturze.

Wycieki danych mogą pomóc atakującym w poznaniu architektury, zależności i wersji komponentów.

Niezabezpieczony Actuator może umożliwić wykonanie nieautoryzowanych działań (np. wyłączenie aplikacji).

Zabezpieczenie Actuatora to ochrona danych, dostępności i ciągłości działania systemu produkcyjnego.

2 Które endpointy są wrażliwe?

Szczególnie wrażliwe endpointy (ujawniają dużo informacji lub pozwalają na działania administracyjne):

`/actuator/env`	zmienne środowiskowe (np. hasła, klucze)
`/actuator/configprops`	konfiguracja aplikacji
`/actuator/beans`	lista beanów i zależności
`/actuator/mappings`	mapowanie endpointów
`/actuator/health`	szczegółowe informacje o stanie systemu
`/actuator/loggers`	zmiana poziomu logowania
`/actuator/shutdown`	zatrzymanie aplikacji (jeśli włączony)
`/actuator/threaddump`	zrzut wątków i stosów

3 Ekspozycja tylko wybranych endpointów

Udostępniaj wyłącznie te endpointy, które są naprawdę potrzebne w produkcji.

Przykład minimalnego zestawu:

- `/actuator/health` – do sprawdzania stanu
- `/actuator/info` – podstawowe informacje
- `/actuator/metrics` – metryki dla systemu monitoringu

Nie udostępniaj publicznie:

- `/env`, `/configprops`, `/beans`, `/mappings`
- `/loggers`, `/threaddump`, `/shutdown`

Zasada: mniej znaczy bezpieczniej – udostępniaj tylko to, co konieczne.

4 Spring Security i autoryzacja

Zabezpiecz endpointy Actuatora za pomocą Spring Security.

- Wymagaj uwierzytelnienia (np. Basic Auth, OAuth2, JWT).
- Ogranicz dostęp za pomocą ról i autoryzacji.
- Możesz stosować osobną konfigurację bezpieczeństwa dla endpointów Actuatora.
- Używaj HTTPS, aby chronić dane w transmisji.

```
http.securityMatcher("/actuator/***)
  .authorizeHttpRequests(...)
```

5 Osobny port management server

Uruchom Actuator na osobnym porcie, aby odseparować go od aplikacji biznesowej.

- Zmniejsza ryzyko nieautoryzowanego dostępu z zewnątrz.
- Ułatwia konfigurację firewall i reguł sieciowych.
- Możliwość nasłuchiwania tylko na adresie localhost lub wewnętrznym interfejsie.

Przykład:

```
server.port=8080              # aplikacja
management.server.port=8081   # Actuator
```

6 Sieć wewnętrzna, VPN, firewall

Endpointy Actuatora powinny być dostępne tylko z zaufanej sieci.

- Ogranicz dostęp za pomocą firewall (np. tylko adresy IP z sieci firmowej).
- Używaj VPN do dostępu zdalnego przez zaufane połączenie.
- Nie wystawiaj Actuatora bezpośrednio do Internetu.
- Jeśli korzystasz z platformy chmurowej, stosuj security groups, private endpoints i load balancery z ograniczeniem dostępu.

7 Role użytkowników i minimalne uprawnienia

Stosuj zasadę najmniejszych uprawnień (ang. least privilege).

- Osobne role dla operacji odczytu i zapisu.
- Przykładowe role:
  - `ACTUATOR_READ` – dostęp tylko do odczytu (np. health, info, metrics)
  - `ACTUATOR_ADMIN` – dostęp do operacji administracyjnych (np. loggers, shutdown)
- Integruj z systemem tożsamości firmy (np. LDAP, OAuth2, Keycloak).

8 Przykładowa konfiguracja application.yml

YAML

```yaml
server:
  port: 8080
management:
  server:
    port: 8081
    address: 127.0.0.1   # tylko localhost
  endpoints:
    web:
      exposure:
        include: health,info,metrics
  endpoint:
    health:
      show-details: when-authorized
spring:
  security:
    user:
      name: actuator
      password: ${ACTUATOR_PASSWORD}
```

9 Najczęstsze pułapki

- Ujawnienie wrażliwych endpointów w środowisku produkcyjnym.
- Brak autoryzacji lub zbyt szerokie uprawnienia.
- Używanie HTTP zamiast HTTPS.
- Udostępnianie szczegółowych informacji w /health (np. dane o bazie).
- Wystawienie Actuatora do Internetu.
- Ignorowanie logów i prób nieautoryzowanego dostępu.

10 Dobre praktyki

- Udostępniaj tylko niezbędne endpointy.
- Zawsze używaj HTTPS.
- Stosuj autoryzację i role.
- Uruchamiaj Actuator na osobnym porcie.
- Ogranicz dostęp do zaufanej sieci.
- Regularnie przeglądaj konfigurację.
- Monitoruj próby nieautoryzowanego dostępu.
- Dokumentuj zasady dostępu dla zespołu.

11 W PRAKTYCE

W zespole warto:

- Zdefiniować minimalny zestaw endpointów dla środowisk (dev, stage, prod).
- Skonfigurować osobny port i ograniczenie IP.
- Zintegrować z firmowym systemem autoryzacji (np. OAuth2, Keycloak).
- Przetestować dostęp z zewnętrznej sieci – powinien być zablokowany.
- Regularnie audytować i weryfikować, czy nie pojawiły się nowe, niepotrzebne endpointy.

12 Szybkie podsumowanie

- Actuator jest potężnym narzędziem, ale wymaga dobrej konfiguracji bezpieczeństwa.
- Udostępniaj tylko wybrane endpointy.
- Stosuj Spring Security i autoryzację.
- Rozważ osobny port management server.
- Ogranicz dostęp do sieci wewnętrznej, VPN i firewall.
- Używaj ról i minimalnych uprawnień.
- Unikaj najczęstszych pułapek.
- Stosuj sprawdzone praktyki w zespole.
- Regularnie weryfikuj konfigurację i logi.
- Bezpieczny Actuator = stabilniejsza i bardziej odporna aplikacja produkcyjna.

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 08 — ACTUATOR, MONITORING I OBSERVABILITY

DIAGNOSTYKA JVM I APLIKACJI

Wątki, pamięć, GC, połączenia i problemy wydajnościowe

MAPA 80/100

Pisz testy
Buduj pewniej
Rozwijaj się
w Spring Bootcie!

Lepszy monitoring
Lepsza widoczność
Lepsze aplikacje!

1 Kiedy zaczynać diagnostykę?

- [x] Aplikacja działa wolno lub ma wysokie czasy odpowiedzi (latency).
- [x] Pojawiają się błędy 5xx, time-outy, spadek dostępności.
- [x] Wzrasta zużycie CPU, pamięci lub liczba wątków.
- [x] Częste wykonywanie Garbage Collection lub błędy OutOfMemoryError.
- [x] Wyciek połączeń do bazy danych lub zewnętrznych serwisów.
- [x] Użytkownicy zgłaszają problemy z działaniem.
- [x] Masz podejrzenie blokad (wątków, IO, DB).

Nie czekaj na awarię – diagnozuj już przy pierwszych niepokojących sygnałach!

2 Wątki i thread dump

Wątki (threads) wykonują kod w aplikacji. Ich blokowanie lub zbyt duża liczba wpływa na wydajność.

Jak pobrać thread dump?

```
# jcmd (zalecane)
jcmd <pid> Thread.print > threaddump.txt

# lub jstack
jstack <pid> > threaddump.txt
```

Na co zwrócić uwagę?

- [x] Wątki w stanie BLOCKED, WAITING, TIMED_WAITING.
- [x] Długie stack trace (gdzie wątek się zatrzymał).
- [x] Nadmierna liczba wątków.
- [x] Deadlocki (np. "Found one Java-level deadlock").
- [x] Wątki wykonujące wolne operacje (IO, DB, HTTP).

3 Pamięć JVM i heap

Pamięć JVM dzieli się na obszary, z których najważniejsza jest sterta (heap), gdzie znajdują się obiekty aplikacji.

Podstawowe metryki (z Actuator):

- jvm.memory.used – aktualne użycie pamięci
- jvm.memory.max – maksymalna pamięć
- jvm.memory.committed – zaalokowana pamięć

Analiza heap dump:

```
jmap -dump:live,format=b,file=heap.hprof <pid>
```

- [x] Analizuj za pomocą MAT (Eclipse Memory Analyzer).
- [x] Szukaj obiektów, które nie są zwalniane (memory leak).
- [x] Sprawdzaj dominujące obiekty i ścieżki referencji.

4 Garbage Collection

GC zwalnia nieużywane obiekty z pamięci. Zbyt częste lub długie pauzy GC powodują spadek wydajności aplikacji.

Co monitorować?

- [x] Częstotliwość i czas pauz GC.
- [x] Typ kolektora (np. G1, ZGC, Parallel).
- [x] Rozmiar generacji (young/old).
- [x] Wzrost czasu GC w czasie.

Przydatne opcje JVM:

```
-Xlog:gc*:file=gc.log:time,uptimemillis,level,tags
-XX:+UseG1GC
```

5 Połączenia HTTP i bazy danych

Połączenia HTTP (klient)

- [x] Sprawdzaj time-outy, ponowne próby, błędy 5xx.
- [x] Monitoruj liczbę aktywnych połączeń.
- [x] Używaj metryk z Actuator (http.client.*).
- [x] Weryfikuj opóźnienia zewnętrznych serwisów.

Połączenia do bazy danych

- [x] Sprawdzaj liczbę aktywnych/oczekujących połączeń (HikariCP: hikaricp.*).
- [x] Monitoruj wolne zapytania.
- [x] Upewnij się, że połączenia są zamykane.
- [x] Analizuj blokady w bazie (np. przez EXPLAIN).

6 CPU i latency

Wysokie użycie CPU lub duże czasy odpowiedzi oznaczają problemy z wydajnością.

Co sprawdzać?

- [x] Użycie CPU (system i wątki JVM).
- [x] Czasy odpowiedzi endpointów (HTTP, DB, zewnętrzne).
- [x] Długie operacje (profilowanie, flame graphy).
- [x] Blokady, pętle, kosztowne operacje (np. serializacja).
- [x] Metryki z Actuator: system.cpu.usage, process.cpu.usage, http.server.requests.

Wysokie CPU + dużo wątków w RUNNABLE = aplikacja intensywnie przetwarza dane.
Wysokie CPU + wątki w WAITING/BLOCKED = możliwe blokady.

7 Narzędzia diagnostyczne

- [x] **Actuator** – metryki, health, loggers, threaddump, heapdump.
- [x] **Metrics** – wizualizacja w Prometheus + Grafana.
- [x] **Logs** – analiza logów (ELK, Loki, Cloud).
- [x] **Thread dump** – jcmd, jstack, Actuator (/threaddump).
- [x] **Heap dump** – jmap, Actuator (/heapdump).
- [x] **JFR (Java Flight Recorder)** – profilowanie JVM.
- [x] **VisualVM** – podgląd wątków, pamięci, CPU.
- [x] **MAT** – analiza heap dumpów.
- [x] **Async Profiler / YourKit** – szczegółowe profilowanie.

8 Typowy flow diagnozowania incydentu

1. **Zidentyfikuj problem** (np. wolne odpowiedzi, błędy 5xx)
2. **Zbierz dane** (metryki, logi, threaddump, heapdump)
3. **Przeanalizuj** (wątki, pamięć, GC, połączenia, CPU)
4. **Znajdź przyczynę** (wąskie gardło, wyciek, blokady, konfiguracja)
5. **Wprowadź rozwiązanie i zweryfikuj poprawę**

Powtórz w razie potrzeby

9 Najczęstsze pułapki

- [x] Ignorowanie pierwszych symptomów.
- [x] Brak monitoringu (metryk, logów, tracingu).
- [x] Zbyt mały lub zbyt duży rozmiar heap.
- [x] Memory leak (niewłaściwe zarządzanie obiektami).
- [x] Wyciek połączeń (HTTP, baza danych).
- [x] Długie operacje w wątkach obsługujących żądania.
- [x] Brak time-outów i limitów.
- [x] Nieanalizowanie GC (zbyt częste pauzy).
- [x] Niewłaściwe logowanie (za mało lub za dużo).
- [x] Diagnoza tylko na podstawie jednego źródła danych.

10 Dobre praktyki

- [x] Włącz Actuator i monitoruj kluczowe metryki.
- [x] Ustal sensowne rozmiary heap (-Xms, -Xmx).
- [x] Używaj G1GC (domyślnie w nowszych JDK).
- [x] Stosuj time-outy i limity (HTTP, DB, zewnętrzne).
- [x] Regularnie analizuj logi i metryki.
- [x] Automatyzuj zbieranie thread/heap dumpów.
- [x] Używaj profilera w środowiskach testowych.
- [x] Reaguj na anomalie (alerty, dashboardy).
- [x] Dokumentuj przyczyny i rozwiązania.

11 W PRAKTYCE

- [x] Skonfiguruj Actuator (health, metrics, threaddump, heapdump, loggers).
- [x] Zintegruj metryki z Prometheus i Grafana.
- [x] Ustaw logowanie GC.
- [x] Przetestuj generowanie thread dump i heap dump.
- [x] Zasymuluj problem (np. wyciek pamięci, blokada).
- [x] Przeanalizuj dane (VisualVM, MAT).
- [x] Wdróż poprawki i porównaj metryki przed/po.
- [x] Dodaj alerty na kluczowe wskaźniki.

12 Objawy i możliwe przyczyny

Objaw	Możliwe przyczyny
Wysokie użycie CPU	Intensywne przetwarzanie, pętle, blokady, zła konfiguracja
Duże czasy odpowiedzi	Wolne zapytania DB, problemy z HTTP, przeciążenie, GC, brak zasobów
Częste GC / długie pauzy	Za mało pamięci, memory leak, zbyt duże obiekty
OutOfMemoryError	Wyciek pamięci, zbyt mały heap, za dużo wątków/połączeń
Brak dostępnych połączeń DB	Wyciek połączeń, zbyt mały pool, długie transakcje
Dużo wątków w WAITING	Oczekiwanie na IO, blokady, zewnętrzne serwisy, baza danych

13 SZYBKIE PODSUMOWANIE

- [x] Rozpoznawaj symptomy i zaczynaj diagnostykę wcześnie.
- [x] Analizuj wątki, pamięć, GC, połączenia, CPU i latency.
- [x] Korzystaj z Actuator, metryk, logów i narzędzi JVM.
- [x] Stosuj sprawdzony flow diagnozowania.
- [x] Unikaj najczęstszych pułapek.
- [x] Wdrażaj dobre praktyki i monitoruj efekty.
- [x] Lepsza diagnostyka = stabilniejsze i szybsze aplikacje!

Monitoruj
Diagnozuj
Rozwiązuj
Buduj lepsze aplikacje!

SPRING BOOT

Obserwuj to, co ważne
Twórz stabilne systemy!



Pisz testy
Buduj pewniej
Rozwijaj szybciej

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

SPRING MVC vs WEBFLUX

Jak wybrać właściwy model programowania?

MAPA 81/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest Spring MVC?

Spring MVC to klasyczny, oparty na Servletach stos do budowy aplikacji webowych w Springu.

- Model żądanie/odpowiedź (request/response)
- Styl programowania synchroniczny i imperatywny
- Działa w kontenerach servletowych (np. Tomcat, Jetty, Undertow)
- Najczęściej używany w tradycyjnych aplikacjach webowych i API

Java

```
@RestController
@RequestMapping("/mvc")
public class HelloMvcController {

  @GetMapping("/hello")
  public String hello() {
    return "Witaj z Spring MVC!";
  }
}
```

2 Czym jest WebFlux?

Spring WebFlux to reaktywny stos webowy, oparty na nieblokującym I/O, wykorzystujący projekt Reactor (Mono i Flux).

- Model nieblokujący (non-blocking)
- Programowanie reaktywne z użyciem Mono i Flux
- Działa na Netty (domyślnie) lub w kontenerach servletowych
- Idealny do aplikacji o wysokiej współbieżności, strumieniowania danych i komunikacji w czasie rzeczywistym

Java

```
@RestController
@RequestMapping("/flux")
public class HelloFluxController {

  @GetMapping("/hello")
  public Mono<String> hello() {
    return Mono.just("Witaj
    z WebFlux!");
  }
}
```

3 Porównanie architektoniczne

Cecha	Spring MVC	Spring WebFlux
Stos	Servlet (javax/jakarta)	Reaktywny (Reactor)
Model	Imperatywny	Reaktywny
I/O	Blokujące	Nieblokujące
Wątki	Wątek na żądanie	Event-loop (mało wątków)
Kontenery	Tomcat, Jetty, Undertow	Netty (domyślnie) lub servletowe
Programowanie	Klasyczne (imperatywne)	Reaktywne (Mono/Flux)

4 Kiedy wybrać Spring MVC?

- ✅ Klasyczne aplikacje CRUD
- ✅ Tradycyjne API REST
- ✅ Zespoły znające styl imperatywny
- ✅ Prostsze debugowanie i łatwiejszy start
- ✅ Wystarczająca wydajność dla większości aplikacji
- ✅ Integracje z istniejącymi, blokującymi bibliotekami

5 Kiedy wybrać WebFlux?

- ✅ Wysoka współbieżność (dużo równoczesnych połączeń)
- ✅ Strumieniowanie danych (streaming)
- ✅ SSE (Server-Sent Events) i WebSocket
- ✅ Reaktywne potoki danych (np. zewnętrzne API, bazy reaktywne)
- ✅ Aplikacje czasu rzeczywistego (real-time)
- ✅ Cały stos technologiczny jest reaktywny (np. Spring Data R2DBC, WebClient)

6 Zalety i ograniczenia Spring MVC

Zalety	Ograniczenia
✅ Prostota i dojrzałość	❌ Blokujące I/O
✅ Duża społeczność i ekosystem	❌ Mniejsze możliwości przy bardzo dużej współbieżności
✅ Łatwiejsze debugowanie	❌ Wątek na każde żądanie (zwiększone zużycie zasobów)
✅ Szerokie wsparcie bibliotek	❌ Ograniczenia przy strumieniowaniu danych
✅ Sprawdzone w produkcji	

7 Zalety i ograniczenia WebFlux

Zalety	Ograniczenia
✅ Nieblokujące I/O	❌ Większa złożoność
✅ Wysoka współbieżność	❌ Trudniejsze debugowanie
✅ Obsługa strumieniowania, SSE i WebSocket	❌ Wymaga reaktywnego ekosystemu (np. R2DBC)
✅ Niższe zużycie zasobów (przy dużej liczbie połączeń)	❌ Nie zawsze daje lepszą wydajność
✅ Nowoczesny, reaktywny model programowania	❌ Stromsza krzywa nauki dla zespołów

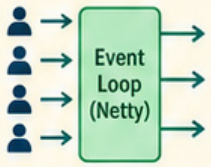
8 Model wątków

Spring MVC
Wątek na żądanie (serwletowy)



Każde żądanie obsługiwane jest przez osobny wątek. Podczas operacji blokujących wątek jest zablokowany.

WebFlux
Event-loop (nieblokujący)



Wiele żądań obsługiwanych jest przez niewielką liczbę wątków w modelu nieblokującym.

9 Wydajność — co naprawdę oznacza?

WebFlux nie jest automatycznie szybszy od Spring MVC!

- ✅ Zyski z wydajności pojawiają się przy dużej liczbie równoczesnych połączeń, operacjach I/O i nieblokujących integracjach.
- ✅ Jeśli używasz blokujących bibliotek (np. klasyczne JDBC), zyski będą niewielkie.
- ✅ Dla prostych aplikacji CRUD różnice w wydajności mogą być niezauważalne.
- ✅ Kluczowe jest end-to-end reaktywne podejście (cały stos technologiczny).

10 Pułapki

- ⚠️ Używanie WebFlux tylko dlatego, że jest „nowoczesny".
- ⚠️ Mieszanie blokującego kodu w reaktywnym przepływie (np. Thread.sleep(), JDBC).
- ⚠️ Oczekiwanie natychmiastowych wzrostów wydajności.
- ⚠️ Zbyt skomplikowana architektura dla prostych aplikacji.
- ⚠️ Brak reaktywnego ekosystemu (np. używanie JPA zamiast R2DBC).
- ⚠️ Niedoszacowanie krzywej nauki dla zespołu.

11 W PRAKTYCE — jak wybrać?

1. Określ wymagania aplikacji (obciążenie, typ danych, wymagania realtime).
2. Sprawdź, czy Twoje zależności są reaktywne (np. R2DBC, WebClient).
3. Oceń doświadczenie zespołu.
4. Rozważ koszty złożoności i utrzymania.
5. Zacznij od prostego rozwiązania — wybierz MVC, a na WebFlux przejdź, gdy naprawdę tego potrzebujesz.

12 SZYBKIE PODSUMOWANIE

- ✅ Spring MVC = klasyczny, sprawdzony, blokujący.
- ✅ WebFlux = reaktywny, nieblokujący, dla wysokiej współbieżności.
- ✅ Nie ma jednego „lepszego" rozwiązania — wszystko zależy od Twojego przypadku.
- ✅ WebFlux nie zawsze jest szybszy.
- ✅ Wybieraj świadomie, na podstawie wymagań.
- ✅ Obie technologie są częścią ekosystemu Springa.

Technologia to narzędzie, a nie cel. Wybieraj mądrze! ☺

- ✅ Lepszy kod
- ✅ Lepsze aplikacje
- ✅ Lepsza przyszłość



Dobre decyzje tworzą lepsze systemy! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

REACTOR: MONO, FLUX I BACKPRESSURE

Podstawy programowania reaktywnego w Springu

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 82/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest Project Reactor?

Project Reactor to biblioteka do programowania reaktywnego, na której oparty jest Spring WebFlux. Dostarcza niskopoziomowe typy **Mono** i **Flux** oraz implementację specyfikacji Reactive Streams.

- Reaktywne, nieblokujące przetwarzanie danych
- Wysoka wydajność i skalowalność
- Obsługa backpressure
- Bogaty zestaw operatorów
- Ścisła integracja ze Springiem (WebFlux, Data, Security)
- Działa w oparciu o specyfikację Reactive Streams

"Małe kroki do bardziej reaktywnego świata!"

2 Publisher, Subscriber i Subscription

Publisher — Źródło danych, które emituje elementy do Subscribera.

Subscriber — Otrzymuje dane, przetwarza je i reaguje na sygnały (onNext, onError, onComplete).

Subscription — Reprezentuje połączenie między Publisherem a Subscriberem. Pozwala kontrolować żądanie elementów (request) – kluczowe dla backpressure.

Reactor implementuje specyfikację **Reactive Streams** (`org.reactivestreams`).

3 Mono<T> — 0 lub 1 element

Mono reprezentuje sekwencję, która może wyemitować 0 lub 1 element, a następnie zakończyć się.

```java
Mono<String> mono = Mono.just("Hello");

mono.map(String::toUpperCase)
  .subscribe(
    value -> System.out.println("Wartość: " + value),
    error -> System.err.println("Błąd: " + error),
    () -> System.out.println("Zakończono")
);
```

Przykłady użycia:

- Pojedynczy wynik z bazy danych
- Odpowiedź z zewnętrznego API
- Operacja zakończona statusem (np. zapis, usunięcie)

4 Flux<T> — wiele elementów/strumień

Flux reprezentuje sekwencję 0..N elementów, które są emitowane w czasie i mogą być przetwarzane reaktywnie.

```java
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

flux.filter(n -> n % 2 == 0)
   .map(n -> n * 10)
   .subscribe(
      n -> System.out.println("Liczba: " + n),
      error -> System.err.println("Błąd: " + error),
      () -> System.out.println("Strumień zakończony")
 );
```

Przykłady użycia:

- Strumień danych z bazy, pliku, API, zdarzeń
- Przetwarzanie kolekcji, danych w czasie rzeczywistym
- Pipeline wielu operacji na danych

5 Jak działa backpressure?

Backpressure to mechanizm kontroli szybkości emisji danych. Chroni Subscriber przed przeciążeniem, pozwalając określić ile elementów chce otrzymać.

- Subscriber wysyła żądanie (request) określonej liczby elementów.
- Publisher emituje dane zgodnie z żądaniem.
- Zapobiega przepełnieniu buforów i OutOfMemory.
- W Reactorze działa automatycznie dla Flux oraz przy użyciu operatorów.

Przykład: kontrola żądania elementów

```java
Flux.range(1, 100)
   .doOnRequest(n ->
      System.out.println("Zażądano: " + n))
   .subscribe();
```

6 Typowe operatory: map, flatMap, filter, zip

Operatory pozwalają transformować, łączyć i filtrować dane w strumieniach reaktywnych.

```java
Flux<User> users = Flux.just(u1, u2, u3);

users
   .filter(u -> u.isActive())          // filtracja
   .map(User::getEmail)                // transformacja
   .flatMap(email -> sendEmail(email)) // operacja async
   .zipWith(getStatus(), (e, s) ->     // łączenie
      new EmailResult(e, s))
   .subscribe(result -> System.out.println(result));
```

Najczęściej używane operatory:

- map() – transformuje elementy
- flatMap() – mapowanie z operacją asynchroniczną
- filter() – filtruje elementy
- zip() – łączy kilka strumieni
- take(), skip(), distinct(), collectList() i wiele innych

7 Łańcuch reaktywny i lazy execution

W Reactorze operacje są leniwe (lazy) – nic się nie dzieje, dopóki nie nastąpi subskrypcja. Tworzymy łańcuch operacji, który jest wykonywany dopiero przy subscribe().

```java
Flux<String> pipeline = Flux.just("A", "B", "C")
  .map(String::toLowerCase)
  .filter(s -> s.startsWith("a") || s.startsWith("b"));

System.out.println("łańcuch utworzony, ale nie wykonany");

pipeline.subscribe(
   v -> System.out.println("Element: " + v));
```

8 Obsługa błędów: onErrorResume, onErrorReturn, retry

Reaktywne strumienie mogą kończyć się błędem. Reactor oferuje wiele możliwości obsługi błędów.

```java
Flux<String> flux = Flux.just("1", "2", "error", "3")
  .map(s -> {
    if (s.equals("error")) throw new RuntimeException
      ("Coś poszło nie tak");
    return s;
  })
  .onErrorResume(e -> Flux.just("wartość domyślna"))
  .retry(2)
  .subscribe(v -> System.out.println("Otrzymano: " + v),
             e -> System.err.println("Błąd: " + e));
```

9 Scheduler i wykonywanie asynchroniczne

Scheduler pozwala kontrolować na jakim wątku wykonywane są operacje. Dzięki temu można oddzielić operacje blokujące od strumienia reaktywnego.

```java
Flux.fromIterable(getBlockingData())
   .subscribeOn(Schedulers.boundedElastic())
   .map(this::process)
   .publishOn(Shedullers.parallel())
   .subscribe(v -> System.out.println(v));
```

Najważniejsze schedulery:

- Schedulers.immediate() – bieżący wątek
- Schedulers.single() – pojedynczy wątek
- Schedulers.parallel() – wiele wątków (CPU)
- Schedulers.boundedElastic() – operacje blokujące (I/O)

10 Hot vs cold publisher

Cold publisher – każdy subskrybent otrzymuje własną sekwencję danych (produkcja zaczyna się od nowa).

Hot publisher – emituje dane niezależnie od subskrybentów, wszyscy dostają te same elementy.

- Flux.just() – zimny (cold)
- Flux.interval() – gorący (hot)
- share(), publish(), replay() – tworzą gorące publikery

11 Dobre praktyki

- Używaj Mono dla pojedynczych wyników, Flux dla strumieni.
- Stosuj backpressure i odpowiednie operatory.
- Unikaj blokowania w strumieniach reaktywnych.
- Wybieraj właściwe schedulery (boundedElastic dla operacji blokujących).
- Obsługuj błędy w sposób jawny.
- Pisz czytelne i małe łańcuchy operacji.
- Testuj strumienie przy użyciu StepVerifier.
- Stosuj logowanie i debugowanie (checkpoint()).

12 Najczęstsze pułapki

- Używanie .block() i .toIterable() w aplikacji reaktywnej.
- Mieszanie kodu blokującego z operacjami reaktywnymi.
- Brak obsługi błędów (onError).
- Zbyt duże bufory, brak backpressure.
- Nieodpowiedni scheduler (np. operacje blokujące na wątkach CPU).
- Wyciek zasobów (np. połączeń).
- Zbyt złożone łańcuchy bez podziału na metody.

13 Szybkie podsumowanie

- Project Reactor – baza dla Spring WebFlux.
- Mono – 0 lub 1 element.
- Flux – wiele elementów.
- Backpressure chroni przed przeciążeniem.
- Bogaty zestaw operatorów.
- Operacje są leniwe (lazy).
- Scheduler pozwala kontrolować wątki.
- Obsługuj błędy i testuj strumienie.
- To fundament nowoczesnych, reaktywnych aplikacji w Springu!

Reaktywność dziś — skalowalność jutro!

- Większa wydajność
- Lepsza skalowalność
- Nowe możliwości



Reaktywne pomysły tworzą lepsze aplikacje!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

PRZEPŁYW ŻĄDANIA W WEBFLUX

Od requestu do odpowiedzi w modelu nieblokującym

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 83/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest WebFlux?

Spring WebFlux to reaktywny stos webowy w Springu, umożliwiający nieblokujące przetwarzanie żądań w oparciu o modele **Mono** i **Flux**.

Zbudowany dla skalowalności i dużej liczby współbieżnych połączeń.

Działa na Netty (domyślnie), ale może też używać innych serwerów.

Kluczowe: programowanie reaktywne, brak blokowania, strumieniowanie danych, backpressure.

2 Wejście żądania: Netty / HttpHandler / DispatcherHandler

HTTP Request (z przeglądarki/klienta)

↓

Netty (Reactor Netty) — **Netty** odbiera połączenie HTTP nieblokująco (event-loop).

↓

HttpHandler (adapter) — **HttpHandler** (np. ReactorHttpHandler) przekazuje żądanie do Springa.

↓

DispatcherHandler (centralny komponent) — **DispatcherHandler** znajduje pasującą trasę (router lub kontroler) i buduje łańcuch obsługi.

3 RouterFunction vs kontroler adnotacyjny

RouterFunction (funkcyjny styl)	@RestController (adnotacje)
✅ Jawna konfiguracja tras	✅ Znany i wygodny styl
✅ Większa elastyczność	✅ Szybszy start
✅ Dobre do małych i średnich aplikacji	✅ Czytelny kod
✅ Brak adnotacji – czysty kod funkcyjny	✅ Dobre dla większości projektów

Java

```
@Bean
public RouterFunction<ServerResponse>
  routes(UserHandler handler) {
  return RouterFunctions
   .route(GET("/users"), handler::all)
   .andRoute(POST("/users"),
    handler::create);
}
```

Java

```
@RestController
@RequestMapping("/users")
public class UserController {
  @GetMapping
  public Flux<User> all() {
    return service.findAll();
  }
}
```

4 WebFilter i przetwarzanie do drodze

WebFilter pozwala na obsługę żądań i odpowiedzi w sposób nieblokujący – np. logowanie, autoryzacja, metryki.

Java

```
@Component
public class LoggingFilter implements WebFilter {
  @Override
  public Mono<Void> filter(ServerWebExchange ex,
        WebFilterChain chain) {
    long start = System.currentTimeMillis();
    return chain.filter(ex)
      .doFinally(signal -> {
        long time = System.currentTimeMillis() - start;
        System.out.println("[" + ex.getRequest().getPath()
        + "] " + time + " ms");
      });
  }
}
```

5 Deserializacja / body / codecs

WebFlux używa **HttpMessageReader** i **HttpMessageWriter** (codecs) do deserializacji i serializacji danych.

- ✅ Obsługa JSON, XML, form-data, tekstu itp.
- ✅ Oparto o nieblokujące strumienie (DataBuffer).
- ✅ Automatyczna konwersja do typu (np. DTO).
- ✅ Możliwość własnych, niestandardowych codecs.

Java

```
public Mono<User> create(
  @RequestBody Mono<CreateUserRequest> req) {
  return req
    .map(this::toUser)
    .flatMap(service::save);
}
```

6 Handler / service i łańcuch Mono/Flux

Handler (kontroler lub funkcja) wywołuje serwisy, które zwracają **Mono** lub **Flux**. Łańcuch operatorów przetwarza dane w sposób nieblokujący.

- ✅ Używaj operatorów (map, flatMap, filter, itp.)
- ✅ Nie blokuj w łańcuchu (brak .block()).
- ✅ Obsługuj przepływ: dane mogą być strumieniem.

Java

```
@GetMapping("/users/{id}")
public Mono<ResponseEntity<User>> get(
  @PathVariable String id) {
  return service.findById(id)
    .map(ResponseEntity::ok)
    .defaultIfEmpty(ResponseEntity.notFound()
      .build());
}
```

7 Tworzenie odpowiedzi HTTP

Wynik handlera to Mono<ServerResponse> (lub obiekt zwracany z kontrolera). Spring automatycznie serializuje go i wysyła do klienta.

- ✅ Możesz ustawić status, nagłówki, ciało.
- ✅ Obsługa strumieniowania (Flux).
- ✅ Wspiera SSE, pliki, dane binarne.

Java

```
return ServerResponse.ok()
  .contentType(MediaType.APPLICATION_JSON)
  .bodyValue(new UserDto("1", "Ala"));

// lub w kontrolerze
@GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> stream() {
  return Flux.interval(Duration.ofSeconds(1))
    .map(i -> "tick " + i);
```

8 Obsługa błędów i wyjątków

W WebFlux błędy propagują się w strumieniu. Możesz obsłużyć je globalnie lub lokalnie.

- ✅ Używaj onErrorResume, onErrorReturn.
- ✅ Globalna obsługa za pomocą @ControllerAdvice.
- ✅ Mapuj błędy na odpowiednie kody HTTP.

Java

```
@ControllerAdvice
public class GlobalErrorHandler {
  @ExceptionHandler(IllegalArgumentException.class)
  public Mono<ResponseEntity<ErrorResponse>>
  handle(IllegalArgumentException ex) {
  return Mono.just(ResponseEntity.badRequest()
    .body(new ErrorResponse("Niepoprawne dane")));
}
```

9 Event-loop i model wątków

WebFlux wykorzystuje model event-loop Netty. Kilka wątków obsługuje tysiące połączeń.

- ✅ Brak tworzenia wątku na żądanie.
- ✅ Wątki event-loop nie mogą być blokowane.
- ✅ Długie operacje deleguj na boundedElastic().
- ✅ Domyślnie liczba wątków = liczba rdzeni CPU.



10 Gdzie blokowanie szkodzi?

Blokowanie w wątkach event-loop może zatrzymać całą aplikację i spowodować spadek wydajności.

- ✅ Nie używaj .block(), .blockFirst(), .blockLast().
- ✅ Nie wykonuj blokujących operacji (np. JDBC, File I/O) w event-loop.
- ✅ Jeśli musisz – przenieś na Scheduler.boundedElastic().

Java

```
// ŹLE - blokuje wątek event-loop!
public Mono<User> bad() {
  return Mono.just(repository.findById(1L).block()); // NIE!
}
// DOBRZE - delegacja na boundedElastic
public Mono<User> good() {
  return Mono.fromCallable(() -> repository.findById(1L))
  .subscribeOn(Schedulers.boundedElastic());
```

11 W praktyce — typowy przepływ krok po kroku

1. Klient wysyła żądanie HTTP do aplikacji WebFlux.
2. **Netty** odbiera połączenie (event-loop) i przekazuje do **HttpHandler**.
3. DispatcherHandler znajduje trasę (router lub kontroler).
4. Żądanie przechodzi przez **WebFilter** (np. logowanie, autoryzacja).
5. Deserializacja ciała żądania (codecs) do obiektu/DTO.
6. Handler wywołuje serwis, który zwraca Mono/Flux.
7. Dane są przetwarzane w nieblokującym łańcuchu operatorów.
8. Tworzona jest odpowiedź HTTP (status, nagłówki, ciało).
9. Odpowiedź trafia z powrotem przez Netty do klienta.

12 Dobre praktyki

- ✅ Pisz nieblokujący kod – używaj Mono/Flux.
- ✅ Nie blokuj wątków event-loop.
- ✅ Stosuj WebFilter do cross-cutting concerns.
- ✅ Waliduj dane i obsługuj błędy globalnie.
- ✅ Używaj DTO i reactrive codecs.
- ✅ Monitoruj aplikację (metryki, logi, tracing).
- ✅ Testuj przepływ (WebTestClient).
- ✅ Planowanie blokujących operacji na boundedElastic().
- ✅ Dokumentuj API (np. z springdoc-openapi).

13 Najczęstsze pułapki

- ❌ Używanie .block() w kodzie aplikacji.
- ❌ Wykonywanie blokujących operacji (JDBC, pliki) w event-loop.
- ❌ Brak obsługi błędów w strumieniu.
- ❌ Niewłaściwe zarządzanie backpressure.
- ❌ Zwracanie bardzo dużych kolekcji zamiast strumieniowania.
- ❌ Nieprawidłowa konfiguracja timeouts.
- ❌ Nieużywanie WebFilter do logowania i metryk.
- ❌ Pominięcie testów wydajnościowych.
- ❌ Nieodpowiedni dobór schedulera.

Technologia to narzędzie, a nie cel. Wybieraj mądrze! ☺

- ✅ Lepszy kod
- ✅ Lepsze aplikacje
- ✅ Lepsza przyszłość



Dobre decyzje tworzą lepsze systemy! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

MAPA 84/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

R2DBC I REAKTYWNY DOSTĘP DO DANYCH

Jak pracować z bazą danych bez blokującego JDBC

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest R2DBC?

R2DBC (Reactive Relational Database Connectivity) to specyfikacja dla nieblokującego, reaktywnego dostępu do relacyjnych baz danych.

- Działa w pełni asynchronicznie (non-blocking)
- Wykorzystuje model reaktywny (Publisher/Subscriber)
- Jest częścią ekosystemu Spring (Spring Data R2DBC)
- Umożliwia skalowanie aplikacji przy dużej liczbie równoczesnych połączeń
- Alternatywa dla tradycyjnego, blokującego JDBC

2 R2DBC vs JDBC

Cecha	R2DBC	JDBC
Model	Reaktywny (nieblokujący)	Tradycyjny (blokujący)
API	Reaktywne (Publisher)	Synchroniczne
Wątki	Nie blokuje wątków (ciągłość I/O)	Blokuje wątki podczas I/O
Skalowalność	Wysoka (dużo połączeń)	Ograniczona
Wydajność	Lepsza przy dużej liczbie połączeń	Dobra w prostych przypadkach
Obsługa transakcji	Reaktywne transakcje	Klasyczne transakcje JDBC
Przypadki użycia	Aplikacje reaktywne, mikroserwisy, API	Tradycyjne aplikacje, proste systemy

3 Kiedy warto użyć R2DBC?

- Budujesz aplikację w pełni reaktywną (WebFlux).
- Oczekujesz dużej liczby równoczesnych połączeń.
- Wykonujesz operacje I/O do wielu zewnętrznych usług.
- Zależy Ci na lepszej skalowalności i wykorzystaniu zasobów.
- Pracujesz w środowisku chmurowym (mikroserwisy).
- Aplikacja wykonuje dużo krótkich zapytań.

Nie zawsze R2DBC jest konieczne!
Jeśli Twoja aplikacja jest prosta, ma umiarkowany ruch albo korzysta tylko z jednego połączenia, klasyczny JDBC może być wystarczający.

4 Startery i podstawowa konfiguracja

Aby użyć R2DBC w Spring Boot, dodaj odpowiedni starter w pliku pom.xml (lub build.gradle) i skonfiguruj połączenie z bazą danych.

pom.xml (Maven)

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
  </dependency>
  <dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-postgresql</artifactId>
  </dependency>
</dependencies>
```

5 ConnectionFactory i sterowniki

ConnectionFactory to reaktywny odpowiednik **DataSource** – tworzy nieblokujące połączenia z bazą.

application.yml

```yaml
spring:
  r2dbc:
    url: r2dbc:postgresql://localhost:5432/demo
    username: demo
    password: secret
  sql:
    init:
      mode: always
```

Popularne sterowniki R2DBC:

- r2dbc-postgresql – PostgreSQL
- r2dbc-mysql – MySQL
- r2dbc-h2 – H2 (do testów)
- r2dbc-mssql – Microsoft SQL Server
- r2dbc-mariadb – MariaDB

6 DatabaseClient / R2dbcEntityTemplate

Spring oferuje dwa główne podejścia do pracy z bazą danych:

- **DatabaseClient** – niskopoziomowe, elastyczne API do wykonywania zapytań SQL.
- **R2dbcEntityTemplate** – wyższe poziom abstrakcji, podobny do JdbcTemplate.

DatabaseClient – przykład

```java
@Service
public class UserService {
  private final DatabaseClient db;

  public UserService(DatabaseClient db) {
    this.db = db;
  }

  public Flux<User> findAll() {
    return db.sql("SELECT * FROM users")
        .map((row, meta) -> new User(
          row.get("id", Long.class),
          row.get("name", String.class)))
        .all();
}
```

7 Reaktywne repozytoria

Spring Data R2DBC umożliwia tworzenie repozytoriów reaktywnych, podobnie jak w przypadku JPA.

UserRepository.java

```java
import org.springframework.data.r2dbc.repository
    .R2dbcRepository;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface UserRepository
  extends R2dbcRepository<User, Long> {

  Flux<User> findByStatus(String status);

  Mono<User> findByEmail(String email);
}
```

8 Przykład encji i zapytania

Encja – User.java

```java
@Table("users")
public class User {
  @Id
  private Long id;
  private String name;
  private String email;
  private String status;
  // konstruktory, gettery,
} // settery
```

Użycie repozytorium w serwisie

```java
Mono<User> user = userRepository
        .findById(1L);

Flux<User> activeUsers = userRepository
        .findByStatus("ACTIVE");
```

9 Transakcje reaktywne — co trzeba wiedzieć

- Użyj adnotacji @Transactional z menedżerem dedykowanym dla R2DBC.
- Transakcje działają w modelu reaktywnym (w ramach Mono/Flux).
- Wszystkie operacje w transakcji muszą być wykonywane w tym samym łańcuchu reaktywnym.
- Nie można używać tradycyjnych, blokujących operacji.

Przykład transakcji reaktywnej

```java
@Service
public class UserService {
  private final UserRepository repository;

  @Transactional
  public Mono<Void> createUser(User user) {
    return repository.save(user)
        .then();
}
```

10 Dobre praktyki

- Używaj R2DBC tylko w aplikacjach reaktywnych (WebFlux).
- Wybieraj odpowiedni sterownik (np. r2dbc-postgresql).
- Stosuj reaktywne repozytoria lub DatabaseClient.
- Pamiętaj o obsłudze błędów (onErrorResume, doOnError).
- Wykorzystuj walidację danych (Bean Validation).
- Monitoruj wydajność i pulę połączeń (R2DBC Pool).
- Testuj aplikację z użyciem Testcontainers (np. PostgreSQL).
- Unikaj mieszania R2DBC i JDBC w tej samej aplikacji.

11 Najczęstsze pułapki

- Używanie R2DBC w aplikacji MVC (zamiast WebFlux).
- Mieszanie kodu blokującego (np. JDBC, RestTemplate) z reaktywnym.
- Brak obsługi błędów w łańcuchu reaktywnym.
- Nieprawidłowa konfiguracja URL lub sterownika.
- Utrata kontekstu transakcji między operatorami.
- Oczekiwanie, że wszystkie biblioteki wspierają R2DBC.
- Niewłaściwe zarządzanie pulą połączeń.
- Zbyt wczesne przejście na R2DBC bez realnej potrzeby.

12 Szybkie podsumowanie

- R2DBC to reaktywny, nieblokujący dostęp do relacyjnych baz.
- Działa świetnie z WebFlux i Spring Data R2DBC.
- Wybieraj odpowiedni sterownik i konfiguruj ConnectionFactory.
- Używaj DatabaseClient lub reaktywnych repozytoriów.
- Pamiętaj o transakcjach reaktywnych i obsłudze błędów.
- Stosuj dobre praktyki i testuj rozwiązanie.
- R2DBC zapewnia lepszą skalowalność i wydajność w aplikacjach reaktywnych.

Reaktywne dane to większe możliwości!

- Szybsze aplikacje
- Lepsza skalowalność
- Nowoczesna architektura

SPRING BOOT

Dane też mogą być reaktywne!

INTEGRUJ
TESTUJ
ROZWIJAJ
WDRAŻAJ
SUKCEDUJ

Pisz testy
Buduj pewniej
Rozwijaj szybciej

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

VIRTUAL THREADS W SPRING BOOT

Lekka współbieżność w praktyce

MAPA 85/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym są virtual threads?

Virtual threads (wątki wirtualne) to lekka forma współbieżności wprowadzona w Javie (Project Loom) i dostępna od JDK 21, a w Spring Boot 3.2+.

Działają jak zwykłe wątki (Thread), ale są znacznie lżejsze i tworzone w bardzo dużej liczbie.

Idealne do aplikacji, które wykonują wiele operacji blokujących (np. wywołania HTTP, zapytania do bazy, operacje plikowe).

To nadal model programowania imperatywnego — ale z masową skalowalnością.

2 Platform threads vs virtual threads

Cecha	Platform thread (tradycyjny)	Virtual thread (wirtualny)
Koszt utworzenia	Wysoki	Bardzo niski
Zużycie pamięci	Duże (MB)	Małe (KB)
Liczba wątków	Ograniczona (tysiące)	Bardzo duża (miliony)
Planowanie	Przez system operacyjny	Przez JVM (scheduler)
Przypadek użycia	CPU-bound, długie obliczenia	I/O-bound, operacje blokujące
Dostępność	Od zawsze	JDK 21 (Spring Boot 3.2+)

3 Dlaczego są lekkie?

Virtual threads nie są mapowane 1:1 do wątków systemu operacyjnego.

Blokujące operacje I/O powodują „odstawienie" wątku i zwolnienie wątku systemowego, co pozwala obsłużyć inne wątki.

Wątki wirtualne mają znacznie mniejsze zużycie pamięci (kilka KB zamiast MB).

JVM efektywnie zarządza ich planowaniem i przełączaniem, dzięki czemu można tworzyć miliony wątków bez dużych kosztów.

4 Jak włączyć w Spring Boot?

Wystarczy ustawić właściwość w konfiguracji aplikacji. Spring Boot automatycznie użyje wirtualnych wątków do obsługi żądań oraz dla zadań wykonywanych przez @Async, @schedule i inne mechanizmy.

YAML

```yaml
spring:
  threads:
    virtual:
      enabled: true
```

Działa od Spring Boot 3.2+ (wymaga JDK 21+).

5 Kiedy pomagają?

- Aplikacje z wieloma operacjami blokującymi I/O (np. wywołania REST, bazy danych, pliki).
- Duża liczba równoczesnych użytkowników.
- Serwisy integracyjne i aplikacje typu backend for frontend (BFF).
- Migracja z modeli reaktywnych, gdy chcesz zachować styl imperatywny.
- Uproszczenie kodu — brak potrzeby stosowania reaktywnych API (Flux/Mono).
- Poprawa skalowalności i lepsze wykorzystanie zasobów serwera.

6 Gdzie nie pomogą?

- Długie, intensywne obliczenia CPU (np. przetwarzanie obrazów, symulacje).
- Kod, który blokuje wątki w sposób niepodlegający współpracy z JVM (np. natywne biblioteki).
- Sytuacje, gdy problemem jest wydajność samej bazy danych, zewnętrznych usług lub sieci.
- Aplikacje z bardzo małą liczbą żądań (nie odczujesz różnicy).
- Nie zastępują architektury reaktywnej, gdy potrzebujesz strumieniowego przetwarzania danych i backpressure.

7 Wpływ na aplikacje webowe i I/O

Spring Boot z virtual threads obsługuje każde żądanie HTTP w osobnym wątku wirtualnym.

Pozwala to obsłużyć znacznie większą liczbę jednoczesnych połączeń, szczególnie przy operacjach blokujących (np. wywołania REST).

Każde wywołanie do bazy danych blokuje tylko wątek wirtualny, który może zostać odstawiony bez blokowania zasobów systemowych.

W rezultacie aplikacje webowe mogą osiągać znacznie wyższą przepustowość przy mniejszym zużyciu pamięci i prostszym kodzie.

8 Executor i przykłady użycia

Możesz tworzyć własne executory oparte o wątki wirtualne lub korzystać z domyślnej konfiguracji Spring Boota.

Java

```java
@Bean
public Executor virtualThreadExecutor() {
    return Executors.newVirtualThreadPerTaskExecutor();
}
```

Przykładowe użycie:

Java

```java
@Service
public class ReportService {
    private final Executor executor;
    public ReportService(Executor executor) {
        this.executor = executor;
    }
    public CompletableFuture<String> generujRaport() {
        return CompletableFuture.supplyAsync(() -> {
            // operacja blokująca, np. wywołanie REST
            return "OK";
        }, executor);
    }
}
```

9 Pinning i ograniczenia

Pinning wątku wirtualnego występuje, gdy wątek nie może zostać odstawiony, np. z powodu:

- używania zsynchonizowanych bloków (synchronized),
- wywołań metod natywnych (JNI),
- długich operacji w sekcjach krytycznych,
- niektórych bibliotek, które blokują wątki systemowe.

W takich sytuacjach wątek wirtualny zachowuje się jak tradycyjny i może zajmować wątek systemowy, co ogranicza skalowalność.

Stosuj współczesne, nieblokujące biblioteki i unikaj długich synchronized.

10 Monitoring i debugging

Virtual threads są widoczne w narzędziach JVM (JFR, jstack, VisualVM, IntelliJ IDEA).

W JDK 21+ dostępne są dodatkowe zdarzenia JFR dla virtual threads.

Możesz analizować zrzuty wątków (thread dump) – wątki wirtualne mają nazwy zaczynające się od "VirtualThread".

W Spring Boot nadal możesz używać actuator/metrics (Micrometer) do monitoringu aplikacji.

11 Dobre praktyki

- Włącz virtual threads, gdy Twoja aplikacja jest I/O-bound.
- Używaj najnowszej wersji JDK (21+) i Spring Boot.
- Unikaj długich synchronized i operacji natywnych.
- Stosuj nieblokujące biblioteki i klienci (np. WebClient, JDBC z nowymi sterownikami).
- Testuj obciążenie i monitoruj zachowanie aplikacji.
- Używaj własnych executorów tylko gdy potrzebujesz dodatkowej kontroli.
- Pamiętaj o ograniczeniach – to nie srebrna kula.

12 Najczęstsze pułapki

- Oczekiwanie magicznego przyspieszenia w aplikacjach CPU-bound.
- Używanie bibliotek, które pinują wątki.
- Zakładanie, że virtual threads rozwiążą problemy z wydajnością bazy danych.
- Ignorowanie monitoringu i analizy wątków.
- Łączenie z przestarzałymi bibliotekami, które nie współpracują z Loom.
- Brak testów obciążeniowych przed wdrożeniem na produkcję.

13 Szybkie podsumowanie

- Virtual threads = lekka współbieżność w Javie 21+.
- Włącz w Spring Boot jednym ustawieniem.
- Idealne do aplikacji I/O-bound.
- Prostszy kod, większa skalowalność.
- Nie zastępują programowania reaktywnego.
- Monitoruj, testuj i unikaj pułapek.
- Świetne narzędzie w nowoczesnych aplikacjach Spring Boot!

Technologia to narzędzie, a nie cel. Wybieraj mądrze!

- Lepszy kod
- Lepsze aplikacje
- Lepsza przyszłość

SPRING BOOT

Dobre decyzje tworzą lepsze systemy!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

MAPA 86/100

BLOCKING vs ASYNC vs VIRTUAL THREADS vs REACTIVE

Jak wybrać model współbieżności i przetwarzania?

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Co oznacza blocking?

- Model synchroniczny, w którym wątek czeka na zakończenie operacji (np. I/O) blokując się.
- Prosty i intuicyjny model programowania.
- Wątek jest zablokowany podczas wykonywania operacji.
- Dobrze sprawdza się w prostych aplikacjach i przy niewielkim obciążeniu.

2 Co oznacza async?

- Model asynchroniczny, w którym operacje wykonywane są w tle, a wątek nie czeka na ich zakończenie.
- Wykorzystuje mechanizmy takie jak CompletableFuture, @Async, TaskExecutor.
- Umożliwia obsługę wielu zadań równolegle.
- Wymaga zarządzania wynikami (Callback, Future) i obsługą błędów.

3 Co dają virtual threads?

- Lekkie wątki wprowadzone w Java 21 (Project Loom), pozwalające pisać kod synchroniczny przy wysokiej współbieżności.
- Znacznie niższy koszt tworzenia i przełączania wątków.
- Idealne do aplikacji I/O-bound (np. mikroserwisy, API, HTTP).
- Umożliwiają obsługę tysięcy równoczesnych połączeń bez skomplikowanego kodu.

4 Co oznacza reactive?

- Model reaktywny oparty na strumieniach danych (Reactive Streams), nieblokujący i oparty na zdarzeniach.
- W Spring Boot realizowany przez Spring WebFlux i biblioteki Reactor (Mono, Flux).
- Obsługuje backpressure, strumieniowanie i przepływ danych.
- Wymaga innego, reaktywnego sposobu myślenia (łańcuchy operatorów, brak blokowania).

5 Porównanie podejść

Cecha	Blocking	Async	Virtual Threads	Reactive
Złożoność	Niska	Średnia	Niska	Wysoka
Model wątków	Klasyczne wątki	Pula wątków (Executor)	Lekkie wątki (wirtualne)	Event loop (nieblokujący)
Zużycie zasobów	Wysokie	Średnie	Niskie	Bardzo niskie
Skalowalność	Niska	Średnia	Wysoka	Bardzo wysoka
Krzywa nauki	Niska	Średnia	Niska	Wysoka

6 Typowe zastosowania

Blocking
- Proste aplikacje CRUD
- Aplikacje wewnętrzne (admin, backoffice)
- Mały ruch i niewielka liczba użytkowników
- Szybkie prototypy
- Integracje z bibliotekami blokującymi

Async
- Wywołania zewnętrznych API (HTTP, REST)
- Operacje w tle (np. powiadomienia)
- Równoległe przetwarzanie zadań
- Długie operacje I/O
- Aplikacje hybrydowe (mieszane podejścia)

Virtual Threads
- Mikroserwisy z dużą liczbą połączeń
- Aplikacje I/O-bound
- Migracja z kodu blokującego
- Serwery HTTP (z tysiącami żądań)
- Prosty kod przy wysokiej współbieżności

Reactive
- Systemy o bardzo wysokiej skali
- Strumieniowanie danych (np. SSE, WebSocket)
- Aplikacje real-time
- Przetwarzanie strumieni (Flux)
- Systemy odporne na przeciążenia (backpressure)

7 Zalety i ograniczenia

Podejście	Zalety	Ograniczenia
Blocking	✓ Prostota ✓ Czytelny kod ✓ Łatwa obsługa błędów	✗ Niska skalowalność ✗ Duże zużycie wątków Nie nadaje się do dużego obciążenia
Async	✓ Nieblokujące I/O ✓ Lepsze wykorzystanie zasobów ✓ Równoległość zadań	✗ Większa złożoność ✗ Zarządzanie wynikami Trudniejsza obsługa błędów
Virtual Threads	✓ Prosty kod (synchroniczny styl) ✓ Wysoka współbieżność ✓ Niskie zużycie zasobów	✗ Wymaga Javy 21+ ✗ Nie rozwiązuje problemu blokujących bibliotek ✗ Naduj ograniczenia I/O (np. bazy danych)
Reactive	✓ Najwyższa skalowalność ✓ Obsługa strumieni ✓ Backpressure ✓ Niskie zużycie zasobów	✗ Wysoka złożoność ✗ Stroma krzywa nauki ✗ Inny paradygmat programowania ✗ Trudniejsze debugowanie

8 Kiedy które podejście wybrać?

Wybierz blocking gdy:
- prosty projekt,
- niewielkie obciążenie,
- zależy Ci na szybkości wdrożenia,
- korzystasz z bibliotek blokujących.

Wybierz async gdy:
- potrzebujesz równoległości,
- wywołujesz zewnętrzne API,
- chcesz wykonać zadania w tle,
- nie chcesz blokować wątków.

Wybierz virtual threads gdy:
- masz dużo jednoczesnych połączeń,
- używasz Javy 21+,
- chcesz prostego kodu przy wysokiej współbieżności,
- migracja z blokującego kodu.

Wybierz reactive gdy:
- budujesz system o dużej skali,
- potrzebujesz strumieniowania danych,
- zależy Ci na niskim zużyciu zasobów,
- akceptujesz wyższą złożoność kodu i nauki.

9 Błędne założenia

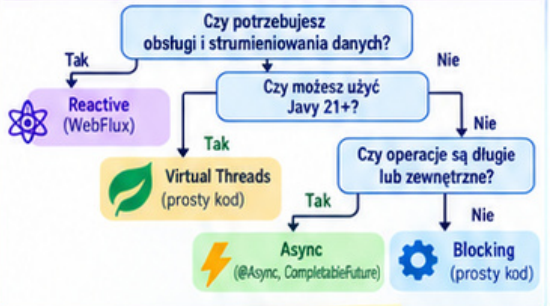
Blocking
- ✗ „Blocking zawsze jest zły."
- ✓ Jest dobry w prostych aplikacjach i małych projektach.

Async
- ✗ „@Async rozwiązuje wszystkie problemy skalowalności."
- ✓ Nie zastąpi architektury i dobrej obsługi błędów.

Virtual Threads
- ✗ „Virtual threads sprawiają, że każda aplikacja staje się automatycznie skalowalna."
- ✓ Nie rozwiążą problemów z blokującymi bibliotekami i wolnym I/O.

Reactive
- ✗ „Reactive jest zawsze szybsze."
- ✓ Nie zawsze – wymaga innego modelu myślenia i ma sens głównie przy dużych obciążeniach i strumieniach danych.

10 W praktyce — drzewko decyzji



11 Najczęstsze pułapki

1. Mieszanie podejść bez uzasadnienia.
2. Blokowanie w kodzie reaktywnym (np. .block(), Thread.sleep()).
3. Używanie @Async w metodach prywatnych (lub w tej samej klasie).
4. Brak obsługi błędów w operacjach async/reactive.
5. Niewłaściwe dobranie liczby wątków w puli dla async.
6. Oczekiwanie, że virtual threads rozwiążą problemy z wolnymi bazami danych.
7. Nadmierna komplikacja prostych projektów przez użycie reactive.

12 Szybkie podsumowanie

- Nie ma jednego idealnego podejścia – wybierz świadomie, zgodnie z wymaganiami.
- Blocking – prostota, ale niska skalowalność.
- Async – elastyczność i równoległość.
- Virtual threads – wysoka współbieżność przy prostym kodzie (Java 21+).
- Reactive – maksymalna skalowalność i obsługa strumieni danych.
- Liczy się kontekst – typ aplikacji, obciążenie, zespół i dostępne technologie.
- Dobrze dobrany model to lepsza wydajność, prostszy kod i bardziej przyszłościowa architektura.

Dobry wybór dziś, lepsza skalowalność jutro! ☺

- Analizuj potrzeby projektu.
- Wybierz odpowiednie podejście.
- Testuj, mierz, optymalizuj.
- Buduj lepsze aplikacje!

SPRING BOOT

Lepsza architektura to większe możliwości! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

WEBSOCKET I SERVER-SENT EVENTS

Komunikacja w czasie rzeczywistym w Spring Boot

MAPA 87/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest WebSocket?

WebSocket to protokół komunikacji dwukierunkowej (full-duplex) między klientem a serwerem, pojedyncze, trwałe połączenie oparte na HTTP (handshake).

Po nawiązaniu połączenia obie strony mogą wysyłać wiadomości w dowolnym momencie, bez dodatkowych żądań HTTP.

W Spring Boot obsługiwany przez spring-websocket (STOMP) lub natywny WebSocket.

2 Czym jest SSE?

Server-Sent Events (SSE) to mechanizm jednokierunkowej komunikacji, w którym serwer wysyła strumień wydarzeń do klienta przez zwykłe połączenie HTTP.

Połączenie pozostaje otwarte, a serwer może wysyłać wiele wiadomości w formacie tekstowym (text/event-stream).

SSE jest prostszy od WebSocket, wspierany natywnie przez przeglądarki i idealny do scenariuszy z aktualizacjami od serwera (np. powiadomienia).

3 WebSocket vs SSE — porównanie

Cecha	WebSocket	SSE
Kierunek komunikacji	Dwukierunkowa	Jednokierunkowa (serwer → klient)
Protokół	WebSocket (RFC 6455)	HTTP (text/event-stream)
Format danych	Dowolny (np. JSON, binarne)	Tekstowy (eventy)
Obsługa w Springu	spring-websocket (STOMP)	Spring MVC / WebFlux
Wsparcie w przeglądarkach	Szerokie	Bardzo dobre
Zastosowanie	Czat, gry, współpraca w czasie rzeczywistym	Powiadomienia, aktualizacje danych, monitoring
Złożoność	Wyższa	Niższa

4 Typowe zastosowania

- Czaty i komunikatory
- Powiadomienia w czasie rzeczywistym
- Aktualizacje danych (np. kursy, statystyki)
- Gry online
- Śledzenie postępu zadań (np. przetwarzanie plików)
- Transmisje na żywo (live dashboard)
- Współpraca w czasie rzeczywistym (np. edytory dokumentów)

5 Konfiguracja w Springu

Java

```java
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements
        WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(
            MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(
            StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOriginPatterns("*");
    }
}
```

6 Przykład endpointu WebSocket

Java

```java
@Controller
public class ChatController {

    @MessageMapping("/chat")
    @SendTo("/topic/messages")
    public ChatMessage handleMessage(
            @Payload ChatMessage message,
            Principal principal) {

        message.setSender(principal.getName());
        message.setTimestamp(LocalDateTime.now());
        return message;
    }
}

public record ChatMessage(String content,
                          String sender,
                          LocalDateTime timestamp) {}
```

7 Przykład SSE

Java

```java
@RestController
@RequestMapping("/api/events")
public class EventController {

    @GetMapping(produces = "text/event-stream")
    public Flux<ServerSentEvent<String>> stream() {
        return Flux.interval(Duration.ofSeconds(1))
                .map(i -> ServerSentEvent.<String>builder()
                        .id(String.valueOf(i))
                        .event("tick")
                        .data("Aktualny czas: " +
                                LocalDateTime.now())
                        .build());
    }
}
```

8 Skalowanie i load balancing

WebSocket i SSE utrzymują długotrwałe połączenia, co wymaga odpowiedniej konfiguracji infrastruktury.

- Używaj load balancera z obsługą połączeń długotrwałych (np. Nginx, HAProxy).
- W przypadku wielu instancji stosuj rozproszony broker wiadomości (np. Redis, RabbitMQ).
- Stosuj mechanizmy sticky sessions (jeśli wymagane).
- Monitoruj liczbę połączeń i zasoby serwera.
- W klastrach Kubernetes używaj Ingress z poprawną konfiguracją dla WebSocket/SSE.

9 Bezpieczeństwo i autoryzacja

- Używaj TLS (HTTPS/WSS).
- Autoryzuj użytkowników (np. JWT, OAuth2).
- W Spring Security skonfiguruj zabezpieczenia dla endpointów WebSocket i SSE.
- Sprawdzaj uprawnienia w kontrolerach (@PreAuthorize lub ręcznie).
- Waliduj i filtruj przesyłane dane.
- Chroń przed nadużyciami (rate limiting, limity połączeń).

10 Dobre praktyki

- Wybieraj SSE, gdy wystarczy komunikacja od serwera.
- Używaj WebSocket, gdy potrzebna jest komunikacja dwukierunkowa.
- Definiuj jasny format wiadomości (np. JSON).
- Obsługuj ponowne połączenia i sprawdzaj stan klienta.
- Loguj ważne zdarzenia i błędy.
- Testuj scenariusze obciążeniowe.
- Dbaj o obserwowalność (metryki, logging, tracing).

11 Najczęstsze pułapki

- Używanie WebSocket, gdy wystarczy SSE.
- Brak obsługi ponownego połączenia po rozłączeniu.
- Problemy z CORS i konfiguracją proxy.
- Niewłaściwe zarządzanie sesjami użytkowników.
- Brak limitów i zabezpieczeń (możliwość nadużyć).
- Wyciek zasobów (niezamknięte połączenia).
- Zakładanie, że WebSocket działa „z pudełka" za każdym load balancerem.

12 Szybkie podsumowanie

- WebSocket i SSE umożliwiają komunikację w czasie rzeczywistym w Spring Boot.
- WebSocket = dwukierunkowa, SSE = jednokierunkowa.
- Wybieraj rozwiązanie odpowiednie do potrzeb.
- Pamiętaj o skalowaniu, bezpieczeństwie i monitoringu.
- Stosuj dobre praktyki i unikaj typowych pułapek.
- Dzięki nim możesz budować nowoczesne, reaktywne aplikacje.

Technologia to narzędzie, a nie cel. Wybieraj mądrze!

Lepszy kod
Lepsze aplikacje
Lepsza przyszłość

SPRING BOOT

Dobre decyzje tworzą lepsze systemy!

ANALIZUJ
WYBIERAJ
BUDUJ
ROZWIJAJ
SUKCEDUJ

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

GRAPHQL W SPRING BOOT

Schema, query, mutation i praktyczne użycie

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 88/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest GraphQL?

GraphQL to język zapytań do API oraz środowisko wykonawcze, stworzone przez Facebooka, które pozwala klientom precyzyjnie określić, jakie dane chcą pobrać.

- Zapytania o dokładnie takie dane, jakie są potrzebne
- Jeden endpoint (zwykle /graphql)
- Silne typowanie, introspekcja (GraphiQL)
- Mniejsze zużycie pasma i mniej nadmiarowych danych

2 GraphQL vs REST

Cecha	GraphQL	REST
Endpoint	Jeden (/graphql)	Wiele endpointów (/api/...)
Pobierane dane	Tylko to, czego potrzebujesz	Często za dużo lub za mało danych
Wersjonowanie	Zazwyczaj nie wymaga (ewolucja schematu)	Wymaga wersjonowania (np. /v1, /v2)
Cache'owanie	Bardziej złożone	Prostsze (HTTP)
Złożoność zapytań	Jedno złożone zapytanie	Wiele prostych wywołań
Technologia	Język zapytań + wykonanie	Zasoby (URI) + metody HTTP

3 Schema i typy

Schemat (schema) definiuje dostępne typy, pola, zapytania i mutacje. Jest zapisany w języku SDL (Schema Definition Language).

graphql

```
type User {
  id: ID!
  name: String!
  email: String
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  content: String
  author: User!
}

type Query {
  user(id: ID!): User
  users: [User!]!
}

type Mutation {
  createUser(input: CreateUserInput!): User!
}
```

4 Query, Mutation, Subscription

Query — Służy do pobierania danych (read-only). Nie zmienia stanu aplikacji.

Mutation — Służy do modyfikowania danych (tworzenie, aktualizacja, usuwanie). Zmienia stan aplikacji.

Subscription — Umożliwia subskrypcję zmian i otrzymywanie danych w czasie rzeczywistym (np. przez WebSocket).

5 @QueryMapping, @MutationMapping, @SchemaMapping

@QueryMapping — Mapuje metodę na pole typu Query. Uproszczona alternatywa dla @GraphQLQuery.

@MutationMapping — Mapuje metodę na pole typu Mutation. Uproszczona alternatywa dla @GraphQLMutation.

@SchemaMapping — Mapuje metodę na pole typu (obiektowe, zagnieżdżone), np. User.posts. Używana do pobierania powiązanych danych.

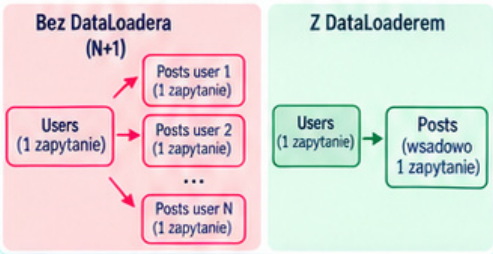
6 Przykładowy resolver

Java

```
@Controller
public class UserResolver {

  private final UserService userService;

  public UserResolver(UserService userService) {
    this.userService = userService;
  }

  @QueryMapping
  public User user(@Argument Long id) {
    return userService.findById(id);
  }

  @QueryMapping
  public List<User> users() {
    return userService.findAll();
  }

  @MutationMapping
  public User createUser(@Argument CreateUserInput input) {
    return userService.create(input);
  }
}
```

7 DataLoader i problem N+1

Problem N+1 występuje, gdy dla listy obiektów wykonujemy wiele dodatkowych zapytań do bazy danych (np. pobranie posts dla każdego usera).
DataLoader grupuje zapytania i ładuje dane wsadowo.



8 Walidacja i błędy

- Używaj **Bean Validation** (@Valid, @NotNull, @Size). Sprawdzenie danych wejściowych w mutacjach.
- Rzuć zrozumiałe wyjątki (np. GraphQLException).
- Obsługuj błędy globalnie (DataFetcherExceptionResolver).
- Zwracaj czytelne komunikaty błędów dla klienta.
- Nie ujawniaj wrażliwych informacji w komunikatach.

Java

```
@MutationMapping
public User createUser(@Argument @Valid CreateUserInput input) {
  try {
    return userService.create(input);
  } catch (UserAlreadyExistsException ex) {
    throw new GraphQLException("Użytkownik już istnieje");
  }
}
```

9 Kiedy warto użyć GraphQL?

- Masz aplikację kliencką (np. web, mobile) o różnych potrzebach danych.
- Chcesz ograniczyć ilość przesyłanych danych.
- Potrzebujesz elastycznego API, które łatwo ewoluuje.
- Masz złożone relacje między danymi.
- Aplikacja wymaga funkcji czasu rzeczywistego (Subscriptions).
- Masz wielu różnych klientów (frontend, aplikacje mobilne, partnerzy).

10 Dobre praktyki

- Projektuj czytelny i spójny schemat.
- Używaj DataLoadera do eliminacji problemu N+1.
- Stosuj walidację danych wejściowych.
- Dziel resolvery na mniejsze klasy.
- Używaj DTO jako argumentów w mutacjach.
- Zabezpiecz endpoint (autoryzacja, limity, monitoring).
- Wykorzystuj GraphiQL do testowania.
- Dbaj o cache'owanie (np. na poziomie DataLoadera).
- Pisz testy integracyjne dla zapytań i mutacji.

11 Najczęstsze pułapki

- Problem N+1 (brak DataLoadera).
- Zbyt duże i nieoptymalne zapytania.
- Brak walidacji danych wejściowych.
- Ujawnianie szczegółów błędów w odpowiedziach.
- Brak autoryzacji i kontroli dostępu do pól.
- Niewłaściwe użycie Subscriptions (brak obsługi WebSocketów).
- Zbyt dużą logika biznesowa w resolverach.
- Brak testów dla zapytań i mutacji.
- Nieprzemyślany schemat – trudny w utrzymaniu.

12 Szybkie podsumowanie

- GraphQL to elastyczny sposób na tworzenie API.
- Używa schematu, zapytań, mutacji i subskrypcji.
- Spring Boot zapewnia świetne wsparcie dla GraphQL.
- Stosuj DataLoader, walidację i dobre praktyki.
- Sprawdza się w nowoczesnych aplikacjach, szczególnie z frontendami.

Dokładnie takie dane, jakie potrzebujesz – to siła GraphQL!



Lepsze API to szczęśliwi klienci!

- Lepsza wydajność
- Większa elastyczność
- Nowe możliwości
- Twoja przewaga na rynku



Dobry schemat to podstawa sukcesu!

Pisz testy
Buduj pewniej
Rozwijaj szybciej

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I ARCHITEKTURA

ARCHITEKTURA MODUŁOWA APLIKACJI SPRING BOOT

Jak porządkować kod i granice odpowiedzialności?

MAPA 89/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Po co modularność?

- Lepsza organizacja kodu i czytelniejsza struktura.
- Niezależny rozwój modułów i mniejsze ryzyko konfliktów.
- Łatwiejsze testowanie i utrzymanie.
- Możliwość pracy równoległej w zespołach.
- Gotowość na rozwój systemu i zmiany wymagania biznesowych.

2 Package-by-layer vs package-by-feature

Package-by-layer (warstwowy)

```
com.example.app
├── controller
├── service
├── repository
├── model
└── config
```

- ✗ Dobre na początek, ale gorzej się skaluje.
- ✗ Kodu rozrzucony po warstwach.
- ✗ Trudniejsze zrozumienie kontekstu biznesowego.

Package-by-feature (modułowy lub domenowy)

```
com.example.app
├── user
│   ├── controller
│   ├── service
│   ├── repository
│   └── model
├── order
├── product
└── ...
```

- ✓ Lepsza spójność biznesowa.
- ✓ Łatwiejsza praca zespołów.
- ✓ Prostsze utrzymanie dużych aplikacji.
- ✓ Dobrze współgra z DDD.

3 Moduł aplikacyjny — co powinien zawierać?

- Logikę biznesową związaną z jednym obszarem domenowym (np. użytkownicy, zamówienia).
- Encje, DTO i logikę domenową.
- Serwisy, repozytoria i kontrolery (jeśli moduł udostępnia API).
- Konfigurację specyficzną dla modułu.
- Testy jednostkowe i integracyjne.
- Opcjonalnie: klienty do innych modułów, mapery, walidatory, zdarzenia domenowe.

4 API modułu i ukrywanie implementacji

- Każdy moduł powinien udostępniać jasno zdefiniowane API (interfejsy, REST, komunikaty).
- Ukryj implementację wewnętrzną (np. klasy, encje, repozytoria).
- Używaj interfejsów i pakietów „public api".
- Eksportuj tylko to, co jest niezbędne.
- Dzięki temu moduły są niezależne i łatwiej je rozwijać lub zastępować innymi implementacjami.

5 Zależności między modułami

- ✓ Zależności powinny być jednokierunkowe (moduł A może zależeć od B, ale nie odwrotnie).
- ✓ Używaj zasad Clean Architecture / Dependency Inversion — zależ od interfejsów, nie implementacji.
- ✓ Ogranicz zależności transtiive (łańcuchowe).
- ✓ Unikaj cyklicznych zależności.
- ✓ W większych projektach rozważ podział na osobne moduły Maven/Gradle (multi-module).
- ✓ Komunikacja między modułami może odbywać się przez wywołania lokalne, zdarzenia lub API REST.

6 Granice domeny i bounded contexts

- Każdy moduł powinien odpowiadać jednej dziedzinie biznesowej (bounded context).
- Stosuj zasady DDD — oddziel modele domenowe.
- Unikaj dzielenia encji między modułami.
- Komunikuj się między kontekstami przez API, zdarzenia lub integracje, nie przez bezpośredni dostęp do bazy.
- Granice kontekstu wynikają z języka biznesowego i odpowiedzialności zespołów.
- Dobrze zdefiniowane konteksty ułatwiają skalowanie i utrzymanie systemu.

7 Porty i adaptery / architektura heksagonalna

- Logika biznesowa znajduje się w centrum (domena).
- **Porty** — interfejsy definiujące, co aplikacja potrzebuje (np. repozytorium, bramka płatności).
- **Adaptery** — implementacje portów (np. JPA, REST, Kafka, pliki).
- Dzięki temu możemy łatwo podmieniać technologie (np. baza danych) bez zmian w logice domenowej.
- Spring Boot dobrze wspiera ten model przez wstrzykiwanie zależności i konfigurację.

8 Przykładowa struktura pakietów

```
com.example.shop
├── user
│   ├── api          (publiczne API modułu)
│   ├── controller
│   ├── service
│   ├── repository
│   ├── model
│   ├── exception
│   └── mapper
├── order
│   └── ...          (podobna struktura)
├── product
│   └── ...          (podobna struktura)
└── common
    ├── config       (wspólna konfiguracja)
    ├── security
    ├── event        (zdarzenia, komunikaty)
    └── util         (narzędzia pomocnicze)
```

Każdy moduł to spójny fragment domeny! ☺

9 Testowanie modułów

- ✓ Testy jednostkowe — logika biznesowa, serwisy, mapery, walidatory.
- ✓ Testy warstwy API — kontrolery (np. z MockMvc).
- ✓ Testy integracyjne — z bazą testową (np. @SpringBootTest + Testcontainers).
- ✓ Testuj moduły w izolacji — używaj mocków dla zależności z innych modułów.
- ✓ Weryfikuj kontrakty API modułów.
- ✓ Dbaj o pokrycie testami kluczowej logiki i przypadków brzegowych.

10 Dobre praktyki

- ✓ Projektuj moduły wokół funkcjonalności biznesowych.
- ✓ Stosuj zasady DDD i czyste architektury.
- ✓ Używaj interfejsów i ukrywaj implementację.
- ✓ Minimalizuj zależności między modułami.
- ✓ Dbaj o spójne nazewnictwo i strukturę pakietów.
- ✓ Dokumentuj API modułów.
- ✓ Automatyzuj testy i integrację (CI/CD).
- ✓ Planuj modularność od początku projektu.

11 Najczęstsze pułapki

- ⚠ Zbyt duże moduły (tzw. „god classes").
- ⚠ Brak jasno zdefiniowanych granic kontekstu.
- ⚠ Współdzielenie encji między modułami.
- ⚠ Cykliczne zależności.
- ⚠ Zbyt wiele logiki w kontrolerach.
- ⚠ Ujawnianie wewnętrznych klas i encji w API.
- ⚠ Brak testów integracyjnych.
- ⚠ Przedwczesne wydzielanie modułów (bez realnej potrzeby biznesowej).

12 Szybkie podsumowanie

- ✓ Modularność to lepsza organizacja, skalowalność i łatwiejsze utrzymanie aplikacji Spring Boot.
- ✓ Stosuj podejście package-by-feature i zasady DDD.
- ✓ Każdy moduł ma jasno określone API i granice odpowiedzialności.
- ✓ Używaj portów i zdapterów dla większej elastyczności.
- ✓ Testuj moduły w izolacji i dbaj o jakość kodu.
- ✓ Unikaj najczęstszych pułapek, projektuj świadomie.
- ✓ Modularna architektura to inwestycja w przyszłość Twojej aplikacji!

Dobrze zaprojektowane moduły rosną razem z Twoim biznesem! ☺

- ✓ Czytelna struktura.
- ✓ Niezależne moduły.
- ✓ Lepsze aplikacje.
- ✓ Większa przyszłość!



Modularność dziś – lepsze jutro! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 09 — REAKTYWNOŚĆ, WYDAJNOŚĆ I A ARCHITEKTURA

MAPA 90/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

GRANICE TRANSAKCJI, DOMENY I INTEGRACJI

Gdzie kończy się transakcja lokalna, a zaczyna system rozproszony

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest granica transakcji?

- Granica transakcji określa, które operacje w bazie danych stanowią jedną, niepodzielną całość (ACID).
- W Spring Boot transakcje deklarujemy za pomocą **@Transactional**.
- Granica transakcji zwykle pokrywa się z granicą jednego serwisu i jednej bazy danych (monolit lub pojedynczy mikroserwis).
- Nie rozciągamy transakcji lokalnej na wiele usług ani zewnętrzne systemy!

2 Transakcja lokalna vs system rozproszony

Transakcja lokalna	System rozproszony
Jedna baza danych	Wiele usług i baz danych
ACID (spójność, trwałość, izolacja, atomowość)	Brak globalnej transakcji ACID (trudna lub niemożliwa)
Szybka i niezawodna	Konieczność mechanizmów takich jak Saga, Outbox, eventy domenowe
Prosta obsługa (@Transactional)	Wyższa złożoność i ryzyko niespójności
Przykład: zapis zamówienia w jednej bazie	Przykład: zamówienie + płatność + wysyłka w różnych serwisach

3 Warstwa serwisowa i spójność danych

- Logika biznesowa powinna znajdować się w warstwie serwisowej (@Service).
- Granica transakcji obejmuje operacje domenowe w ramach jednego serwisu.
- Spójność danych wewnątrz serwisu zapewnia @Transactional.
- Do komunikacji z innymi serwisami używamy zdarzeń domenowych lub integracji asynchronicznych.

```java
@Service
public class OrderService {
    @Transactional
    public Order createOrder(CreateOrderCommand cmd) {
        Order order = orderRepository.save(new Order(cmd));
        order.addDomainEvent(new OrderCreatedEvent(order.getId()));
        return order;
    }
}
```

4 Zdarzenia domenowe

- Zdarzenie domenowe to fakt z przeszłości (np. OrderCreated), który ma znaczenie biznesowe.
- Publikujemy je po udanej transakcji (lokalnie lub przez mechanizm outbox).
- Umożliwiają luźne powiązanie serwisów i reaktywne podejście do integracji.
- Zdarzenia opisują coś, co się wydarzyło, a nie to, co ma się wydarzyć.

```java
public class OrderCreatedEvent {
    private final UUID orderId;
    private final Instant occurredAt = Instant.now();

    // konstruktory  gettery
}
```

5 Eventual consistency

- Model spójności ostatecznej – dane w różnych serwisach będą spójne, ale nie natychmiast.
- Dopuszcza tymczasową niespójność.
- Wymaga mechanizmów takich jak zdarzenia, retry, kompensacje.
- Dobrze sprawdza się w architekturach asynchronicznych i mikroserwisach.
- Użytkownik może przez chwilę widzieć nieaktualne dane – to normalne.

„Eventual consistency to nie błąd, to świadomy kompromis na rzecz skalowalności."

6 Transactional outbox

- Wzorzec zapewniający niezawodną publikację zdarzeń.
- Zdarzenie zapisywane jest w tej samej transakcji co dane domenowe (tabela outbox).
- Osobny proces (np. scheduler) wysyła zdarzenia do brokera (Kafka, RabbitMQ).
- Chroni przed utratą zdarzeń przy awarii.
- Popularny sposób integracji w systemach rozproszonych.

```java
@Entity
public class OutboxMessage {
    @Id private UUID id;
    private String type;
    private String payload;
    private LocalDateTime createdAt;
    private boolean published;
}
```

7 Saga / orkiestracja vs choreografia

Orkiestracja (Saga orchestrator)

- Centralny koordynator zarządza przepływem.
- Łatwiejsze monitorowanie i obsługa błędów.
- Przykład: serwis Zamówienia steruje procesem płatności i wysyłki.

Choreografia (Saga choreography)

- Serwisy reagują na zdarzenia.
- Brak centralnego koordynatora.
- Większa niezależność, ale trudniejsze śledzenie przepływu.
- Dobra dla prostszych scenariuszy.

Oba podejścia realizują transakcję rozproszoną przez sekwencję lokalnych transakcji i kompensacje.

8 Idempotencja przy integracjach

- Idempotencja oznacza, że wielokrotne wykonanie tego samego żądania daje ten sam efekt biznesowy.
- Chroni przed duplikatami (np. ponowne dostarczenie wiadomości).
- Stosuj klucze idempotencji (np. requestId), unikalne constrainty lub sprawdzanie stanu.
- Ważna przy obsłudze eventów, webhooków i komunikacji z brokerem.

```java
public boolean handleEvent(DomainEvent event) {
    if (eventStore.exists(event.getId())) {
        return false; // już obsłużone
    }
    eventStore.save(event);
    // dalsza logika
    return true;
}
```

9 Typowy przepływ biznesowy

1. Klient składa zamówienie (serwis Zamówienia) — Zamówienie
2. Zapis danych + event (transakcja lokalna + outbox) — Outbox
3. Publikacja zdarzenia (np. OrderCreated) — Broker zdarzeń
4. Reakcja innych serwisów (Płatności, Magazyn, Wysyłka) — Inne serwisy
5. Aktualizacja stanów (eventual consistency) — Finalny stan

10 Dobre praktyki

- Trzymaj granice transakcji w obrębie jednego serwisu.
- Używaj zdarzeń domenowych do integracji.
- Stosuj wzorzec transactional outbox.
- Projektuj operacje idempotentne.
- Wybierz orkiestrację lub choreografię saga adekwatnie do złożoności.
- Monitoruj przepływ zdarzeń i obsługuj błędy (retry, DLQ).
- Dokumentuj kontrakty zdarzeń (np. schema).
- Testuj scenariusze awaryjne i kompensacje.

11 Najczęstsze pułapki

- Rozciąganie transakcji na wiele serwisów.
- Brak obsługi błędów i retry.
- Utrata zdarzeń (brak outbox).
- Nieidempotentne operacje.
- Zbyt drobno podzielone serwisy.
- Brak kompensacji w sadze.
- Niewłaściwe monitorowanie i logowanie.
- Założenie natychmiastowej spójności.

12 Szybkie podsumowanie

- Transakcja lokalna = jedna baza, ACID, @Transactional.
- System rozproszony = zdarzenia, outbox, saga, spójność ostateczna.
- Zdarzenia domenowe łączą serwisy.
- Stosuj idempotencję i obsługę błędów.
- Wybierz orkiestrację lub choreografię.
- Projektuj z myślą o niezawodności, skalowalności i obserwowalności.

„Dobre granice dają wolność do skalowania."



Architektura to sztuka mądrych kompromisów! ☺

- Lepsze systemy
- Lepsze aplikacje
- Lepsza przyszłość



Dobre decyzje tworzą lepsze systemy! ☺

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 91/100

Pisz kod
Buduj projekty
Osiągaj więcej!

OD KODU DO EXECUTABLE JAR

Jak aplikacja Spring Boot przechodzi od źródeł do uruchamialnego artefaktu

Lepszy kod
Lepsze aplikacje
Lepsza przyszłość!

1 Czym jest executable JAR?

JAR

Executable JAR to samodzielny plik zawierający aplikację Spring Boot wraz ze wszystkimi zależnościami (i wbudowanym serwerem, np. Tomcatem).

- Można go uruchomić poleceniem `java -jar ...`
- Nie wymaga instalacji dodatkowego serwera aplikacyjnego.
- Ułatwia wdrażanie na dowolnym środowisku (lokalnie, serwer, chmura).

„Jeden plik, cała aplikacja, zero problemów z konfiguracją serwera!"

2 Standardowy pipeline builda

Kod źródłowy → Kompilacja (kod → .class) → Testy (jednostkowe i integracyjne) → Package (tworzenie artefaktu) → Executable JAR

- Piszemy kod w katalogu src/
- Kompilujemy kod do plików .class
- Uruchamiamy testy (np. JUnit)
- Tworzymy pakiet (maven package / gradle bootJar)
- Otrzymujemy gotowy plik JAR w target/ lub build/libs/

Od kodu do działającej aplikacji!

3 Rola Mavena i Gradle

Maven

- Popularne i stabilne narzędzie do budowania projektów Java.
- Plik konfiguracyjny pom.xml.
- Zarządzanie zależnościami, wtyczkami i cyklem życia projektu.

Gradle

- Nowoczesne narzędzie z elastyczną konfiguracją.
- Plik konfiguracyjny build.gradle.
- Szybsze buildy dzięki mechanizmowi cache i inkrementalności.

Oba narzędzia świetnie współpracują ze Spring Bootem!

4 Układ projektu Spring Boot

```
demo-app/
├── src/
│   ├── main/
│   │   ├── java/          # kod źródłowy
│   │   └── resources/     # pliki konfiguracyjne
│   └── test/
│       ├── java/          # testy jednostkowe
│       └── resources/     # zasoby testowe
├── pom.xml                # (Maven)
└── build.gradle           # (Gradle)
```

Czytelna struktura = łatwiejszy rozwój i utrzymanie projektu!

5 spring-boot-maven-plugin / bootJar

Maven (spring-boot-maven-plugin)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Tworzy executable JAR

Gradle (bootJar)

```
plugins {
    id 'org.springframework.boot' version '3.2.0'
}
tasks.bootJar {
    archiveFileName = 'demo-app.jar'
}
```

Zadanie bootJar tworzy executable JAR

6 Co trafia do target/ lub build/libs/

target/ (Maven)

- demo-app-0.0.1-SNAPSHOT.jar ← executable
- demo-app-0.0.1-SNAPSHOT.jar.original
- classes/
- generated-sources/
- maven-archiver/
- ...

build/libs/ (Gradle)

- demo-app-0.0.1-SNAPSHOT.jar ← executable
- demo-app-0.0.1-SNAPSHOT-plain.jar
- ...

Plik z dopiskiem .jar to gotowa aplikacja do uruchomienia!

7 Przykładowe komendy

Maven

```
# Czyszczenie, testy i budowanie
mvn clean package
```

Gradle

```
# Czyszczenie, testy i tworzenie JAR
./gradlew clean bootJar
```

Po wykonaniu komendy otrzymasz executable JAR w katalogu target/ lub build/libs/.

8 Uruchomienie aplikacji

```
java -jar demo-app-0.0.1-SNAPSHOT.jar
```

- Aplikacja startuje z wbudowanym serwerem (np. Tomcatem).
- Domyślnie działa na porcie 8080.
- Można przekazać parametry:

```
java -jar demo-app.jar
     --spring.profiles.active=prod
```

Kilka sekund i aplikacja działa! Tak proste jest to w Spring Bootcie.

9 Różnica między JAR a WAR

JAR (executable)	WAR
Samodzielny plik z wbudowanym serwerem.	Przeznaczony do deploymentu na zewnętrznym serwerze (np. Tomcat, JBoss).
Uruchamiany przez java -jar.	Nie zawiera wbudowanego serwera (zwykle).
Domyślny i zalecany w Spring Boot.	Stosowany, gdy firma ma własną infrastrukturę serwerów aplikacyjnych.

Dla większości projektów Spring Boot wybieraj JAR!

10 Dobre praktyki builda

- Używaj najnowszej stabilnej wersji Spring Boota.
- Uruchamiaj testy przed tworzeniem artefaktu.
- Stosuj profile (dev, test, prod).
- Minimalizuj rozmiar JAR (wyklucz niepotrzebne zależności).
- Włączaj weryfikację jakości kodu (np. Checkstyle, SonarQube).
- Automatyzuj buildy w CI/CD (np. GitHub Actions, GitLab CI).

Czyste buildy dziś = mniej problemów na produkcji!

11 Najczęstsze pułapki

- Pominięcie testów – błąd trafia na produkcję.
- Nieprawidłowa konfiguracja wtyczki.
- Zbyt duży rozmiar JAR (niepotrzebne zależności).
- Brak profili środowiskowych.
- Twardo zakodowane dane konfiguracyjne.
- Niezgodne wersje JDK i Spring Boot.
- Problemy z uprawnieniami przy uruchomieniu.

Testuj, weryfikuj, automatyzuj – unikniesz wielu problemów!

12 Szybkie podsumowanie

- Executable JAR = samodzielna aplikacja.
- Maven i Gradle ułatwiają budowanie.
- Plik tworzony jest w target/ lub build/libs/.
- Uruchamiamy go poleceniem java -jar.
- JAR różni się od WAR sposobem uruchomienia.
- Warto stosować dobre praktyki i automatyzację.

Od kodu do produkcji – prosto, szybko i niezawodnie!

13 Diagram przepływu procesu

Kod źródłowy (src/) → Kompilacja (kod → .class) → Testy (JUnit) → Package (mvn package / bootJar) → Executable JAR (z target/ lub build/libs/) → Uruchomienie (java -jar)

Małe kroki. Wielkie możliwości!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

ANATOMIA SPRING BOOT JAR

Loader, BOOT-INF, klasy, zależności i warstwy

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 92/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest fat jar / executable jar?

- Spring Boot tworzy tzw. **fat jar** (ang. executable jar) – pojedynczy plik JAR zawierający aplikację oraz wszystkie jej zależności.
- Można go uruchomić poleceniem `java -jar` bez dodatkowej konfiguracji.
- Ułatwia wdrażanie i działa na każdym środowisku z JDK.
- Nie wymaga zewnętrznego serwera aplikacji (np. Tomcat).

„Jeden plik, cała aplikacja!"

2 Główne elementy archiwum

Spring Boot JAR ma charakterystyczną strukturę z kilkoma kluczowymi elementami:

META-INF	Metadane archiwum, manifest i informacje dla loadera.
BOOT-INF/classes	Skompilowane klasy naszej aplikacji (zasoby, pliki konfiguracyjne).
BOOT-INF/lib	Wszystkie zależności w postaci plików JAR (wewnątrz archiwum).

„BOOT-INF to serce aplikacji Spring Boot."

3 Rola Spring Boot Loader

- Spring Boot Loader to zestaw klas odpowiedzialnych za uruchomienie aplikacji z fat jar.
- Znajduje się w pakiecie org.springframework.boot.loader.
- Obsługuje specjalny system klas (nested jars).
- Umożliwia ładowanie zależności z katalogu BOOT-INF/lib.
- Nie rozpakowuje plików na dysku – działa bezpośrednio z archiwum.

Główne klasy loadera: Java

```
org.springframework.boot.loader.JarLauncher
  // domyślny launcher
org.springframework.boot.loader.WarLauncher
  // dla plików WAR
org.springframework.boot.loader.PropertiesLauncher
  // zewnętrzna konfiguracja
```

4 Jak aplikacja startuje z JarLauncher

1. Uruchamiamy: `java -jar app.jar`
2. JVM czyta manifest: `Main-Class`
3. Wskazuje na `org.springframework.boot.loader.JarLauncher`
4. `JarLauncher` tworzy własny classloader
5. Ładuje klasy aplikacji i zależności z BOOT-INF
6. Uruchamia metodę main naszej aplikacji

„JarLauncher robi całą magię za Ciebie."

5 Gdzie znajdują się klasy aplikacji

- Skompilowane klasy naszej aplikacji znajdują się w katalogu BOOT-INF/classes.
- Znajdują się tam również zasoby (np. application.yml, pliki static/).
- Struktura katalogów wewnątrz BOOT-INF/classes jest taka sama jak w projekcie (pakiety).

Przykład zawartości:

```
BOOT-INF/
├── classes/
│   ├── com/example/demo/
│   │   └── DemoApplication.class
│   └── service/
│       └── UserService.class
├── application.yml
└── static/
```

6 Gdzie są zależności

- Wszystkie zależności (biblioteki) znajdują się w katalogu BOOT-INF/lib.
- Każda zależność to osobny plik JAR (nested jar).
- Nie trzeba ich rozpakowywać – loader ładuje je bezpośrednio z archiwum.
- Zawiera zarówno zależności compile, jak i runtime.

Przykład:

```
BOOT-INF/
├── lib/
│   ├── spring-boot-3.2.0.jar
│   ├── spring-context-6.1.0.jar
│   ├── logback-classic-1.4.0.jar
│   └ ...
```

7 Zwykły jar vs boot jar

	Zwykły JAR	Spring Boot JAR
Zawartość	Tylko klasy aplikacji	Klasy aplikacji + wszystkie zależności
Struktura	Prosta	Z katalogiem BOOT-INF
Uruchamianie	Wymaga zewnętrznych zależności w classpath	Wystarczy `java -jar`
Zastosowanie	Biblioteki, małe projekty	Samodzielne aplikacje
Manifest	Zwykły	Wskazuje na JarLauncher

„Boot jar jest samowystarczalny!"

8 Layered jar (warstwy)

- Spring Boot może tworzyć warstwowy JAR, który ułatwia budowanie obrazów Docker i przyspiesza wdrażanie.
- Każda warstwa zawiera inny zestaw plików (zależności, klasy aplikacji, zasoby).
- Pozwala na lepsze cachowanie warstw w Dockerze – zmiana kodu nie wymusza pobierania wszystkich zależności.

Włącz warstwy w Maven:
`<layers><enabled>true</enabled></layers>`
(lub użyj domyślnej konfiguracji Spring Boot)

9 Przykład struktury katalogów

```
app.jar (zawartość)
├── META-INF/
│   ├── MANIFEST.MF
│   └── spring-boot/
│       └ ...
├── BOOT-INF/
│   ├── classes/
│   │   ├── com/example/demo/
│   │   │   └── DemoApplication.class
│   │   └── application.yml
│   └── lib/
│       ├── spring-boot-3.2.0.jar
│       ├── spring-context-6.1.0.jar
│       └ ...
└── org/
    └── springframework/
        └── boot/loader/
            └ ...
```

10 Dobre praktyki

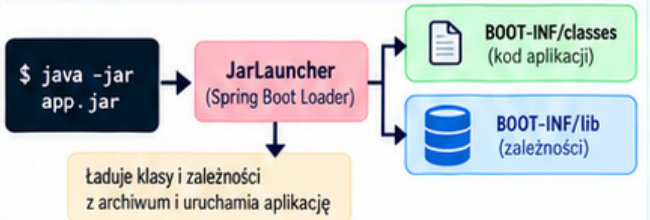
- Używaj warstwowego JAR przy konteneryzacji.
- Nie dodawaj niepotrzebnych plików do archiwum.
- Optymalizuj rozmiar – wyklucz zbędne dependencje.
- Używaj *.dockerignore* przy budowaniu obrazu.
- Sprawdzaj zawartość JAR (np. jar tf app.jar).
- Stosuj skinny jar tylko w szczególnych przypadkach.

Mniejszy, czytelny i powtarzalny obraz = szybsze wdrożenia!

11 Najczęstsze pułapki

- ✗ Próba uruchomienia zwykłego JAR bez zależności w classpath.
- ✗ Ręczne modyfikowanie zawartości BOOT-INF.
- ✗ Dodawanie dużych, niepotrzebnych zależności.
- ✗ Brak warstw przy budowaniu obrazów Docker – wolne wdrożenia.
- ✗ Nieprawidłowa konfiguracja manifestu.
- ✗ Próba rozpakowania nested jars (zwykle nie jest to potrzebne).

12 Schemat działania java -jar



13 Szybkie podsumowanie

- ✓ Spring Boot JAR = aplikacja + zależności.
- ✓ Kluczowe katalogi: META-INF, BOOT-INF/classes, BOOT-INF/lib.
- ✓ JarLauncher uruchamia aplikację.
- ✓ Layered jar ułatwia konteneryzację.
- ✓ Wystarczy polecenie `java -jar`.
- ✓ Znajomość struktury pomaga w diagnozowaniu problemów.

„Zrozumienie wnętrza JAR to lepsze wdrożenia i pewniejsze działanie w produkcji!"

Ta sama wiedza daje większe możliwości! ☺



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 93/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

DOCKERFILE VS BUILDPACKS

Dwa sposoby konteneryzacji apiikacji Spring Boot

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Po co konteneryzować?

- Łatwe i powtarzalne wdrożenia
- To samo środowisko w każdych warunkach
- Izolacja aplikacji i jej zależności
- Szybsze uruchamianie i skalowanie
- Prostsze zarządzanie infrastrukturą
- Idealne dla chmury i mikroserwisów

Konteneryzacja eliminuje problem „u mnie działa"!

2 Czym jest Dockerfile?

- Plik tekstowy z instrukcjami budowania obrazu Docker.
- Pełna kontrola nad każdym krokiem budowania.
- Możesz użyć dowolnego obrazu bazowego i narzędzi.
- Popularne rozwiązanie, używane w całym ekosystemie.
- Wymaga więcej wiedzy o Dockerze i utrzymania pliku.
- Daje maksymalną elastyczność.

3 Czym są Cloud Native Buildpacks?

- Automatycznie wykrywają typ aplikacji (np. Spring Boot) i budują obraz.
- Nie wymagają pisania Dockerfile.
- Używają gotowych, bezpiecznych buildpacków (np. Paketo).
- Tworzą zoptymalizowane obrazy z najlepszymi praktykami.
- Integracja z Mavenem, Gradle i Spring Boot.
- Mniej konfiguracji, szybszy start.
- Idealne dla zespołów, które chcą prostoty.

4 Porównanie krok po kroku

Krok	Dockerfile	Buildpacks
1. Przygotowanie	Piszesz Dockerfile	Używasz wtyczki (Maven/Gradle)
2. Wykrycie typu	Ręczna konfiguracja	Automatyczne wykrywanie
3. Instalacja Javy	Ty określasz	Zarządza buildpack
4. Dodanie aplikacji	Kopiujesz pliki	Automatycznie
5. Optymalizacje	Ręcznie	Wbudowane warstwy (cache)
6. Efekt końcowy	Obraz Docker	Obraz Docker (zgodny z OCI)

5 Przykładowy Dockerfile dla Spring Boot

Dockerfile

```
# Używamy lekkiego obrazu z JDK
FROM eclipse-temurin:21-jre

WORKDIR /app

# Kopiujemy plik JAR aplikacji
COPY target/app.jar app.jar

# Uruchamiamy aplikację
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

To najprostszy możliwy Dockerfile dla aplikacji Spring Boot.

6 bootBuildImage / buildpack workflow

Maven (pom.xml)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <image>
          <name>moj-aplikacja:1.0.0</name>
        </image>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Gradle (build.gradle)

```
tasks.bootBuildImage {
    imageName = "moj-aplikacja:1.0.0"
}
```

Wystarczy jedno polecenie: **mvn spring-boot:build-image** lub ./gradlew bootBuildImage

7 Kontrola vs wygoda

Dockerfile	Buildpacks
✓ Pełna kontrola nad procesem	✓ Prostota i wygoda
✓ Dowolne narzędzia i skrypty	✓ Automatyzacja
✓ Możliwość dostosowania	✓ Najlepsze praktyki
✗ Więcej konfiguracji	✓ Regularne aktualizacje
✗ Trzeba dbać o aktualizacje	✗ Mniejsza kontrola
✗ Większa odpowiedzialność	✗ Ograniczone możliwości niestandardowych zmian

8 Obraz bazowy i bezpieczeństwo

- Wybieraj minimalne obrazy (np. distroless, alpine, temurin).
- Regularnie aktualizuj obrazy bazowe.
- Skanuj obrazy pod kątem podatności (np. Trivy, Docker Scout).
- Buildpacks domyślnie używają bezpiecznych i aktualizowanych obrazów Paketo.
- Minimalny obraz = mniejsza powierzchnia ataku.
- Zadbaj o uruchamianie aplikacji jako nie-root user.

9 Cache i szybkość budowania

- Zarówno Dockerfile, jak i Buildpacks używają warstw.
- Buildpacks automatycznie dzielą obraz na warstwy (zależności, aplikacja, runtime).
- Dzięki cache kolejne buildy są znacznie szybsze.
- Zmiana kodu aplikacji nie wymusza pobierania wszystkich zależności.
- Możesz korzystać z cache w CI/CD (np. GitHub Actions, GitLab CI, Jenkins).
- Szybsze budowanie = krótszy czas dostarczania zmian.

10 Kiedy wybrać Dockerfile?

- Potrzebujesz niestandardowej konfiguracji.
- Musisz użyć specyficznych narzędzi lub bibliotek systemowych.
- Wymagasz pełnej kontroli nad obrazem.
- Masz specjalne wymagania bezpieczeństwa.
- Integracja z istniejącymi procesami i skryptami.
- Tworzysz obrazy nie tylko dla aplikacji Spring Boot.

11 Kiedy wybrać Buildpacks?

- Chcesz szybko i prosto zbudować obraz.
- Nie chcesz pisać i utrzymywać Dockerfile.
- Korzystasz ze Spring Boot i standardowych zależności.
- Zależy Ci na aktualizacjach i bezpieczeństwie.
- Pracujesz w zespole i stawiasz na konwencję.
- Chcesz skupić się na kodzie, a nie na infrastrukturze.

12 Najczęstsze pułapki

- ✗ Zbyt duże obrazy (brak minimalizacji).
- ✗ Uruchamianie jako root.
- ✗ Brak aktualizacji obrazów bazowych.
- ✗ Kopiowanie niepotrzebnych plików (np. całego projektu).
- ✗ Brak .dockerignore.
- ✗ Nieużywanie cache.
- ✗ Nieuwzględnienie zmiennych środowiskowych.
- ✗ Brak skanowania podatności.

13 Szybkie podsumowanie

- ✓ Dockerfile = pełna kontrola, większa elastyczność.
- ✓ Buildpacks = prostota, konwencja, automatyzacja.
- ✓ Oba podejścia tworzą obrazy Docker zgodne z OCI.
- ✓ Wybór zależy od potrzeb projektu i zespołu.
- ✓ Konteneryzacja ułatwia wdrażanie, skalowanie i utrzymanie aplikacji.

„Ważne nie jak budujesz, ale że dostarczasz wartość szybciej!"

Kontenery przyspieszają dobre pomysły!

SPRING BOOT

Lepsze wdrożenia tworzą lepsze produkty!

KONTENERY
AUTOMATYZACJA
BEZPIECZEŃSTWO
SKALOWANIE
SUKCES

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 94/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

DOCKER COMPOSE I SERVICE CONNECTIONS

Jak szybko uruchamiać zależności lokalnie w Spring Boot

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest Docker Compose?

Docker Compose to narzędzie do definiowania i uruchamiania wielu kontenerów za pomocą jednego pliku konfiguracyjnego (**compose.yml**).

- Ułatwia uruchamianie całego środowiska
- Definiuje usługi, sieci i wolumeny
- Działa identycznie na każdym systemie
- Idealny do developmentu i testów
- Pozwala szybko uruchomić zależności lokalnie

„Jeden plik – całe środowisko!"

2 Typowe zależności lokalne

Spring Boot często korzysta z zewnętrznych usług. Dzięki Docker Compose uruchomisz je lokalnie:

- **Postgres** — Baza danych relacyjna (np. dla aplikacji webowych)
- **Redis** — Pamięć podręczna i kolejki (cache, sesje)
- **Kafka** — Platforma do streamingu zdarzeń (np. zdarzenia domenowe)
- **RabbitMQ** — Broker wiadomości (np. komunikacja asynchroniczna)

Uruchamiaj tylko to, czego naprawdę potrzebujesz!

3 Przykładowy plik compose.yml

yaml

```yaml
version: "3.8"
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: app
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app
    ports:
      - "5432:5432"
  redis:
    image: redis:7
    ports:
      - "6379:6379"
  kafka:
    image: bitnami/kafka:3
    ports:
      - "9092:9092"
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
```

Definicja wielu usług w jednym pliku

Każda usługa ma własny obraz, porty i ustawienia

4 Idea Service Connections w Spring Boot

Service Connections (od Spring Boot 3.1) to mechanizm, który automatycznie łączy aplikację z kontenerami (np. uruchomionymi przez Docker Compose lub Testcontainers).

- Automatyczna konfiguracja połączeń
- Brak ręcznego ustawiania adresów i portów
- Wykorzystuje metadane z kontenerów
- Działa z Docker Compose i Testcontainers
- Wspiera najpopularniejsze technologie (Postgres, Redis, Kafka, RabbitMQ i inne)

5 Jak Boot rozpoznaje kontenery?

Spring Boot wykrywa uruchomione kontenery i automatycznie konfiguruje połączenia na podstawie:

- nazw usług w Docker Compose
- otwartych portów
- etykiet i metadanych kontenerów
- znanych obrazów (np. postgres, redis)
- biblioteki spring-boot-docker-compose (lub wbudowanego wsparcia)

Nie musisz pisać adresów – Boot zrobi to za Ciebie!

6 Korzyści dla developmentu i testów

- Szybsze uruchamianie środowiska
- Mniej konfiguracji i błędów
- Spójne środowisko dla całego zespołu
- Łatwe testowanie integracyjne
- Możliwość uruchomienia tylko potrzebnych usług
- Lepsza izolacja zależności
- Działa na każdym systemie (Windows, macOS, Linux)

„Więcej czasu na kodowanie, mniej na konfigurację!"

7 Integracja z Testcontainers

Testcontainers pozwala uruchamiać kontenery do testów integracyjnych. Może współpracować z Docker Compose lub uruchamiać kontenery programowo.

java

```java
@Testcontainers
@SpringBootTest
class OrderServiceTest {

  @Container
  static PostgreSQLContainer<?> postgres =
      new PostgreSQLContainer<>("postgres:16");
}
```

Kontenery dla testów integracyjnych

8 Uruchamianie i zatrzymywanie usług

▶ **Podstawowe komendy:**

bash

```bash
# Uruchom wszystkie usługi
docker compose up -d

# Zatrzymaj usługi
docker compose down

# Zobacz status
docker compose ps

# Zobacz logi
docker compose logs -f
```

Kilka prostych komend i całe środowisko działa lokalnie!

9 Typowy workflow programisty

1. Uruchom zależności: **docker compose up -d**
2. Uruchom aplikację Spring Boot
3. Pracuj i testuj lokalnie
4. W razie potrzeby sprawdź logi kontenerów
5. Po zakończeniu zatrzymaj usługi: **docker compose down**

Prosty workflow = większa produktywność!

10 Dobre praktyki

- ✓ Używaj oficjalnych obrazów (np. postgres, redis)
- ✓ Określ wersje obrazów (nie używaj "latest")
- ✓ Zapisuj dane w wolumenach
- ✓ Używaj profili (np. compose.dev.yml)
- ✓ Uruchamiaj tylko potrzebne usługi
- ✓ Dodaj healthchecki
- ✓ Trzymaj plik compose.yml w repozytorium
- ✓ Dokumentuj sposób uruchomienia

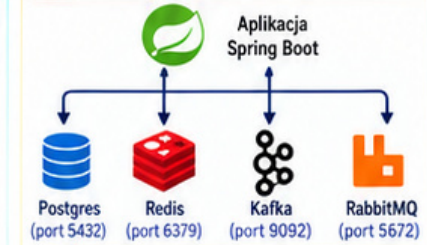
11 Najczęstsze pułapki

- ✗ Używanie obrazów "latest"
- ✗ Brak mapowania portów
- ✗ Konflikty portów z lokalnym systemem
- ✗ Brak wolumenów (utrata danych)
- ✗ Zbyt wiele niepotrzebnych usług
- ✗ Brak healthchecków (usługa nie jest gotowa)
- ✗ Ręczne konfigurowanie adresów w aplikacji
- ✗ Problemy z uprawnieniami na systemie (Windows/macOS)

12 Szybkie podsumowanie

- ✓ Docker Compose = wiele usług w jednym pliku
- ✓ Service Connections = automatyczne połączenia
- ✓ Szybszy i prostszy development
- ✓ Współpraca z Testcontainers
- ✓ Spójne środowisko dla zespołu
- ✓ Mniej konfiguracji, więcej kodu!
- ✓ Idealne do lokalnych testów i integracji
- ✓ Działa w każdym środowisku

13 Aplikacja ↔ kontenery



Jedna aplikacja, wiele możliwości! Wszystko działa lokalnie.

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

SPRING BOOT NA KUBERNETES

Probes, konfiguracja, zasoby i wdrożenie w praktyce

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 95/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Po co Kubernetes?

- Automatyzacja wdrożeń i skalowania aplikacji
- Wysoka dostępność i odporność na awarie
- Lepsze wykorzystanie zasobów infrastruktury
- Łatwe zarządzanie konfiguracją i sekretami
- To standard w chmurze i w nowoczesnych systemach

Kubernetes pozwala uruchamiać aplikacje produkcyjne w sposób bezpieczny, skalowalny i powtarzalny.

2 Podstawowe obiekty w Kubernetes

Pod	Najmniejsza jednostka. Uruchamia kontenery z aplikacją.
Deployment	Zarządza replikami Podów, umożliwia aktualizacje (rolling update).
Service	Udostępnia aplikację wewnątrz klastra lub na zewnątrz (stały adres, load balancing).
ConfigMap	Przechowuje konfigurację (nie wrażliwe dane).
Secret	Przechowuje wrażliwe dane (np. hasła, klucze, tokeny).

Te obiekty współpracują, aby zapewnić działanie, konfigurację i dostęp do Twojej aplikacji.

3 Readiness, Liveness i Startup Probes

Readiness Probe	Sprawdza, czy aplikacja jest gotowa do obsługi ruchu.
Liveness Probe	Sprawdza, czy aplikacja działa. Jeśli nie — Kubernetes ją zrestartuje.
Startup Probe	Daje czas na uruchomienie aplikacji (przydaje się dla ciężkich aplikacji).

yaml
```
livenessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
```

Używaj endpointów z Spring Boot Actuator, np. /actuator/health /actuator/ready

4 Zmienne środowiskowe i konfiguracja

- Używaj ConfigMap do zwykłej konfiguracji.
- Używaj Secret do wrażliwych danych.
- Stosuj zmienne środowiskowe (env) w aplikacji.
- Możesz też korzystać z plików konfiguracyjnych (np. application.yml) z ConfigMap.
- W Spring Boot preferuj @Value lub @ConfigurationProperties.

yaml
```
env:
  - name: SPRING_PROFILES_ACTIVE
    value: prod
  - name: DB_URL
    valueFrom:
      secretKeyRef:
        name: app-secret
        key: db-url
```

5 Zasoby CPU/RAM i requests/limits

- requests – minimalne zasoby potrzebne do uruchomienia kontenera.
- limits – maksymalne zasoby, jakie może użyć.
- Dzięki temu Kubernetes lepiej planuje zasoby na węzłach.
- Ustawiaj realne wartości — unikniesz problemów z wydajnością.
- Monitoruj użycie zasobów i dostosowuj je w czasie.

yaml
```
resources:
  requests:
    cpu: "250m"
    memory: "512Mi"
  limits:
    cpu: "500m"
    memory: "1024Mi"
```

6 Rolling update i skalowanie

- Deployment umożliwia bezpieczne wdrażanie nowych wersji (rolling update).
- Aplikacja jest aktualizowana stopniowo bez przestoju.
- Możesz łatwo skalować liczbę replik.
- Używaj HPA (Horizontal Pod Autoscaler) do automatycznego skalowania.
- Sprawdzaj status wdrożeń komendą kubectl.

yaml
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
```

7 Ingress / dostęp HTTP

- Ingress udostępnia aplikację na zewnątrz klastra (HTTP/HTTPS).
- Obsługuje routing po ścieżkach i domenach.
- Wymaga Ingress Controller (np. NGINX, Traefik).
- Możesz skonfigurować TLS (z certyfikatami).
- Umożliwia dostęp do wielu usług przez jeden adres.

yaml
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
spec:
  rules:
    - host: api.twojadomena.pl
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-app-service
                port:
                  number: 8080
```

8 Logi i observability

- Logi z kontenerów możesz przeglądać przez kubectl logs.
- W produkcji używaj centralnego systemu logowania (np. ELK, Loki).
- Monitoruj metryki z Actuator (np. Prometheus + Grafana).
- Dodawaj tracing (np. OpenTelemetry).
- Obserwuj stan aplikacji i infrastruktury.

Prometheus · Grafana · Loki · Jaeger

9 Graceful shutdown

- Spring Boot obsługuje graceful shutdown (kończące się żądania).
- Ustaw odpowiedni czas zakończenia (terminationGracePeriodSeconds).
- Kubernetes wyśle sygnał SIGTERM, a aplikacja zakończy pracę.
- Używaj endpointu /actuator/shutdown tylko w razie potrzeby.
- Sprawdź, czy połączenia z bazą i kolejkami są poprawnie zamykane.

Dzięki graceful shutdown unikniesz utraty danych i błędów podczas wdrażania nowych wersji.

10 Dobre praktyki

- Używaj Health Checków i odpowiednich probe'ów.
- Trzymaj konfigurację w ConfigMap/Secret.
- Ustawiaj requests i limits.
- Stosuj rolling update i HPA.
- Monitoruj logi, metryki i tracing.
- Używaj niezmiennych obrazów (np. z tagiem wersji).
- Dbaj o bezpieczeństwo (RBAC, Network Policies).
- Automatyzuj wdrożenia (CI/CD).
- Testuj na środowisku zbliżonym do produkcji.

11 Najczęstsze pułapki

- ✗ Brak lub źle skonfigurowane probe'y.
- ✗ Zbyt małe lub zbyt duże zasoby.
- ✗ Używanie zmiennych środowiskowych dla wrażliwych danych (zamiast Secret).
- ✗ Brak limitów zasobów.
- ✗ Niepoprawna konfiguracja Ingress.
- ✗ Brak observability (logi, metryki, tracing).
- ✗ Ręczne wdrożenia zamiast automatyzacji.
- ✗ Ignorowanie zasad bezpieczeństwa.
- ✗ Testowanie tylko lokalnie.

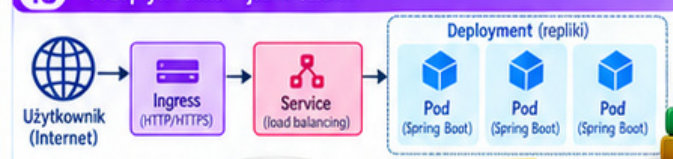
Unikaj tych błędów, aby Twoja aplikacja działała stabilnie i bezpiecznie!

12 Szybkie podsumowanie

- Kubernetes = automatyzacja, skalowanie, niezawodność.
- Kluczowe obiekty: Pod, Deployment, Service, ConfigMap, Secret.
- Używaj probe'ów, zasobów, Ingress i monitoringu.
- Dbaj o dobre praktyki i unikaj typowych błędów.
- Spring Boot świetnie integruje się z ekosystemem Kubernetes.

Kubernetes pozwala skupić się na rozwoju aplikacji, a nie na infrastrukturze!

13 Przepływ ruchu — jak to działa?





Technologia + dobre praktyki = wielkie możliwości!

Małe kroki ≤ duże systemy!



Dobre decyzje tworzą lepsze systemy!

100 MAP MYŚLI ZE SPRING BOOTA
Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

MODUŁ 10
BUILD, DOCKER, CLOUD
I PRODUKCJA

MAPA 95/100
Krok po kroku do eksperta!

Pisz testy
Buduj lepsze jutro!

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 96/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

HTTPS, TLS I SSL BUNDLES

Jak bezpiecznie skonfigurować szyfrowanie i certyfikaty

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Po co HTTPS?

- Szyfruje komunikację między klientem a serwerem.
- Chroni dane przed podsłuchem i modyfikacją.
- Buduje zaufanie użytkowników.
- Jest wymagany w wielu przeglądarkach i przez przepisy (np. RODO).
- Niezbędny w produkcji – standard w nowoczesnych aplikacjach.

„Bez HTTPS nie ma bezpiecznego Internetu!"

2 Czym są TLS i certyfikat?

TLS (Transport Layer Security) to protokół szyfrujący komunikację. Jest następcą SSL (starsza wersja).

Certyfikat cyfrowy to plik z kluczem publicznym i danymi wystawcy, potwierdzający tożsamość serwera (lub klienta).

Certyfikat zawiera m.in.:

- nazwę domeny (CN / SAN)
- klucz publiczny
- okres ważności
- podpis wystawcy (CA)

3 Handshake TLS (wysoki poziom)

1. Klient nawiązuje połączenie i wysyła listę obsługiwanych wersji TLS i szyfrów.
2. Serwer odpowiada certyfikatem i wyborem parametrów szyfrowania.
3. Klient weryfikuje certyfikat i tworzy klucz sesji.
4. Obie strony wymieniają klucze i uzgadniają szyfrowanie.
5. Od tego momentu cała komunikacja jest zaszyfrowana.

„Po udanym handshake dane są szyfrowane symetrycznie – szybko i bezpiecznie!".

4 Key store i trust store

Key store (np. JKS, PKCS12) przechowuje prywatny klucz i certyfikat serwera (nasz certyfikat).

Trust store przechowuje zaufane certyfikaty innych podmiotów (np. certyfikaty urzędów CA), używane do weryfikacji certyfikatów serwerów.

Key store = nasza tożsamość
Trust store = komu ufamy

5 Podstawowa konfiguracja HTTPS w Spring Boot

Najprostsza konfiguracja z plikiem PKCS12 i ustawieniami w application.yml:

```yaml
server:
  port: 8443
  ssl:
    enabled: true
    key-store: classpath:keystore.p12
    key-store-password: zmienTo!
    key-store-type: PKCS12
    key-alias: tomcat
```

Aplikacja będzie dostępna przez https://localhost:8443

6 Czym są SSL bundles?

SSL bundles (od Spring Boot 3.1) to prostszy sposób zarządzania certyfikatami i kluczami w aplikacji.

- Pozwalają zdefiniować zestawy certyfikatów (bundle) w jednym miejscu.
- Ułatwiają konfigurację klientów i serwerów.
- Wspierają różne formaty (JKS, PKCS12, PEM).
- Można używać wielu bundle'i (np. dla różnych usług).
- Integrują się z auto-konfiguracją Spring Boot.

„SSL bundles = mniej konfiguracji, więcej porządku!"

7 Konfiguracja klienta i serwera

Serwer (własny certyfikat):

```yaml
server:
  ssl:
    bundle: server
```

Klient (np. RestClient / WebClient):

```yaml
spring:
  ssl:
    bundle: client
```

Bundle zawiera informacje o key store, trust store i hasłach.

8 Certyfikaty self-signed vs produkcyjne

Self-signed (testowe)	Produkcyjne (zaufane CA)
Tworzone samodzielnie	Wystawiane przez zaufane urzędy (CA)
Brak zaufanego wystawcy	Automatycznie zaufane w przeglądarkach
Dobre do środowisk deweloperskich	Wymagane w produkcji
Przeglądarki wyświetlają ostrzeżenia	Mogą być płatne (np. Let's Encrypt – darmowe)
Szybkie i darmowe	Automatyczne odnawianie (np. ACME)

Do produkcji używaj certyfikatów od zaufanych dostawców (np. Let's Encrypt).

9 Typowe błędy certyfikatów

- Niepoprawna nazwa domeny (w CN lub SAN).
- Wygasły certyfikat.
- Nieprawidłowy łańcuch certyfikatów (brak certyfikatów pośrednich).
- Nieufany wystawca (self-signed w produkcji).
- Niezgodność nazwy hosta (Hostname verification).
- Nieprawidłowe hasło do key store.
- Zły format pliku (JKS / PKCS12 / PEM).
- Brak wymaganych uprawnień do pliku certyfikatu.

10 Dobre praktyki bezpieczeństwa

- Używaj TLS 1.2 lub nowszego (najlepiej 1.3).
- Stosuj silne szyfry i wyłącz słabe algorytmy.
- Automatycznie odnawiaj certyfikaty (np. Let's Encrypt + ACME).
- Przechowuj klucze w bezpieczny sposób (np. Kubernetes Secrets, Vault).
- Używaj różnych certyfikatów dla środowisk.
- Włącz weryfikację nazwy hosta po stronie klienta.
- Regularnie aktualizuj zależności.
- Monitoruj daty ważności certyfikatów.

„Bezpieczeństwo to proces, nie jednorazowa konfiguracja."

11 Najczęstsze pułapki

- Używanie self-signed w produkcji.
- Wyłączanie weryfikacji certyfikatów ("trustAll", "disableVerification").
- Nieprawidłowa konfiguracja bundle'i.
- Brak certyfikatów pośrednich.
- Niezgodność protokołów TLS.
- Problemy z odnowieniem certyfikatów.
- Twardo zakodowane hasła w kodzie.
- Różne certyfikaty dla tej samej domeny w różnych usługach.
- Brak monitoringu i alertów.

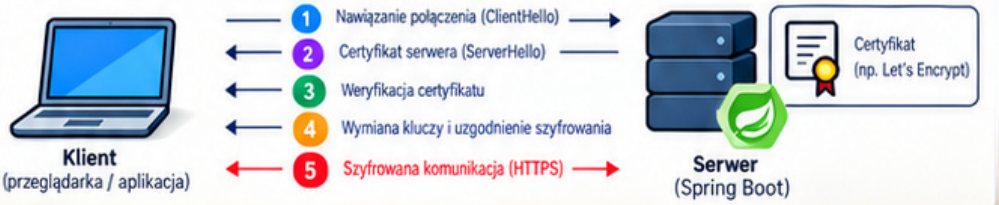
„Nie wyłączaj weryfikacji certyfikatów – to pozorna wygoda, realne ryzyko!"

12 Szybkie podsumowanie

- HTTPS szyfruje i chroni dane.
- TLS to nowoczesny protokół (sukcesor SSL).
- Certyfikat potwierdza tożsamość.
- SSL bundles upraszczają konfigurację.
- W Spring Boot łatwo skonfigurujesz HTTPS dla serwera i klienta.
- Używaj zaufanych certyfikatów w produkcji.
- Przestrzegaj dobrych praktyk i unikaj typowych błędów.

„Szyfrowanie to podstawa bezpiecznych systemów."

13 Schemat: klient ↔ serwer TLS



BEZPIECZNIE DO PRODUKCJI!

- Szyfruj
- Monitoruj
- Odnawiaj
- Zadbaj o użytkowników



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 97/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

GRAALVM NATIVE IMAGE I AOT

Szybszy start aplikacji i mniejsze zużycie pamięci

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Czym jest Native Image?

- Native Image to technologia GraalVM, która kompiluje aplikację Java do samodzielnego, natywnego pliku wykonywalnego.
- Aplikacja nie wymaga uruchamiania maszyny wirtualnej JVM.
- Zawiera tylko niezbędne klasy i zasoby, dzięki czemu jest mała i szybka.
- Idealna do kontenerów, środowisk chmurowych i mikroserwisów.

2 Czym jest AOT (Ahead-of-Time)?

- AOT to proces przygotowania aplikacji do uruchomienia bez dynamicznych operacji w czasie działania.
- Spring AOT analizuje kod podczas budowania i generuje dodatkowy kod oraz konfigurację.
- Eliminuje potrzebę refleksji i dynamicznego ładowania klas.
- Jest kluczowym elementem przy budowaniu natywnych obrazów (GraalVM Native Image).

3 Korzyści: startup, RAM, cold start

- Bardzo szybki start aplikacji (od kilkudziesięciu milisekund).
- Niższe zużycie pamięci RAM (nawet 3–10x mniej).
- Krótszy czas cold start w środowiskach serverless i chmurowych.
- Mniejszy rozmiar obrazu Docker.
- Lepsza skalowalność i gęstość instancji.

4 Ograniczenia

- Nie wszystkie biblioteki są obsługiwane.
- Ograniczona obsługa refleksji, ładowania klas dynamicznie.
- Niektóre funkcje JVM są niedostępne (np. dynamiczne agenty, JIT).
- Większy czas budowania obrazu.
- Wymaga odpowiedniej konfiguracji (hinty, zasoby, proxy).
- Może wymagać zmian w kodzie lub użycia alternatywnych bibliotek.

5 Jak Spring Boot wspiera native build?

- Spring Boot 3+ ma wbudowane wsparcie dla AOT i GraalVM.
- Automatycznie generuje niezbędne konfiguracje (refleksja, zasoby, proxy).
- Udostępnia wtyczki do budowania natyywnego obrazu (Maven/Gradle).
- Startery są dostosowane do środowiska natyywnego.
- Dostępne są narzędzia diagnostyczne i testy zgodności.

6 Przykładowy workflow budowania obrazu natywnego

1. Przygotuj projekt (Spring Boot 3+).
2. Dodaj wtyczkę native do Mavena lub Gradle.
3. Uruchom budowanie natywnego obrazu.
4. Przetestuj lokalnie.
5. Zbuduj obraz Docker (opcjonalnie).
6. Wdróż do środowiska docelowego.

Maven: bash

```bash
# budowanie natywnego obrazu
./mvnw -Pnative native:compile
```

7 Reflection, zasoby i proxy (hinty)

- Native Image wymaga jawnej deklaracji elementów używanych dynamicznie.
- Spring AOT generuje hinty automatycznie, ale czasem trzeba dodać własne.
- Obsługiwane są hinty dla:
  - refleksji (reflection)
  - zasobów (resources)
  - proxy (dynamic proxies)
- Można użyć adnotacji lub plików konfiguracyjnych JSON.

Przykład hintu: java

```java
@RegisterReflectionForBinding(
   MyClass.class
)
```

8 Kiedy warto użyć?

- Mikroserwisy i architektura chmurowa (Kubernetes, serverless).
- Aplikacje o krótkim czasie życia (np. funkcje serverless).
- Wymagania dotyczące niskiego zużycia zasobów.
- Szybkie skalowanie i częste uruchamianie nowych instancji.
- Aplikacje działające w kontenerach, na edge lub w środowiskach z ograniczonymi zasobami.

Idealne dla nowoczesnych, skalowalnych aplikacji!

9 Kiedy nie warto?

- Aplikacje silnie korzystające z refleksji i dynamicznego ładowania klas.
- Biblioteki bez wsparcia dla GraalVM.
- Aplikacje wymagające agentów JVM (np. monitoring, profiling).
- Duże, monolityczne aplikacje z wieloma zależnościami.
- Gdy czas budowania jest krytyczny.
- Projekty, w których nie ma realnej potrzeby szybszego startu ani mniejszego zużycia pamięci.

Nie zawsze native image jest najlepszym wyborem!

10 Dobre praktyki

- Używaj Spring Boot 3+ i aktualnych wersji GraalVM.
- Testuj aplikację w trybie natywnym.
- Dodawaj tylko niezbędne hinty.
- Unikaj nieobsługiwanych bibliotek.
- Monitoruj rozmiar obrazu i czas budowania.
- Używaj kontenerów Docker do testów.
- Sprawdzaj dokumentację bibliotek.
- Stosuj profile i konfigurację środowiskową.

Małe, proste i dobrze przetestowane aplikacje dają najlepsze rezultaty.

11 Najczęstsze pułapki

- Brak hintów dla refleksji, zasobów lub proxy.
- Użycie bibliotek niekompatybilnych z GraalVM.
- Błędy podczas inicjalizacji klas.
- Nieoczywiste problemy z zasobami (np. pliki konfiguracyjne, szablony).
- Zbyt duży rozmiar obrazu.
- Długi czas budowania.
- Różnice w zachowaniu w porównaniu do uruchomienia na JVM.
- Brak testów w środowisku natywnym.

Testuj, testuj i jeszcze raz testuj w środowisku natywnym!

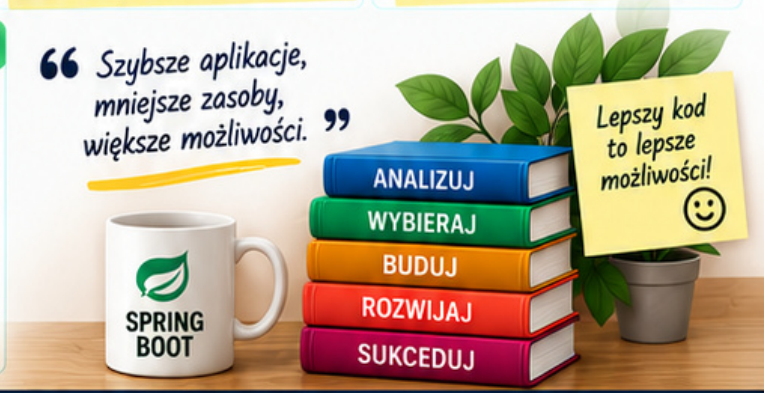
12 Szybkie podsumowanie

- Native Image = natywny plik wykonywalny bez JVM.
- AOT przygotowuje aplikację do uruchomienia natywnego.
- Szybszy start, mniejsze zużycie pamięci, lepszy cold start.
- Wymaga konfiguracji i świadomości ograniczeń.
- Świetnie sprawdza się w mikroserwiach, kontenerach i chmurze.
- Nie zawsze jest najlepszym wyborem.

Wybieraj świadomie i buduj lepsze aplikacje!

13 Porównanie JVM vs Native Image

Cecha	JVM (tradycyjna aplikacja)	Native Image (GraalVM)
Czas startu	Sekundy	Milisekundy
Zużycie pamięci RAM	Większe	Mniejsze (nawet 3–10x)
Rozmiar obrazu	Większy	Mniejszy
Wymaga JVM	Tak	Nie
Wsparcie bibliotek	Pełne	Ograniczone
Czas budowania	Szybki	Dłuższy
Zastosowania	Uniwersalne	Mikroserwisy, chmura, serverless



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 98/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

OPTYMALIZACJA APLIKACJI SPRING BOOT

Start, pamięć, baza danych, HTTP i wydajność w praktyce

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Jak myśleć o optymalizacji?

- Optymalizacja to proces, nie jednorazowa akcja. Zaczynaj od pomiarów i realnych problemów, nie od domysłów.
- Skup się na wąskich gardłach (bottleneckach).
- Najpierw zrozum, potem optymalizuj.
- Małe zmiany mogą przynieść duże efekty.
- Równowaga: wydajność, czytelność, bezpieczeństwo i koszty.

„Nie optymalizuj na zapas – optymalizuj to, co naprawdę ma znaczenie."

2 Czas uruchamiania (Startup Time)

- [x] Używaj Spring Boot 3 i najnowszych wersji.
- [x] Ogranicz auto-konfigurację (tylko to, co potrzebne).
- [x] Rozważ AOT (Ahead-of-Time) i native image.
- [x] Leniwa inicjalizacja (spring.main.lazy-initialization).
- [x] Minimalizuj skanowanie komponentów.
- [x] Usuń niepotrzebne zależności.

application.properties

```
spring.main.lazy-initialization=true
spring.main.banner-mode=off
management.endpoints.web.exposure.include=health,info
```

Szybszy start = krótszy czas wdrożenia i lepsze skalowanie.

3 Pamięć i JVM

- [x] Dobierz odpowiedni rozmiar sterty (-Xms, -Xmx).
- [x] Używaj nowoczesnych kolektorów śmieci (np. G1GC).
- [x] Unikaj wycieków pamięci (memory leaks).
- [x] Monitoruj zużycie pamięci (np. Actuator, VisualVM).
- [x] Stosuj niewielkie, niemutowalne obiekty.
- [x] Uważaj na cache bez limitów.

JVM options

```
-Xms512m
-Xmx1024m
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
```

Dopasuj pamięć do obciążenia, a nie na wyrost!

4 Pule połączeń i baza danych

- [x] Używaj HikariCP (domyślnie w Spring Boot).
- [x] Odpowiednio ustaw rozmiar puli (maximumPoolSize).
- [x] Optymalizuj zapytania i używaj indeksów.
- [x] Unikaj N+1 i niepotrzebnego pobierania danych.
- [x] Stosuj paginację i stronicowanie wyników.
- [x] Monitoruj czas wykonywania zapytań.

application.properties

```
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.connection-timeout=30000
```

5 HTTP client/server tuning

- [x] Wybierz odpowiedni serwer (Tomcat, Jetty, Undertow).
- [x] Dostosuj liczbę wątków i połączeń.
- [x] Stosuj timeouts (connect, read, write).
- [x] Używaj connection pooling dla klientów HTTP.
- [x] Kompresuj odpowiedzi (gzip, brotli).
- [x] Unikaj dużych payloadów i zbędnych danych.

application.properties

```
server.tomcat.threads.max=200
server.tomcat.connection-timeout=20000
server.compression.enabled=true
server.compression.mime-types=application/json
```

6 Cache (pamięć podręczna)

- [x] Używaj Spring Cache (np. Caffeine, Redis).
- [x] Cache'uj wyniki kosztownych operacji.
- [x] Ustaw sensowne TTL i limity rozmiaru.
- [x] Unikaj cache'owania danych, które często się zmieniają.
- [x] Pamiętaj o strategii unieważniania (eviction).
- [x] Monitoruj skuteczność cache (hit/miss ratio).

Przykład z Caffeine

```
@Cacheable(value = "produkty", key = "#id")
public Produkt znajdźPoId(Long id) {
    return repository.findById(id)
        .orElseThrow();
}
```

7 Logowanie i observability a wydajność

- [x] Używaj odpowiednich poziomów logowania.
- [x] Unikaj logowania w pętlach i dużych obiektów.
- [x] Stosuj logowanie asynchroniczne (np. Logback Async).
- [x] Włącz tylko potrzebne endpoints Actuator.
- [x] Używaj metryk (Micrometer, Prometheus).
- [x] Tracing (np. OpenTelemetry) – z rozwagą.

Dobre observability pomaga znaleźć problemy, ale zbyt duża ilość danych może spowolnić aplikację.

8 Blokowanie i wątki

- [x] Unikaj operacji blokujących w wątkach aplikacji.
- [x] Używaj programowania reaktywnego tam, gdzie to ma sens (WebFlux).
- [x] Stosuj asynchroniczność (CompletableFuture, @Async, MessageQueue).
- [x] Nie blokuj wirtualnych wątków (Project Loom).
- [x] Rozważ oddzielne pule wątków dla różnych zadań.
- [x] Monitoruj kolejki i czas oczekiwania.

Blokowanie = mniejsza przepustowość. Działaj asynchronicznie tam, gdzie to możliwe.

9 Profiling i pomiary

- [x] Używaj narzędzi: JFR, VisualVM, YourKit, async-profiler.
- [x] Mierz czas odpowiedzi (response time).
- [x] Analizuj zużycie CPU, pamięci i GC.
- [x] Wykorzystuj benchmarki (np. JMH).
- [x] Testuj pod realistycznym obciążeniem (np. Gatling, k6).
- [x] Porównuj wyniki przed i po zmianach.

Dane, nie opinie! Mierz, analizuj, wyciągaj wnioski.

10 Dobre praktyki

- [x] Używaj aktualnych wersji Spring Boot.
- [x] Regularnie usuwaj nieużywane zależności.
- [x] Dbaj o proste i czytelne rozwiązania.
- [x] Stosuj profilowanie w środowisku zbliżonym do produkcji.
- [x] Automatyzuj testy wydajnościowe.
- [x] Monitoruj aplikację po wdrożeniu.

Proste rozwiązania są często najszybsze i najtrwalsze.

11 Najczęstsze pułapki

- [x] Optymalizacja bez pomiarów.
- [x] Zbyt duża lub zbyt mała sterta JVM.
- [x] Brak indeksów w bazie danych.
- [x] N+1 queries i nieoptymalne zapytania.
- [x] Zbyt agresywne cache'owanie.
- [x] Logowanie dużych obiektów.
- [x] Blokowanie wątków i długie operacje I/O.
- [x] Brak monitoringu w produkcji.
- [x] Przedwczesna optymalizacja.
- [x] Ignorowanie kosztów chmury.

12 Szybkie podsumowanie

- [x] Szybszy start dzięki minimalnej Konfiguracji.
- [x] Stabilna pamięć i dobry dobór JVM.
- [x] Wydajna baza danych i pule połączeń.
- [x] Odpowiednie ustawienia serwera i klienta HTTP.
- [x] Inteligentne cache'owanie.
- [x] Rozsądne logowanie i pełna observability.
- [x] Unikanie blokowania i dobre zarządzanie wątkami.
- [x] Regularne profilowanie i pomiary.
- [x] Dobre praktyki i ciągłe doskonalenie.

13 Najpierw mierz, potem optymalizuj

1. Zdefiniuj cel (np. szybszy start, niższe opóźnienia, mniejsze koszty).
2. Zmierz aktualny stan (metryki, profiling).
3. Zidentyfikuj wąskie gardła.
4. Wprowadź jedną zmianę na raz.
5. Ponownie zmierz i porównaj wyniki.
6. Powtarzaj proces i wyciągaj wnioski.

„Mierz. Analizuj. Optymalizuj. Powtarzaj. Sukces przychodzi z danych."

Wydajna aplikacja to zadowoleni użytkownicy!



*Lepszy kod
Działa szybciej
Ma większy sens!*



100 MAP MYŚLI ZE SPRING BOOTA
Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

MODUŁ 10
BUILD, DOCKER, CLOUD
I PRODUKCJA

MAPA 98/100
Krok po kroku do eksperta!

Pisz testy
Buduj lepsze jutro!

100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 99/100

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MIGRACJA SPRING BOOT 3.5 → 4.x

Co sprawdzić przed aktualizacją i gdzie czyhają zmiany

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Po co migrować do 4.x?

- [x] Nowe funkcje i usprawnienia w Spring Boot i Spring Framework.
- [x] Dłuższe wsparcie i aktualizacje zabezpieczeń.
- [x] Lepsza wydajność i stabilność.
- [x] Zgodność z najnowszym ekosystemem (chmura, narzędzia, biblioteki).
- [x] To naturalny krok rozwoju – unikasz narastającego długu technologicznego.

2 Wymagania Java 17+

- [x] Spring Boot 4.x wymaga Javy 17 lub nowszej.
- [x] Sprawdź wersję JDK w swoim środowisku (lokalnie, CI/CD, produkcja).
- [x] Upewnij się, że wszystkie biblioteki są zgodne z Javą 17+.
- [x] Rozważ aktualizację do Javy 21 (LTS) dla dłuższego wsparcia.

Java 17+ to podstawa dla Spring Boot 4.x

3 Przegląd zależności i starterów

- [x] Zaktualizuj wersję spring-boot-starter-parent lub użyj BOM dla Spring Boot 4.x.
- [x] Sprawdź zgodność wszystkich zależności (np. Spring Cloud, Spring Security, Spring Data).
- [x] Usuń nieużywane lub przestarzałe biblioteki.
- [x] Zwróć uwagę na biblioteki zewnętrzne, które mogą nie być jeszcze zgodne z 4.x.
- [x] Sprawdź dostępność nowych wersji starterów.

Maven (przykład) XML

```xml
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>4.0.0</version>
</parent>
```

4 Spring Framework 7 i konsekwencje

- [x] Spring Boot 4.x opiera się na Spring Framework 7.
- [x] Sprawdź zmiany w API i deprecacje.
- [x] Niektóre klasy i metody mogły zostać przeniesione lub usunięte.
- [x] Zwróć uwagę na zmiany w obsłudze konfiguracji, IOC i AOP.
- [x] Przeanalizuj oficjalne notatki migracyjne (Spring Release Notes).

5 Jackson 3

- [x] Spring Boot 4.x używa Jackson 3.
- [x] Sprawdź niezgodności w API w porównaniu do Jackson 2.
- [x] Zweryfikuj niestandardowe serializery i deserializery.
- [x] Sprawdź ustawienia w konfiguracji i ewentualne zmiany domyślnego zachowania.
- [x] Przetestuj serializację/deserializację dla kluczowych modeli.

6 Przegląd konfiguracji i właściwości

- [x] Sprawdź zmiany w nazwach właściwości w plikach *application.properties/yaml*.
- [x] Zidentyfikuj zdeprecjonowane lub usunięte właściwości.
- [x] Porównaj swoją konfigurację z nowymi domyślnymi wartościami.
- [x] Sprawdź konfigurację Actuatora, logowania, serwera i obsługi błędów.
- [x] Użyj mechanizmów raportowania deprecacji (warnowania podczas startu aplikacji).

7 Testy regresyjne i observability

- [x] Uruchom pełen zestaw testów automatycznych.
- [x] Sprawdź kluczowe scenariusze biznesowe.
- [x] Zweryfikuj metryki, logi i trasy (tracing) po migracji.
- [x] Upewnij się, że endpointy Actuatora działają poprawnie.
- [x] Porównaj zachowanie aplikacji przed i po migracji (wydajność, błędy, logi).

8 Plan bezpiecznej migracji

1. Przeanalizuj zmiany i przygotuj środowisko.
2. Zaktualizuj JDK, build i zależności.
3. Dostosuj kod i konfigurację.
4. Uruchom testy i popraw błędy.
5. Przetestuj w środowisku zbliżonym do produkcji.
6. Wdróż stopniowo (np. blue/green, canary).

9 Najczęstsze obszary ryzyka

- [x] Niezgodne biblioteki z ekosystemu (np. Spring Cloud, bazy danych, klienty).
- [x] Zmiany w konfiguracji i nazwach właściwości.
- [x] Problemy z serializacją (Jackson 3).
- [x] Deprecacje i usunięte API w Spring Framework 7.
- [x] Niespodziewane zmiany w zachowaniu autokonfiguracji.
- [x] Różnice w domyślnych ustawieniach (dotyczące bezpieczeństwa, logowania, CORS).

10 Dobre praktyki

- [x] Czytaj oficjalne notatki migracyjne.
- [x] Aktualizuj etapami, a nie „wszystko naraz".
- [x] Utrzymuj pokrycie testami.
- [x] Używaj BOM i zarządzaj wersjami zależności.
- [x] Testuj w środowisku jak najbardziej zbliżonym do produkcji.
- [x] Monitoruj aplikację po wdrożeniu.
- [x] Dokumentuj wprowadzone zmiany.

11 Najczęstsze pułapki

- [x] Pominięcie testów integracyjnych.
- [x] Używanie niekompatybilnych bibliotek.
- [x] Ignorowanie ostrzeżeń o deprecacjach.
- [x] Założenie, że wszystko zadziała bez zmian.
- [x] Brak testów wydajności i obserwowalności.
- [x] Zbyt szybkie wdrożenie na produkcję.
- [x] Niedoszacowanie czasu potrzebnego na migrację.

12 Szybkie podsumowanie

- [x] Spring Boot 4.x wymaga Javy 17+.
- [x] Sprawdź zależności, konfigurację i zmiany w API.
- [x] Zwróć uwagę na Jackson 3 i Spring Framework 7.
- [x] Uruchom testy, sprawdź observability.
- [x] Migruj etapami i monitoruj po wdrożeniu.

„Dobra migracja to nie sprint, to dobrze zaplanowana podróż."

13 Checklista przed wdrożeniem

- [x] Java 17+ skonfigurowana we wszystkich środowiskach.
- [x] Zależności zaktualizowane i zgodne z 4.x.
- [x] Konfiguracja zweryfikowana (brak deprecacji).
- [x] Wszystkie testy przechodzą.
- [x] Metryki, logi i tracing działają poprawnie.
- [x] Testy wydajnościowe wykonane.
- [x] Środowisko produkcyjne przygotowane.
- [x] Plan wdrożenia i rollbacku gotowy.

Sprawdź, przetestuj, wdroż z pewnością!



100 MAP MYŚLI ZE SPRING BOOTA

MODUŁ 10 — BUILD, DOCKER, CLOUD I PRODUKCJA

SPRING BOOT PRODUCTION CHECKLIST

Od konfiguracji po monitoring — ostatnia kontrola przed produkcją

Pisz testy
Buduj pewniej
Rozwijaj szybciej

MAPA 100/100

Lepsze aplikacje
Lepsza przyszłość!
Większa pewność!

1 Konfiguracja i profile

- [x] Używaj profili: dev, test, prod
- [x] Przechowuj konfigurację poza kodem (np. application-prod.yml)
- [x] Uważaj na wrażliwe dane (sekrety w zmiennych środowiskowych)
- [x] Stosuj @ConfigurationProperties
- [x] Waliduj konfigurację przy starcie
- [x] Używaj Feature Togglerów

```yaml
# application-prod.yml
spring:
  profiles: prod
  datasource:
    url: ${DB_URL}
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
```

2 Security (bezpieczeństwo)

- [x] Skonfiguruj Spring Security
- [x] Używaj JWT lub OAuth2/OIDC
- [x] Zabezpiecz endpointy (role, uprawnienia)
- [x] Szyfruj wrażliwe dane
- [x] Stosuj HTTPS (TLS)
- [x] Zabezpiecz przed atakami (CSRF, XSS, SQL Injection)
- [x] Regularnie aktualizuj zależności
- [x] Używaj narzędzi do skanowania podatności (np. OWASP)

```java
http
  .authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()
);
```

3 Baza danych i migracje

- [x] Używaj migracji (Flyway lub Liquibase)
- [x] Wersjonuj schemat bazy danych
- [x] Testuj migracje na środowisku zbliżonym do produkcji
- [x] Dbaj o indeksy i wydajność zapytań
- [x] Stosuj kopie zapasowe bazy
- [x] Monitoruj rozmiar i wydajność
- [x] Używaj transakcji i sensownych poziomów izolacji
- [x] Planuj strategię odtwarzania danych

```sql
# Flyway - przykład migracji
CREATE TABLE users (
  id BIGINT PRIMARY KEY,
  username VARCHAR(100) NOT NULL,
  email VARCHAR(255) NOT NULL
);
```

4 API i obsługa błędów

- [x] Projektuj spójne REST API
- [x] Używaj właściwych kodów HTTP
- [x] Zwracaj czytelne komunikaty błędów
- [x] Stosuj globalne obsługi wyjątków (@ControllerAdvice)
- [x] Waliduj dane wejściowe (@Valid)
- [x] Dokumentuj API (OpenAPI/Swagger)
- [x] Nie ujawniaj poufnych informacji w komunikatach błędów
- [x] Obsługuj błędy w sposób spójny

```java
@ControllerAdvice
public class GlobalExceptionHandler {
  @ExceptionHandler(Exception.class)
  public ResponseEntity<ErrorResponse>
    handle(Exception ex) { ... }
}
```

5 Testy

- [x] Pisz testy jednostkowe (JUnit, Mockito)
- [x] Pisz testy integracyjne (@SpringBootTest)
- [x] Testuj warstwę web (MockMvc)
- [x] Używaj testcontainers do baz danych
- [x] Utrzymuj dobre pokrycie kodu testami
- [x] Ustal pipeline z automatycznym uruchamianiem testów
- [x] Testuj scenariusze błędów
- [x] Dbaj o szybkie i stabilne testy

6 Observability (monitoring)

- [x] Używaj Actuator (/actuator)
- [x] Monitoruj metryki (Prometheus + Grafana)
- [x] Dodaj health checki i readiness/liveness
- [x] Śledź requesty (tracing – Micrometer/OTel)
- [x] Monitoruj logi i błędy (np. ELK/EFK)
- [x] Ustaw alerty na kluczowe zdarzenia
- [x] Monitoruj zasoby (CPU, RAM, dysk)
- [x] Sprawdzaj dostępność aplikacji (uptime)

7 Logowanie

- [x] Używaj logowania strukturalnego (np. JSON)
- [x] Określ odpowiedni poziom logów
- [x] Nie loguj wrażliwych danych (hasła, tokeny, dane osobowe)
- [x] Używaj korelacji requestów (traceId)
- [x] Przechowuj logi centralnie (ELK/Loki)
- [x] Ustalemia rotacji i retencji logów
- [x] Monitoruj błędy i nietypowe zdarzenia
- [x] Dbaj o czytelne i pomocne komunikaty

```json
{
  "timestamp": "2024-01-01T10:00:002",
  "level": "INFO",
  "message": "Zamówienie utworzone",
  "traceId": "abc123",
  "userId": "42"
}
```

8 Kontenery / deployment

- [x] Używaj Dockera (multi-stage build)
- [x] Twórz lekkie obrazy (np. Eclipse Temurin)
- [x] Nie uruchamiaj aplikacji jako root
- [x] Używaj pliku .dockerignore
- [x] Konfiguruj zmienne środowiskowe
- [x] Stosuj orkiestrację (Kubernetes, Docker Compose)
- [x] Zapewnij zero-downtime deployment
- [x] Automatyzuj build i deployment (CI/CD)

```dockerfile
# Dockerfile (multi-stage)
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY . .
RUN ./mvnw clean package
FROM eclipse-temurin:21-jre
COPY --from=build /app/target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

9 Wydajność

- [x] Monitoruj czas odpowiedzi (latencję)
- [x] Optymalizuj zapytania do bazy danych
- [x] Stosuj cache (np. Caffeine, Redis)
- [x] Używaj paginacji i strumieniowania
- [x] Unikaj niepotrzebnych obliczeń
- [x] Profiluj aplikację (JFR, VisualVM)
- [x] Testuj obciążeniowo (np. JMeter, k6)
- [x] Planuj skalowanie (poziome i pionowe)

10 Backup i odzyskiwanie

- [x] Regularnie wykonuj kopie zapasowe
- [x] Testuj proces odtwarzania danych
- [x] Przechowuj backupy w bezpiecznym miejscu (np. chmura, inna lokalizacja)
- [x] Ustal politykę retencji
- [x] Backupuj nie tylko bazę (np. pliki, konfiguracja)
- [x] Automatyzuj proces backupu
- [x] Monitoruj powodzenie backupów
- [x] Miej plan awaryjny (disaster recovery)

11 Dokumentacja i runbook

- [x] Utrzymuj aktualną dokumentację architektury
- [x] Dokumentuj konfigurację i zmienne środowiskowe
- [x] Przygotuj runbook operacyjny (krok po kroku)
- [x] Opisuj procedury awaryjne
- [x] Dokumentuj proces deploymentu
- [x] Udostępnij diagramy infrastruktury
- [x] Utrzymuj checklisty operacyjne
- [x] Dbaj o wiedzę zespołu (onboarding)

12 Najczęstsze pułapki

- [x] Uruchamianie z domyślną konfiguracją
- [x] Brak zabezpieczeń (otwarte endpointy)
- [x] Logowanie poufnych danych
- [x] Brak testów integracyjnych
- [x] Brak monitoringu i alertów
- [x] Ręczny deployment na produkcję
- [x] Brak planu odzyskiwania po awarii
- [x] Nieaktualne zależności
- [x] Niewystarczające zasoby serwera
- [x] Brak dokumentacji i runbooka

13 Finalna checklista (gotowy do produkcji!)

- [x] Aplikacja działa stabilnie i jest przetestowana
- [x] Konfiguracja prod jest bezpieczna
- [x] Baza danych i migracje działają poprawnie
- [x] Monitoring, logowanie i alerty są skonfigurowane
- [x] Deployment jest zautomatyzowany (CI/CD)
- [x] Zespół ma dokumentację i runbook
- [x] Wydajność została przetestowana
- [x] Backup i plan awaryjny są gotowe
- [x] Bezpieczeństwo zostało zweryfikowane
- [x] API jest udokumentowane
- [x] Infrastruktura jest skalowalna
- [x] Jesteśmy gotowi na użytkowników! 🎉

Gratulacje!

Ukończyłeś 100 map myśli ze Spring Boota!

Teraz możesz budować jeszcze lepsze aplikacje. ♡



Większe możliwości czekają na Ciebie! ☺

SPRING BOOT

Ta wiedza to dopiero początek... ♡

100 MAP MYŚLI ZE SPRING BOOTA
Praktyczna wiedza. Lepsze aplikacje. Twoja przyszłość.

MODUŁ 10
BUILD, DOCKER, CLOUD I PRODUKCJA

MAPA 100/100
Finał serii. Krok w wielką przyszłość!

Pisz testy
Buduj lepsze jutro! ♡