Spring

Cześć

Maciej Koziara

Rules

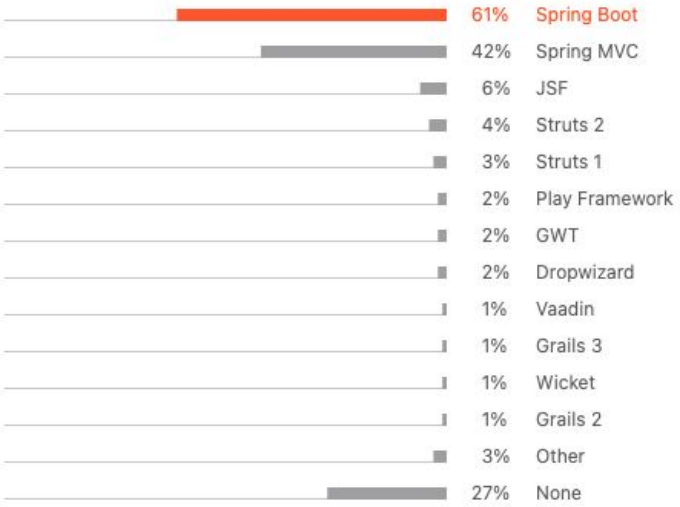
- Przerwa
- Pytania:
  - Nie czekać
  - Przerywać

Chapter 1

Introduction to Spring

Czym jest Spring?

- Framework
- Zbiór modułów
- Zbiór projektów stworzonych przy użyciu Spring Framework
- Najpopularniejszy web framework w ekosystemie Javy



Filozofia

- Wybór na każdym poziomie
- Wsparcie dla różnych perspektyw
- Silna kompatybilność wsteczna
- Przemyślane API
- Wysokie standardy jakości kodu

Spring vs Java Enterprise Edition (JEE)

- Aktualna nazwa - Jakarta EE
- Bezpośrednią przyczyną powstania Springa było skomplikowanie wczesnych wersji JEE
- To nigdy nie była konkurencja, Spring jedynie upraszczał i rozszerzał
- Spring integruje się z wieloma specyfikacjami od JEE

Alternatywy

- Quarkus
- Micronaut
- DropWizard
- Vertx

Spring vs Spring Boot

- **Spring** - dostarcza infrastrukturę do tworzenia aplikacji Webowych
- **Spring Boot** - zajmuje się konfiguracją modułów Springa tak, aby były gotowe do użycia tak szybko jak to możliwe
- Convention over Configuration

Spring Initializr

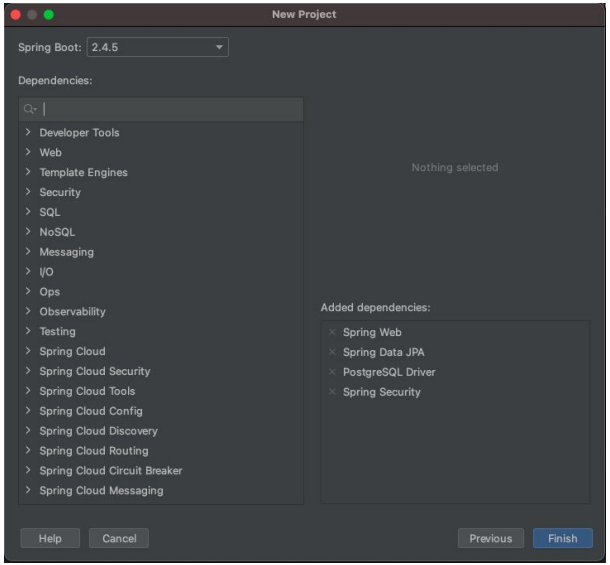
- Link: start.spring.io
- Pozwala na szybkie wygenerowanie projektu Spring Boot
- Zintegrowany ze wszystkimi popularnymi IDE

Aktualne wersje zależności:

Spring - 5.3.6

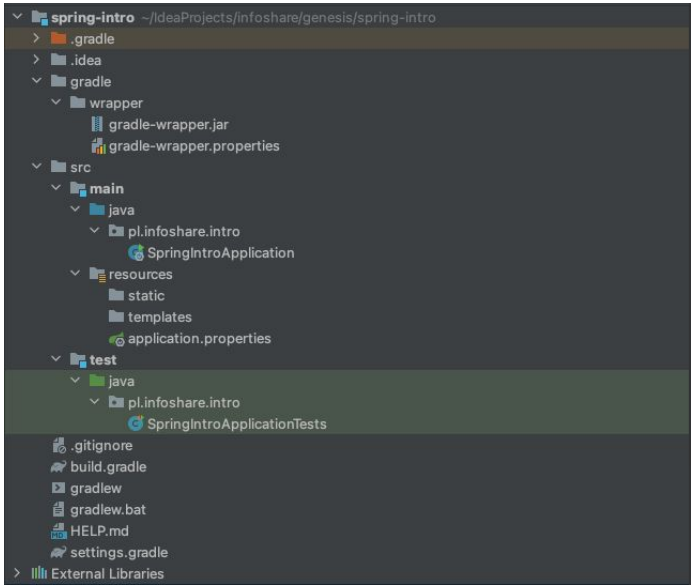
Spring Boot - 2.4.5

Min Java 8



Spring - struktura projektu

- **build.gradle** - plik ze wszystkimi zależnościami i pluginami wymaganymi do uruchomienia projektu
- **gradlew** - wrapper pozwalający na korzystanie z gradle bez jego lokalnej instalacji
- **SpringIntroApplication** - punkt wejścia aplikacji
- **SpringIntroApplicationTests** - podstawowy test sprawdzający czy aplikacja się uruchomi
- **application.properties** - plik z konfiguracją



Podstawowe adnotacje

- **@SpringBootApplication** - adnotacja używana do oznaczenia głównej klasy aplikacji zawierającej metodę main(). To od paczki znajdującej się w tej klasie rozpocznie się skanowanie potencjalnych beanów.

SpringApplication.run() - metodauruchamiająca Embedded Server i inicjalizująca Spring Framework. Powinna zostać użyta w głównej metodzie main.

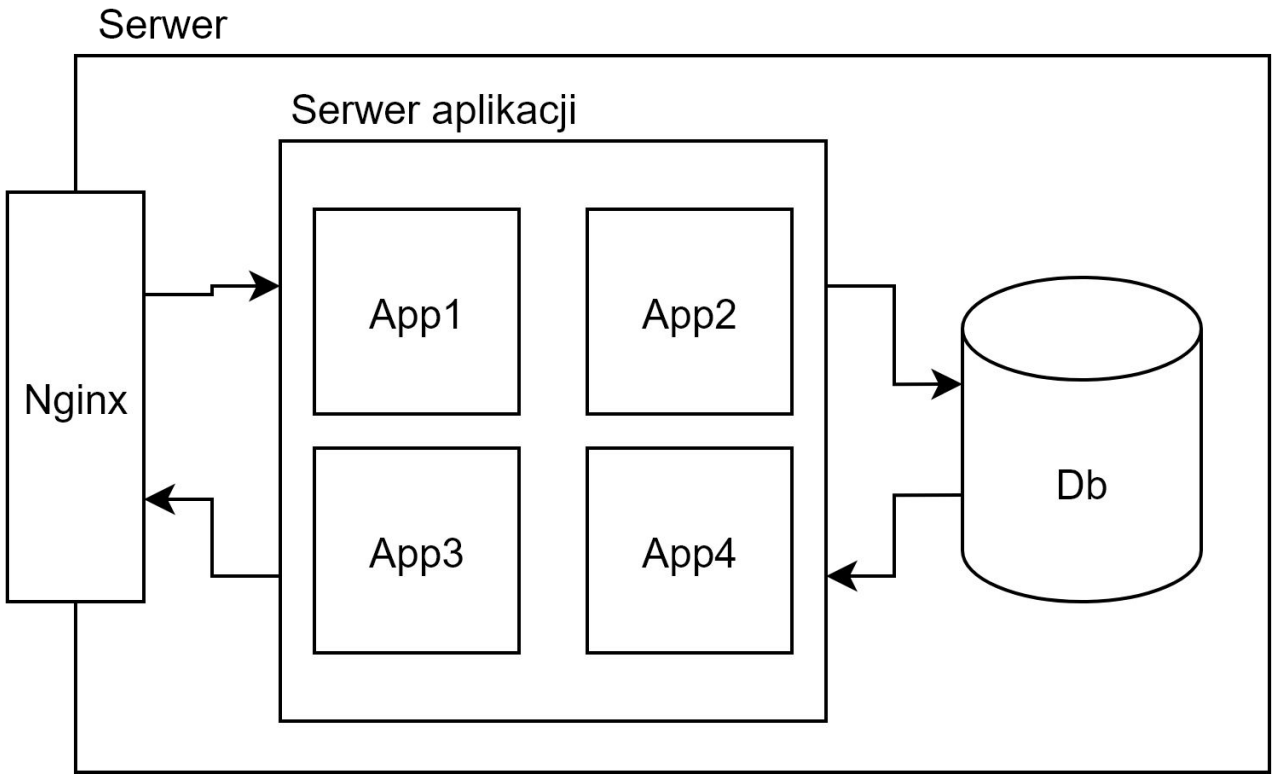
Java i JSR

- Java jest językiem mocno ustandaryzowanym
- Specyfikacje na każdą funkcjonalność:
  - od języka i JVM,
  - przez walidację encji i parsowanie JSON,
  - Kończąc na obsłudze requestów, kolejek i interakcjach z bazą danych
- Różne biblioteki implementują różne specyfikacje

Application Servers

- Jakarta EE - zbiór specyfikacji, które zapewniają takie samo działanie aplikacji na różnych serwerach
- Serwer aplikacji
  - serwer implementujący część lub wszystkie specyfikacje
  - pozwala jednocześnie na tworzenie (poprzez dostarczanie API) oraz uruchamianie aplikacji (środowisko uruchomieniowe)
- Przykłady serwerów: **WildFly (JBoss)**, **WebLogic**, **Tomcat, Jetty**

Application Servers



Embedded Servers

- Spring nie wymaga wcześniej zainstalowanego i skonfigurowanego serwera aplikacji
- Serwer uruchamia się i konfiguruje wraz ze startem aplikacji
- Możemy wpłynąć na konfigurację serwera poprzez modyfikację **application.properties** (lista)
- Domyślny embedded server używany przez Spring Boot - **Tomcat**

Spring MVC

- Podstawowy budulec:
  - @Controller / @RestController
  - @RequestMapping
- **@RestController** - pomocnicza adnot
  danej klasy jako posiadającej metody,
  zmapowane na endpointy REST
- **@RequestMapping(method=)** - użycie tej adnotacji nad metodą pozwala na zmapowanie obsługi żądania HTTP do oznaczonej metody
- **@GetMapping / @PostMapping** - wyspecjalizowane wersje adnotacji **@RequestMapping** obsługujące tylko konkretną metodę HTTP

```java
@RestController
public class SimpleController {

    @RequestMapping(method = RequestMethod.GET, path = "/api/hello")
    public String getHelloWorld() {
        return "Hello world";
    }

    @GetMapping("/api/name")
    public String getName() {
        return "Maciek";
    }
}
```

Dependency Injection

- Pozwala na oddelegowanie zarządzania cyklem życia obiektów w naszej aplikacji do frameworka
- **@Component** - używany do oznaczenia klasy jako kandydata do wstrzyknięcia
- **@Controller** - wyspecjalizowany Component
- **@Autowired** - do tak oznaczonego konstruktora zostaną wstrzyknięte zależności
  Uwaga: od Spring 4.3 ta adnotacja jest opcjonalna jeżeli klasa posiada pojedynczy konstruktor

```java
@RestController
public class SimpleController {

    private final SimpleNameService simpleNameService;
    private final SimpleHelloService simpleHelloService;

    @Autowired
    public SimpleController(SimpleNameService simpleNameService,
                            SimpleHelloService simpleHelloService) {
        this.simpleNameService = simpleNameService;
        this.simpleHelloService = simpleHelloService;
    }

    @RequestMapping(method = RequestMethod.GET, path = "/api/hello")
    public String getHelloWorld() {
        return simpleHelloService.getHello();
    }

    @GetMapping("/api/name")
    public String getName() {
        return simpleNameService.getName();
    }
}
```

```java
@Component
class SimpleHelloService {

    String getHello() {
        return "Hello";
    }
}
```

Spring Boot Gradle plugin

- **bootRun** - pozwala na uruchomienie aplikacji z poziomu gradle
- **bootJar** - pozwala na utworzenie JARa, którego będzie można uruchomić przy pomocy polecenia java -jar
- **launchScript** - pozwala na utworzenie **fully executable JAR**, który może być uruchomiony jak każdy inny executable plik

Chapter 2

Dependency Injection

Process whereby objects define their dependencies only through constructor arguments. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes.

Definicje

- **Inversion of Control** - oddanie kontroli nad tworzeniem obiektów i dostarczaniem ich do innych obiektów frameworkowi.
- **Application Context** - Springowa implementacja kontenera **IoC.** Odpowiedzialny za tworzenie, konfigurację i zarządzanie obiektami.
- **Bean** - obiekt zarządzany przez kontener zależności

Application Context

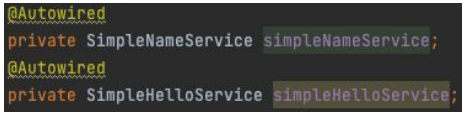
- Pozwala na programistyczny dostęp do beanów
- Posiada wiele wyspecjalizowanych implementacji w zależności od środowiska w jakim uruchamiamy aplikację (np. **WebApplicationContext**)
- Dostęp do instancji **ApplicationContext** uzyskujemy poprzez bezpośrednie wstrzyknięcie jej do klasy lub implementację interfejsu **ApplicationContextAware**
- Oprócz zarządzania zależnościami zapewnia dodatkowe funkcjonalności, np. dostęp do konfiguracji

Uwaga: idealnie nigdy nie powinniśmy bezpośrednio używać **ApplicationContext**

Proces tworzenia ApplicationContext

- **@ComponentScan** - znajduje potencjalnych kandydatów na **beany.**
- **Beany** opisywane są metadanymi.
- Budowa i walidacja grafu zależności.
- Utworzenie **beanów** oraz wychwycenie wszystkich błędów inicjalizacyjnych.

Sposoby na wstrzykiwanie

	Opis	Przykład
Konstruktor	Zalecane przez autorów Springa ze względu na łatwe osiągnięcie niemutowalnych klas i w pełni zainicjalizowanych obiektów	
Setter	Swego czasu używane do wstrzykiwania zależności opcjonalnych. Aktualnie lepiej jest użyć klasy **Optional<>** lub oznaczyć argument konstruktora adnotacją **@Nullable**	
Pola	Obecnie raczej niepraktykowane	

@Autowired

Uwaga: od Spring 4.3 adnotacja **@Autowired** jest opcjonalna jeżeli klasa posiada pojedynczy konstruktor. W połączeniu z Lombokiem i jego **@RequiredArgsConstructor** możemy znacząco zmniejszyć ilość potrzebnego do napisania kodu.

```java
@RestController
public class SimpleController {

    private final SimpleNameService simpleNameService;
    private final SimpleHelloService simpleHelloService;

    @Autowired
    public SimpleController(SimpleNameService simpleNameService,
                            SimpleHelloService simpleHelloService) {
        this.simpleNameService = simpleNameService;
        this.simpleHelloService = simpleHelloService;
    }

    @RequestMapping(method = RequestMethod.GET, path = "/api/hello")
    public String getHelloWorld() {
        return simpleHelloService.getHello();
    }

    @GetMapping("/api/name")
    public String getName() {
        return simpleNameService.getName();
    }
}
```

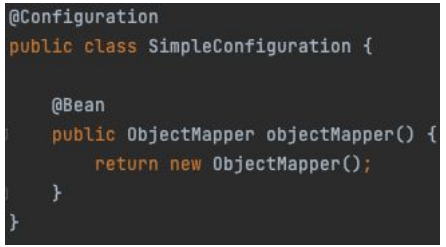
```java
@RestController
@RequiredArgsConstructor
public class SimpleController {

    private final SimpleNameService simpleNameService;
    private final SimpleHelloService simpleHelloService;

    @RequestMapping(method = RequestMethod.GET, path = "/api/hello")
    public String getHelloWorld() {
        return simpleHelloService.getHello();
    }

    @GetMapping("/api/name")
    public String getName() {
        return simpleNameService.getName();
    }
}
```

Definiowanie beanów

	Opis	Przykład
@Component	Najbardziej podstawowa adnotacja służąca do oznaczenia klasy jako kandydata do wstrzyknięcia.	
@Service / @Controller / @Repository / @Configuration	Wyspecjalizowane wersje **@Component**. Dostarczają więcej informacji o swoim przeznaczeniu i mogą pełnić specjalne funkcje w samym frameworku, np. **@Controller** dodatkowo określa, że dana klasa obsługuje żądania HTTP. Są to tak zwane Stereotype Annotation.	
@Bean + @Configuration	Przydatne gdy chcemy dostarczyć do kontekstu aplikacji klasę z zewnętrznej biblioteki lub chcemy mieć kilka beanów tego samego typu, lecz inaczej skonfiguranych (np. kilka wersji HttpClient)	

Cechy kontenera IoC w Spring

- Wszystkie **beany** domyślnie tworzone są jako Singletony -> istnieje tylko jedna instancja danej klasy w kontekście aplikacji.
- Domyślnie wszystkie zadeklarowane w konstruktorze zależności są wymagane. Jeżeli chcemy zmienić zachowanie możemy użyć adnotacji **@Nullable** lub owrapować zależność w **Optional<>.** Jeżeli dany bean nie będzie istniał zostanie dostarczony tam **null**, który musimy odpowiednio obsłużyć.
- Wszystkie **beany** są tworzone i wstrzykiwane przy starcie aplikacji. Dzięki temu informację o niepoprawnej konfiguracji mamy już przy starcie systemu, niewielkim kosztem czasu startu.

Dependency Injection - how it works

- Wszystkie klasy w paczce określonej przez **@SpringBootApplication** (a konkretniej **@ComponentScan**) zostaną zeskanowane w poszukiwaniu kandydatów do wstrzyknięcia
- Na podstawie zeskanowanych klas utworzony jest graf zależności
- Na podstawie analizy grafu wykrywane są błędy konfiguracyjne (np. brakujące zależności)
- Wszystkie zależności zostaną utworzone, dostarczone i zapisane w **ApplicationContext**

Bean Life Cycle

- **@PostConstruct** - pozwala na uruchomienie kodu podczas tworzenia **beana**. Często używany do początkowej inicjalizacji lub upewnienia się, że dany **bean** jest w stanie funkcjonować w danym środowisku (np. ma odpowiedni dostęp do plików).
- **@PreDestoy** - uruchamia kod chwilę przed usunięciem **beana** z kontenera

```java
@Component
class SimpleHelloService {

    String getHello() {
        return "Hello";
    }

    @PostConstruct
    void init() {
        System.out.println("Some init method");
    }

    @PreDestroy
    void cleanUp() {
        System.out.println("Clean up stuff");
    }
}
```

Scopes

- Pozwalają na ograniczenie życia **beana**
- **Singleton** - **bean** tworzony jest raz przy starcie aplikacji. Domyślna wartość.
- **Prototype** - **bean** tworzony jest przy każdym wstrzyknięciu. Dzięki temu może przechowywać stan. Uruchomione są tylko metody związane z inicjalizacją **beana**.
- **Request** - **bean** tworzy się dla każdego żądania HTTP.
- **Session** - **bean** tworzy się dla każdej sesji użytkownika
- Inne projekty Springowe mogą dokładać dodatkowe **Scope**.

Scopes

- **@Scope -** adnotacja pozwalająca na zmianę domyślnej wartości. Może być łączona zarówno z adnotacją **@Component** jak i **@Bean.**

```
@Service
@Scope(WebApplicationContext.SCOPE_REQUEST)
class SimpleNameService {
```

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public ObjectMapper objectMapper() {
    return new ObjectMapper();
}
```

Beany tego samego typu

- Jeżeli próbujemy wstrzyknąć interfejs, który posiada wiele implementacji musimy sprecyzować jaką implementację chcemy wstrzyknąć
- **@Qualifier(“”)**
  - W połączeniu z **@Component** dodaje metadane do definicji **bean**a
  - W połączeniu z **@Autowired** pozwala na sprecyzowanie konkretnej implementacji do wstrzyknięcia

```java
@Component
@Qualifier("simple")
public class SimpleTextExtractor implements TextExtractor {

    @Override
    public String extract(String source) {
        return null;
    }
}
```

```java
private final TextExtractor textExtractor;

public ExtractorService(@Qualifier("simple") TextExtractor textExtractor) {
    this.textExtractor = textExtractor;
}
```

Beany tego samego typu

- **@Primary** - pozwala na określenie domyślnej implementacji jeżeli nie zostanie ona doprecyzowana

```java
@Primary
@Component
@Qualifier("file")
public class FileTextExtractor implements TextExtractor {
    @Override
    public String extract(String source) {
        return null;
    }
}
```

- W celu uproszczenia kodu możemy utworzyć własne adnotacje typu **@Qualifier**

```java
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface SimpleExtractor {

}
```

```java
@Component
@SimpleExtractor
public class SimpleTextExtractor implements TextExtractor
```

```java
@Component
public class ExtractorService {

    private final TextExtractor textExtractor;

    public ExtractorService(@SimpleExtractor TextExtractor textExtractor) {
        this.textExtractor = textExtractor;
    }
}
```

Beany tego samego typu

- W przypadku gdy nie określimy żadnego beana jako **@Primary** i nie dodaliśmy adnotacji **@Qualifier** Spring spróbuje wstrzyknąć zależność na podstawie nazwy parametru
- Jeżeli nazwa parametru jest taka sama jak nazwa klasy zapisana jako **camelCase** to zostanie ona wstrzyknięta.
  Np. dla parametru `TextExtractor simpleTextExtractor`
  zostałaby wybrana klasa `SimpleTextExtractor`

Wstrzykiwanie kolekcji

- Jeżeli chcemy wstrzyknąć wszystkie implementacje danego interfejsu możemy wstrzyknąć klasę **List<>**, **Set<>** lub **Map<>** (jako klucz zostanie użyta nazwa **beana**)
- W celu sterowania kolejnością wstrzykniętych interfejsów można użyć adnotacji **@Ordered**

```
@Component
public class ExtractorService {

    private final List<TextExtractor> textExtractors;

    public ExtractorService(List<TextExtractor> textExtractors) {
        this.textExtractors = textExtractors;
    }
}
```

Chapter 3

Request Mapping

REST

- **REST** - **Re**presentational **S**tate **T**ransfer - styl architektury oprogramowania, opierający się o zbiór reguł opisujących jak definiowane są zasoby i określający sposoby dostępu do zasobów.

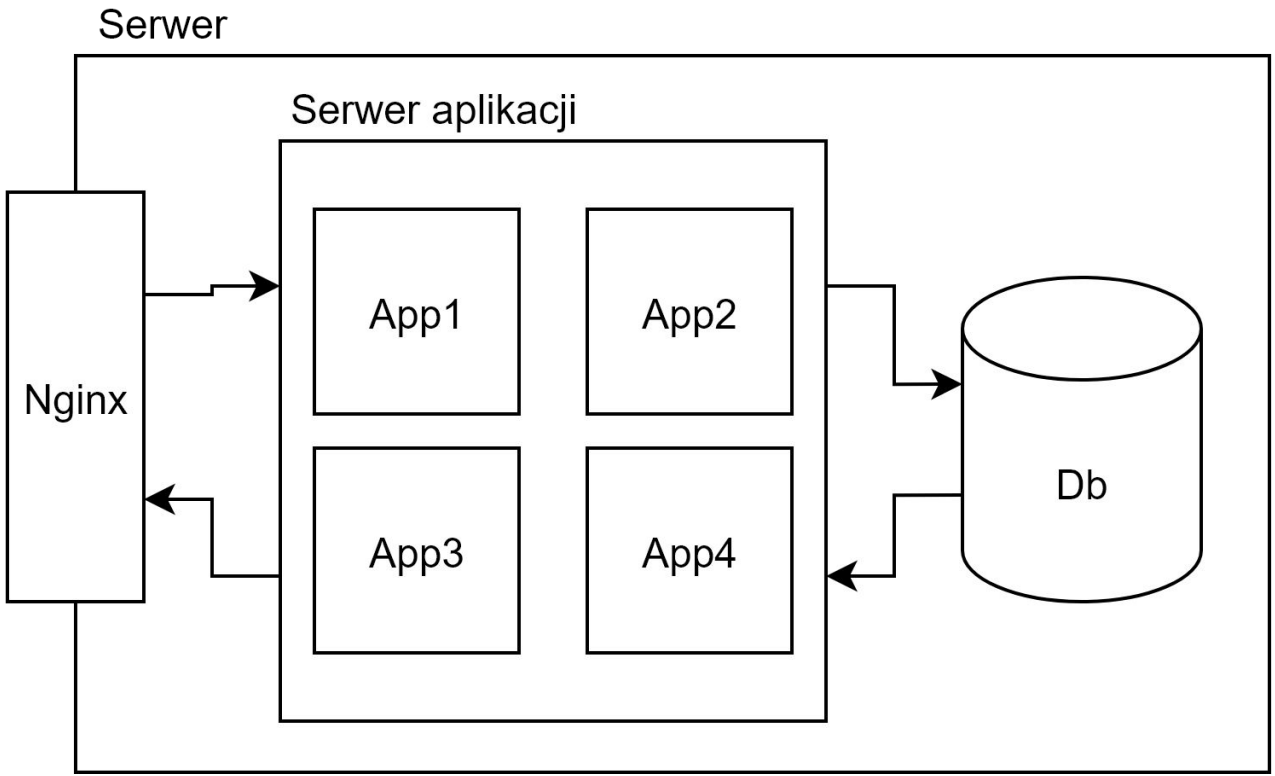
Cechy:

- prostota
- lekkość
- uniwersalność

Application Servers

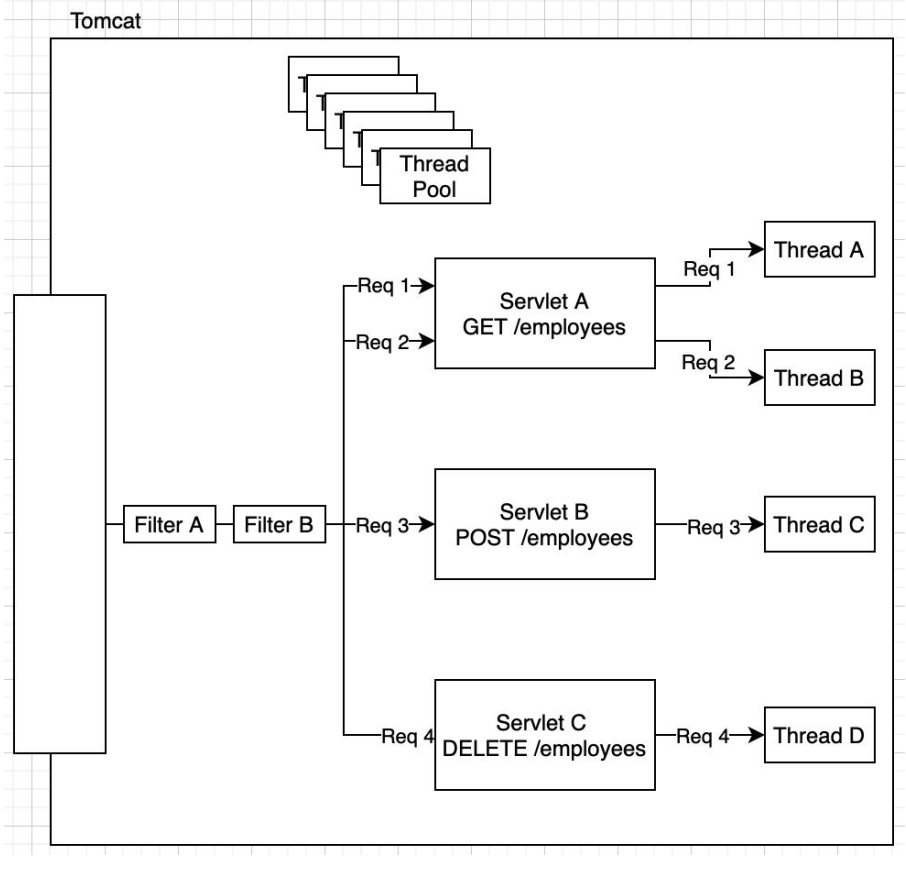
- Jakarta EE - zbiór specyfikacji, które zapewniają takie samo działanie aplikacji na różnych serwerach
- Serwer aplikacji
  - serwer implementujący część lub wszystkie specyfikacje
  - pozwala jednocześnie na tworzenie (poprzez dostarczanie API) oraz uruchamianie aplikacji (środowisko uruchomieniowe)
- Przykłady serwerów: **WildFly (JBoss)**, **WebLogic**, **Tomcat, Jetty**

Application Servers

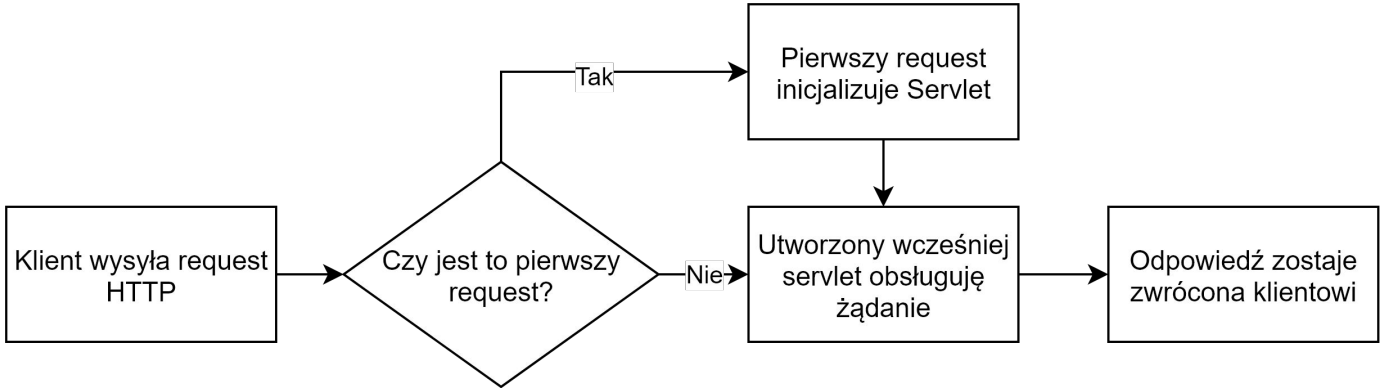


Servlet API - JSR 340

- Specyfikacja określająca w jaki sposób klasy odpowiadają na żądania HTTP
- Servlet - obiekt odpowiadający na żądanie HTTP. Istnieje tylko jedna Instancja Servletu.



Servlet Request Processing



- **Uwaga!**
  Servlet jest reużywany przez wszystkie przychodzące żądania.
  Z tego powodu stan trzymany w klasie nie jest bezpieczny wątkowo i nie należy przechowywać danych specyficznych dla requestów w klasie.

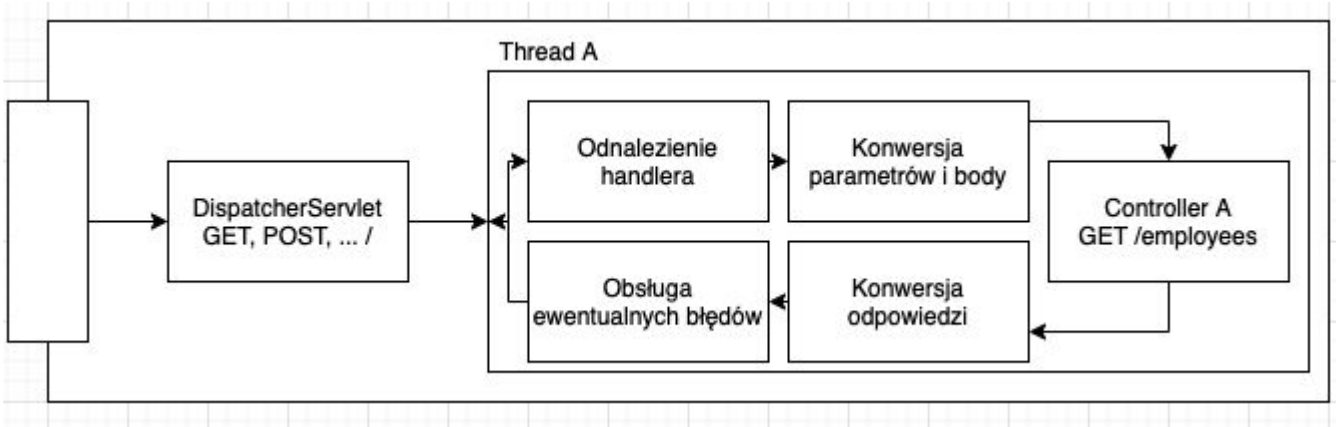
Spring Web MVC

- Jeden z projektów z rodzina Springa pozwalający na tworzenie aplikacji Webowych
- Istnieje od początku historii Springa
- Szerzej znany jako Spring MVC
- Oparty w całości na Servlet API
- Projekty oparte o ten framework deployowane są do Servlet Container (np. **Tomcat**)

Spring MVC - Dispatcher Servlet

- Jeden centralny Servlet (DispatcherServlet), który:
  - przyjmuje requesty i deleguje je do odpowiedniego kontrolera
  - wyciąga dane z żądania tj. path, params, body i konwertuje do odpowiednich typów
  - zwraca odpowiedź w odpowiednim formacie oraz obsługuje ewentualne błędy

Spring MVC - Dispatcher Servlet



Spring MVC - kontroler

- **@Controller** - podstawowy budulec Spring MVC. Określa, że oznaczona w ten sposób klasa jest beanem zawierającym mapowania endpointów do obsługujących je metod.
- **@RequestMapping** - pozwala na określenie warunków, kiedy oznaczona metoda powinna obsłużyć dany request.

Możliwe warunki:

path - ścieżka (URI) danego requesta

method - użyta metoda HTTP

consumes - format w jakim został wysłany ciało żądania (nagłówek **Content-type)**

produces - format odpowiedzi oczekiwany przez klienta (nagłówek **Accept**)

headers - nagłówki, które muszą być ustawione w żądaniu

params - zmienne, które muszą być obecne w URI

Spring MVC - kontroler

- **@RequestMapping** - jeżeli nie określimy konkretnego **HTTP Method** oznaczona tą adnotacją metoda będzie obsługiwać żądania dowolną metodą
- **@GetMapping / @PostMapping / @*Mapping** - wyspecjalizowane wersje **@RequestMapping** obsługujące tylko jedną metodę
- **@ResponseBody** - określa że wartość zwrócona z tej metody ma zostać bezpośrednio skonwertowana jako ciało odpowiedzi. Bez tej adnotacji zostaną uruchomione mechanizmy MVC przygotowujące i renderujące widoki HTML po stronie serwera.

Uwaga - jeżeli użyjemy **@RequestMapping** na poziomie klasy, wszystkie metody będą musiały spełniać warunki określone w adnotacji. Wartość **path** zostanie połączona z wartością **path** adnotacji nad metodą.

```java
@Controller
@RequestMapping("/api")
public class SimpleController {

    @RequestMapping(method = RequestMethod.GET, path = "/hello", headers = "X-HEADER=123")
    @ResponseBody
    public String getHelloWorld() {
        return "Hello!";
    }

    @GetMapping("/employees")
    @ResponseBody
    public SimpleModel getName() {
        return new SimpleModel();
    }
}
```

Spring MVC - kontroler

- **@RestController -** wyspecjalizowany kontroler określający, że dane endpointy powinny być zawsze traktowane jako REST. Połączenie **@Controller** i **@ResponseBody**.

```
@RestController
@RequestMapping("/api")
public class SimpleController {

    @RequestMapping(method = RequestMethod.GET, path = "/hello")
    public String getHelloWorld() {
        return "Hello!";
    }

    @GetMapping("/employees")
    public SimpleModel getName() {
        return new SimpleModel();
    }
}
```

Pobieranie danych z żądania

- **@PathVariable -** pobiera i ustawia wartość zmiennej ze ścieżki. Obsługuje Regex.
- **@RequestParam** - pozwala na pobranie wartości query param. Argument oznaczony tą adnotacją staje się wymagany.
- **@RequestBody** - mapuje ciało żądania na określony typ
- **@RequestHeader** - pobiera wartość nagłówka
- **@CookieValue** - pobiera wartość Cookie

Uwaga: jeżeli argument metody nie zostanie oznaczony jako **@RequestBody** Spring spróbuje spróbuje przypisać wartości pól w danej klasie do wysłanych query params na podstawie nazw.

```java
@RestController
public class SimpleController {

    @GetMapping("/api/employees/{id}")
    public SimpleModel getName(@PathVariable Integer id) {
        return new SimpleModel();
    }

    @GetMapping("/api/files/{name:[a-z-]+}{ext:\\.[a-z]+}")
    public String getFileName(@PathVariable String name, @PathVariable String ext) {
        return name + ext;
    }

    @GetMapping("/api/models")
    public List<SimpleModel> getModels(@RequestParam String search,
                                       @RequestParam(required = false) String city,
                                       @RequestParam(defaultValue = "true") boolean adultOnly) {
        return Collections.emptyList();
    }

    @PostMapping("/api/models")
    public void createModel(@RequestBody SimpleModel model) {
    }
}
```

Zwracanie danych

- Jeżeli używamy **@RestController** to domyślnie każdy obiekt zwrócony z metody obsługującej endpoint zostanie skonwertowany w JSON
- **@ResponseStatus** - pozwala na określenie statusu, który ma zostać zwrócony
- **ResponseEntity<>** - pozwala na dynamiczne określenie statusu i ustawienie nagłówków odpowiedzi

```java
@RestController
public class SimpleController {

    @PostMapping("/api/models")
    public ResponseEntity<SimpleModel> createModel(@RequestBody SimpleModel model) {
        return ResponseEntity.ok()
                .header( headerName: "X-Custom",  ...headerValues: "test")
                .body(model);
    }

    @DeleteMapping("/api/models/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void getModels(@PathVariable("id") String modelId) {
    }
}
```

Chapter 4

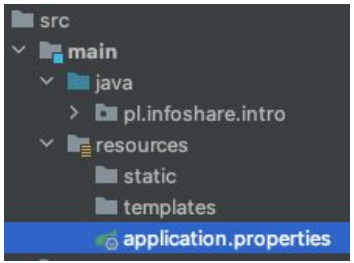
Application Configuration

Spring Environment

- **Environment** - abstrakcja Springowa modelująca dwa aspekty: **properties** i **profile**
- **properties**
  - pozwalają na konfigurację aplikacji
  - mogą pochodzić z różnych źródeł (env variables, pliki konfiguracyjne, JVM itp.)
  - **@PropertySource** - pozwala na określenie pliku z którego mają zostać załadowane propertiesy (wspierane są pliki z rozszerzeniem **.properties** oraz **.yaml**)
  - **Environment** służy jako centralne miejsce do ich rejestracji i zapewniające jednolity dostęp do ich wartości
- **profiles**
  - pozwalają na warunkową rejestrację beanów, w zależności od tego czy profil jest aktywny
  - **Environment** określa, który profil jest aktualnie aktywny
  - domyślnie aktywny jest profil **default**

Application Properties

- Plik **application.properties** zawierający konfigurację dla aplikacji opartych na Spring Boot
- Domyślnie umieszczany w katalogu resources
- Może być umieszczony na zewnątrz uruchamianego JARa. Pozwala to na dostosowanie konfiguracji do środowiska, na którym zostanie uruchomiona aplikacja.
- Wspierany jest zarówno format **.properties** jak i **.yaml**




```
spring.application.name=Demo application
server.port=8081

my.own.custom.property=Value
```

Application Properties - wczytywanie

- Kolejność wczytywania:
  - **application.properties** inside JAR
  - **application.properties** outside JAR
  - command line properties
- Dokładnie opisana kolejność https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config
- Przy uruchomieniu aplikacji możemy określić własną ścieżkę do pliku **application.properties** poprzez użycie **spring.config.location,** jeżeli plik niekoniecznie istnieje ścieżkę należy poprzedzić prefixem **optional:**, np.

```
java -jar myproject.jar --spring.config.location=optional:file:./config/custom.properties
```

Wstrzykiwanie konfiguracji

- **@Value("${some.key}")** - pozwala na wstrzyknięcie wartości z konfiguracji
- Syntax **${}** wskazuje na to, że wartość ma zostać odczytana z konfiguracji
- Spring konwertuje wartość na właściwy typ użyty w Javie. Wspierane są nie tylko typy proste, ale też **LocalDate, Path** itp.
- Tak zdefiniowana wartość musi istnieć. Jeżeli chcemy możemy określić domyślną wartość poprzez zapis **${some.key:default-value}**

```java
@Component
public class SimpleConfiguration {

    private final String customProperty;

    public SimpleConfiguration(@Value("${my.own.custom.property}") String customProperty) {
        this.customProperty = customProperty;
    }
}
```

Configuration Properties

- **@ConfigurationProperties** - pozwala na dostęp do konfiguracji w otypowany sposób
- Wspiera zarówno podejście **JavaBean** (domyślny konstruktor, get i set) jak i **immutable** (wstrzykiwanie przez konstruktor)
- **@ConstructorBinding** - wymagana adnotacja jeżeli chcemy użyć podejścia **immutable**
- **@DefaultValue** - pozwala na określenie domyślnej wartości
- **@ConfigurationPropertiesScan** - automatycznie wykrywa i dostarcza do kontekstu klasy **@ConfigurationProperties**
- Wspierana jest walidacja propertiesów
- Relaxed binding - nie ma ściśle określonego nazewnictwa, którego należy trzymać się w pliku **application.properties**. Klucze mogą być zarówno **camelCase**, **snake_case** jak i **kebab-case.** Dokumentacja zaleca korzystanie z **kebab-case.**

```
simple.name=Maciek
simple.age=28
simple.birth-date=1992-08-25
```

```
@SpringBootApplication
@ConfigurationPropertiesScan
public class SpringIntroApplication {

    public static void main(String[] ar
}
```

```
@ConstructorBinding
@ConfigurationProperties("simple")
public class SimpleConfiguration {

    private final String name;
    private final Integer age;
    private final LocalDate birthDate;

    public SimpleConfiguration(String name,
                               @DefaultValue("99") Integer age,
                               LocalDate birthDate) {
        this.name = name;
        this.age = age;
        this.birthDate = birthDate;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    public LocalDate getBirthDate() {
        return birthDate;
    }
}
```

Spring Profile

- Pozwalają na ładowanie **@Configuration**, **@ConfigurationProperties** lub **bean** w zależności od aktywnego profilu
- **@Profile("")** - pozwala na określenie, który profil musi być aktywny, aby załadować daną klasę do kontekstu
- **spring.profiles.active** - property odpowiedzialne za aktywację profilu
- **application-{profile}.properties** - plik z dodatkowymi propertiesami, który zostanie załadowany tylko gdy profil jest aktywne
- Wspierają proste operatory logiczne: **!**, **|** oraz **&**
- Domyślnie aktywny jest profil **default**

```java
@Profile("dev")
@RestController
public class DevController {

    @GetMapping("/some-dev-data")
    public String getDevData() {
        return "";
    }
}
```

```
spring.profiles.active=dev

spring.application.name=Demo application
server.port=8081

my.own.custom.property=Value

simple.name=Maciek
simple.age=28
simple.birth-date=1992-08-25
```

Chapter 5

Spring Boot Philosophy

Spring Boot goals

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects.
- Absolutely no code generation and no requirement for XML configuration.

Spring Boot Starter

- Służą do grupowania wszystkich zależności potrzebnych do działania danej funkcjonalności
- np. **spring-boot-starter-json** dostarcza **spring-web** i wszystkie moduły biblioteki **Jackson** potrzebne do sprawnego jej używania
- w przypadku tworzenia własnego startera ich nazwa nie powinna zaczynać się od **spring-boo**

```
plugins {
        id "org.springframework.boot.starter"
}

description = "Starter for reading and writing json"

dependencies {
        api(project(":spring-boot-project:spring-boot-starters:spring-boot-starter"))
        api("org.springframework:spring-web")
        api("com.fasterxml.jackson.core:jackson-databind")
        api("com.fasterxml.jackson.datatype:jackson-datatype-jdk8")
        api("com.fasterxml.jackson.datatype:jackson-datatype-jsr310")
        api("com.fasterxml.jackson.module:jackson-module-parameter-names")
}
```

Spring Auto Configuration

- Pozwala na dostarczenie domyślnej konfiguracji przez twórców Springa i uniknięcie powtarzalnego kopiowania przykładów z dokumentacji
- Oparta jest na analizie dołączonych zależności i kontekstu aplikacji
- Pozwala na łatwe nadpisanie i dostosowanie jej elementów do naszych potrzeb

Chapter 6

Logging

Logowanie w Springu

- Wsparcie dla kilku popularnych rozwiązań: **Logback, Java Util Logging, Log4J2**
- **Logback -** domyślnie używany przez projekty oparte na Spring
- Domyślnie logowanie odbywa się tylko do konsoli
- Jeżeli domyślna konfiguracja jest niewystarczająca możemy na nią wpłynąć na dwa sposoby:
  - **application.properties** - Spring dostarcza zestaw podstawowych propertiesów pozwalających na wpłynięcie na loggery
  - **logback-spring.xml** - plik dodawany w resources. Pozwala w całości nadpisać domyślną konfigurację loggerów
- Format logów:

```
<Date> <Time> <Level> <PID> --- <Thread name> <Logger name> <Message>
2019-03-05 10:57:51 INFO 45469 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat
```

Konfiguracja

- **Application.properties** - Spring dostarcza podstawowe propertiesy wpływające na zachowanie loggerów:
  - **debug** - uruchamia dodatkowe wiadomości pochodzące z internali Springa. **Nie zmienia poziomu logów, jedynie dostarcza dodatkowe logi.**
  - **logging.file.name, logging.file.path** - określa plik / folder do którego zapisane zostaną logi
  - **logging.logback.rollingpolicy.*** - określają zachowanie rotacji plików. Domyślnie nowy powstaje co każde 10 MB
  - **logging.level.root** - globalny poziom logów
  - **logging.level.<*package*>** - poziom logów aktywny w danej paczce

Chapter 7

Spring Boot - practices and tools

Spring Boot - package structure

- Preferowany układ paczek to koncepcja **package-by-feature**
- Każdy **feature** dostaje swoją niezależną paczkę, w której znajduje się cała logika dotycząca danej funkcjonalności
- Często wprowadza się dodatkową paczkę **domain**, która zawiera modele biznesowe i repozytoria umożliwiające interakcję z bazą danych
- W celu uproszczenia kodu zaleca się przygotowanie osobnych modeli dla warstwy REST i biznesowej. Konwersją pomiędzy modelami powinny zajmować się tzw. mappery
- Aby przyśpieszyć pracę możemy skorzystać z narzędzia, które automatycznie mapuje obiekty pomiędzy typami - MapStruct

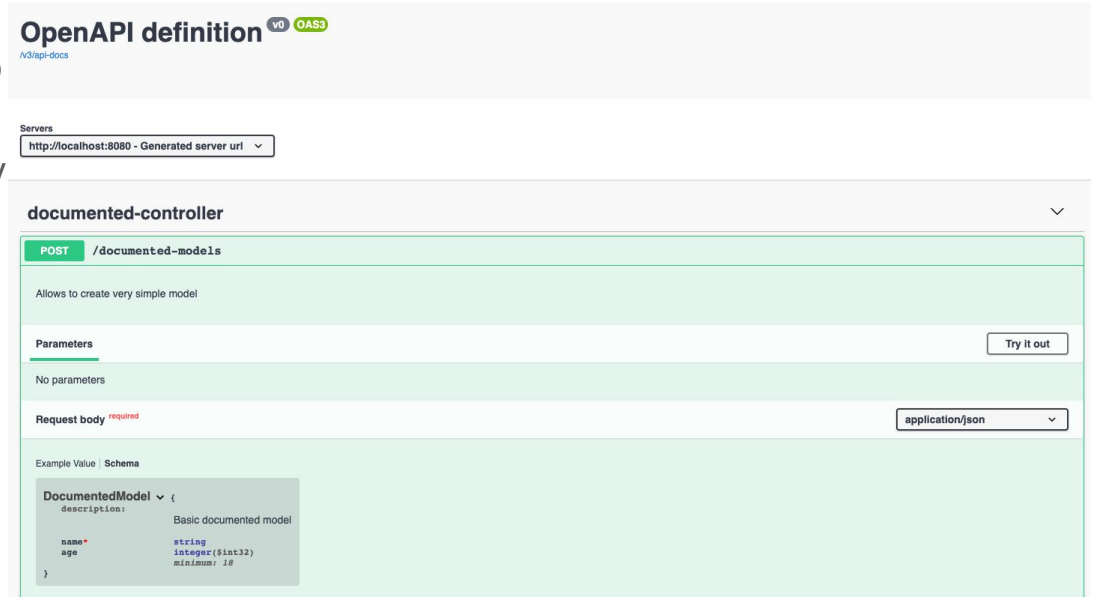
Swagger

- Zbiór narzędzi pozwalających na dokumentowanie API
- Oparty na OpenApi Specification v3
- **springdoc-openapi -** nieoficjalny projekt pozwalający na autokonfigurację Swagger i wygenerowanie podstawowego opisu endpointów
- Automatycznie konfiguruje UI pozwalający na przeglądanie dokumentacji
- Umożliwia dostosowanie wygenerowanej dokumentacji przy pomocy [swagger annotations]

```java
@RestController
public class DocumentedController {

    @PostMapping("/documented-models")
    @Operation(description = "Allows to create very simple model")
    public DocumentedModel createModel(@Valid @RequestBody DocumentedModel model) {
        return model;
    }
}
```

```java
@Value
public class DocumentedModel {
    @NotBlank
    String name;
    @Min(18)
    Integer age;
}
```



Spring Actuator

- Zbiór gotowych narzędzi pozwalających na monitorowanie i konfigurację aplikacji
- Składa się z endpointów wystawionych po HTTP lub JMX
- Przykładowe endpointy:
  - `/health` - info o stanie aplikacji
  - `/beans` - beany zarejestrowane w kontekście
  - `/flyway` - wyświetla zaaplikowane migracje bazodanowe
  - `/loggers` - wyświetla i modyfikuje ustawienia loggera
  - `/metrics` - wyświetla metryki dla aplikacji
- Pełna lista endpointów znajduje się tutaj

Spring Actuator - konfiguracja

- **spring-boot-starter-actuator** - starter zapewniający autokonfigurację actuatora
- **management.endpoint.<endpoint>.enabled** - property pozwalające na włączanie / wyłączanie podanego endpointa
- **management.endpoints.web.exposure.include** - property określające, które endpointy mają być dostępne na zewnątrz. Domyślnie tylko **/health** i **/info,** są dostępne
- **info.*** - pozwala na dostarczenie dodatkowych informacji dla **/info**
- **Management.endpoint.health.show-details** - pozwala na odczytanie szczegółów health checks (ilość pamięci, zużycie procesora, połączenia z bazą danych itp.)
- **management.endpoint.health.probes.enabled -** wystawia dodatkowe endpoint dla Kubernetes określające liveness i readiness
- Jeżeli dostarczone endpointy nie spełniają oczekiwań możemy dopisać własne rejestrując **bean** oznaczony adnotacją **@Endpoint** z metodą oznaczoną jedną z adnotacji: **@ReadOperation, @WriteOperation, @DeleteOperation**

Micrometer

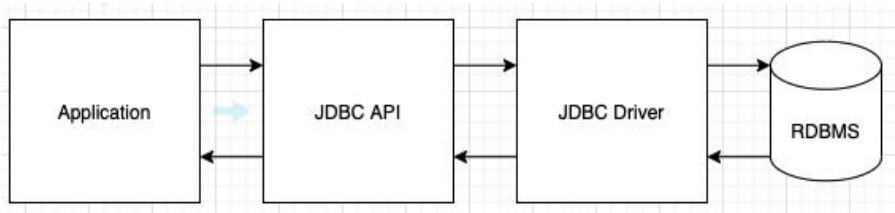
- Stanowi fasadę dla różnych narzędzi analizujących metryki, pozwalając na jednolity sposób zbierania ich w aplikacji bez uzależnienia się od konkretnego narzędzia
- Wspierany przez twórców Springa
- Wartości metryk wystawiane są na zewnątrz poprzez endpoint **/metrics** udostępniony przez **Spring Actuator**
- W zależności od wybranego narzędzia sposób konfiguracji będzie nieco inny - szczegóły tutaj

Chapter 8

JPA and Hibernate

JDBC

- **JDBC** (Java Database Connectivity) **API** dostarcza interfejs pozwalający na spójną komunikację z różnymi bazami danych
- Podstawowy sposób interakcji z bazą danych w Java
- **Driver** - interfejs będący punktem wejścia w interakcję z JDBC. Każda relacyjna baza danych dostarcza swoją własną implementację tego interfejsu.
- **DriverManager** - zajmuje się tworzeniem fizycznych połączeń z bazą danych
- **DataSource** - wrapuje **DriverManager**. Odpowiedzialny za dostarczanie połączenia z bazą danych. Pod spodem zazwyczaj optymalizuje interakcje poprzez wykorzystanie Connection Pool.
- **HikariCP** - biblioteka odpowiedzialna za zarządzanie pulą połączeń do bazy danych



JDBC

- **PreparedStatement** - pozwala na prekompilację zapytania wysłanego do bazy danych, zabezpieczają przed SQL Injection
- **ResultSet** - reprezentuje rezultat zwrócony z bazy danych. Pod spodem operuje na kursorze. Wymaga zmapowania wyniku na obiekty Javove.

```
try (var connection : Connection = dataSource.getConnection();
     var statement : Statement = connection.createStatement();
     var resultSet : ResultSet = statement.executeQuery( sql: "select id, full_name, age from simple_entity")) {

    var result = new ArrayList<SimpleEntity>();
    while (resultSet.next()) {
        var entity = new SimpleEntity();
        entity.setId(resultSet.getLong( columnLabel: "id"));
        entity.setName(resultSet.getString( columnLabel: "full_name"));
        entity.setAge(resultSet.getInt( columnLabel: "age"));
        result.add(entity);
    }

    return result;
} catch (SQLException ex) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, ex.getMessage(), ex);
}
```

JdbcTemplate

- Springowy wrapper na JDBC. Upraszcza podstawowe operacja i chroni przed popełnieniem częstych błędów
- **PreparedStatementCreator** - odpowiada za przygotowanie zapytania
- **RowMapper** - odpowiedzialny za konwersję **ResultSet** do obiektu Javovego

```java
@Override
public List<SimpleEntity> findAll() {
    return jdbcTemplate.query(
            sql: "select id, full_name, age from simple_entity",
            this::simpleEntityRowMapper
    );
}

private SimpleEntity simpleEntityRowMapper(ResultSet rs, int rowNum) throws SQLException {
    var simpleEntity = new SimpleEntity();
    simpleEntity.setId(rs.getLong( columnLabel: "id"));
    simpleEntity.setName(rs.getString( columnLabel: "full_name"));
    simpleEntity.setAge(rs.getInt( columnLabel: "age"));

    return simpleEntity;
}
```

JPA

- **JPA (Jakarata Persistence, kiedyś Java Persistence API)** - standard opisujący opisujący mapowanie między klasami Java,
  a tabelami w relacyjnej bazie danych
- Oparta na dwóch elementach:
  - Adnotacje - pozwalają na zdefiniowanie mapować
  - JPQL - język pozwalający na tworzenie zapytań do bazy danych niezależnych od konkretnej bazy danych
- Znacząco zmniejsza ilość kodu potrzebnego do interakcji z bazą danych
- Najpopularniejsze implementacje: **Hibernate** (domyślnie używany przez Spring), **EclipseLink**

Entities

- Opisują mapowanie pomiędzy modelem (klasą), a wierszem w tabeli
- Zasady
  - Klasa musi być oznaczona adnotacją **@Entity**
  - Muszą posiadać domyślny konstruktor o dostępie publicznym lub package-private
  - Klasa nie może być **final**
  - Dostęp do pól powinien odbywać się poprzez get / set

Powyższe reguły opisane są przez specyfikację **JPA**. Sama implementacja dostarczona przez **Hibernate** nie jest tak restrykcyjna, jednak zalecane jest stosowanie się do zasad opisanych w **JPA**.

Podstawowe adnotacje

- **@Entity -** oznacza daną klasę jako encję
- **@Table -** pozwala doprecyzować informacje dotyczące tabeli, która jest mapowana do encji tj. **name, catalog, schema**
- **@Id -** oznacza dane pole w klasie jako te zawierające referencję do **primary key**
- **@GeneratedValue -** używane w połączeniu z **@Id**. Określa, że wartość pola powinna zostać ustawiona przez implementację **JPA**
- **@Column -** pozwala nadpisać domyśle wartości mapujące pole do kolumny w tabeli.

Jedynymi wymaganymi adnotacjami są **@Entity** i **@Id**. Pozostałe adnotacje są wymagane tylko wtedy gdy potrzebujemy nadpisać domyślne wartości.

Domyślna strategia nazw gdy korzystamy ze Spring zakłada, że nazwy klas / pól mapowane są do tabel / kolumn jako **snake_case**.

```java
@Entity
public class SimpleEntity {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "full_name", columnDefinition = "text")
    private String name;

    private Integer age;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

Konfiguracja

- **spring.datasource.url** - lokalizacja bazy danych. Protokołem jest zazwyczaj **jdbc:<nazwa-bazy danych>**
- **spring.datasource.username**
- **spring.datasource.password**
- **spring.datasource.driver-class-name** - klasa określająca sterownik, który powinien zostać użyty do nawiązania połączenia z bazą danych
- **spring.jpa.show-sql -** zaloguj wygenerowane zapytania do konsoli
- **spring.jpa.open-in-view -** pozwala na dostęp do lazy relacji poza transakcją

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=admin
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.open-in-view=false
```

Konfiguracja

- **spring.jpa.hibernate.ddl-auto -** określa sposób tworzenia bazy danych na podstawie wykrytych przez **Hibernate** metadanych:
  - create - utwórz nową scheme, usuń poprzednia
  - create-drop - utwórz nową scheme, usuń na zakończenie sesji
  - update - zaktualizuj scheme jeżeli różni się od tej wykrytej przez metadane
  - validate - upewnij się, że scheme i metadane są zgodne. Zablokuj działanie aplikacji jeżeli nie
  - none - domyślna wartość, nie podejmuj żadnej akcji

Queries

- **JPQL** - język zapytań przypominający SQL. Pozwala na spójne pisanie zapytań do bazy danych, które następnie zostaną przetłumaczone na dialekt odpowiedni dla bazy danych
- **Query** - pozwala na wykonanie zapytania przy wykorzystaniu **JPQL**
- **NamedQuery** - pozwala na wcześniejsze przygotowanie i proste reużycie zapytań **JPQL**
- **NativeQuery** - pozwala na wykonanie zapytania napisanego bezpośrednio w **SQL**
- Do zapytań możemy przesyłać parametry i ustawiać je zarówno na podstawie nazwy jak i pozycji.

Jeżeli stosujemy zapytania **JPQL** Hibernate jest w stanie wykorzystać tzw. **1st Level Cache** i nie pobierać encji z bazy, które zostały już pobrane w ramach danej transakcji.

```java
@Entity
@NamedQuery(name = "fetchAdultEntities", query = "select s from SimpleEntity s where s.age > 18")
public class SimpleEntity {
```

```java
@GetMapping("/entities")
public void getEntities() {
    TypedQuery<SimpleEntity> jpql = entityManager.createQuery( qlString: "select s from SimpleEntity s where s.id > :age", SimpleEntity.class);
    TypedQuery<SimpleEntity> namedQuery = entityManager.createNamedQuery( name: "fetchAdultEntities", SimpleEntity.class);
    Query nativeQuery = entityManager.createNativeQuery( sqlString: "select * from simple_entity where age > ?1", SimpleEntity.class);

    jpql.setParameter( name: "age", value: 18);
    nativeQuery.setParameter( position: 1, value: 21);
}
```

Criteria API

- Criteria API pozwalają nam na pisanie rozbudowanych i dynamicznych zapytań w programistyczny sposób
- Pozwala na agregację danych
- **CriteriaBuilder** - punkt wejściowy w tworzenie zapytania
- **CriteriaQuery** - pozwala na dodawanie warunków do zapytania
- **Root** - pozwala na dostęp do pól danej encji i jej relacji
- **JPA metamodel** - zapewnia bezpieczeństwo typów podczas wykorzystywania **Criteria API**.

Projection

- W momencie gdy potrzebujemy pobrać encje i wiemy, że nie będziemy zmieniać ich stanu warto jest pobrać jedynie te dane, których tak naprawdę potrzebujemy.
- W tym celu możemy wykorzystać projekcje - specjalnie przygotowane klasy, nadające się tylko do odczytu.
- Projekcje nie są encjami - nie istnieje możliwość ich zapisu do bazy danych.

Chapter 12

Spring Data

Spring Data

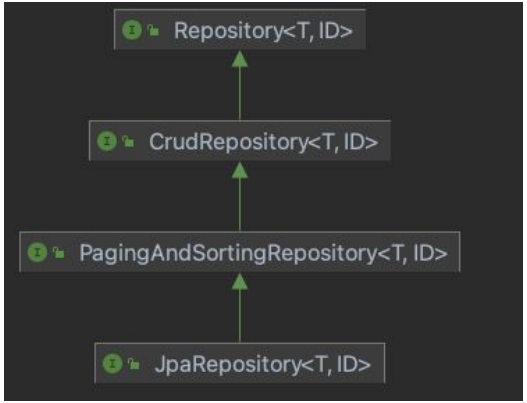
- Grupa projektów dostarczający spójny sposób dostępu do bazy danych, niezależny od rodzaju bazy danych
- Wspiera bazy relacyjne, nierelacyjne i grafowe
- Oparta na koncepcji **Repository**, które pozwalają na uniknięcie większości boilerplate kodu
- Przykładowe projekty:
  - **Spring Data JPA** - pod spodem korzysta z **Hibernate**
  - **Spring Data JDBC** - pod spodem korzysta bezpośrednio z **JDBC**
  - **Spring Data Mongo** - dostarcza własną implementację klas tworzących zapytania do Mongo

Spring Data JPA

- Projekt pozwalający na komunikację z relacyjnymi bazami danych przy pomocy **Hibernate**
- Wspiera wszystkie funkcjonalność JPA oraz dokłada część swoich, np. **Specification**
- Pozwala na łatwe tworzenie transakcji przy pomocy adnotacji **@Transactional**
- Wymaga jedynie następujących zależności:
  - **spring-data-jpa**
  - **driver**, pozwalający połączyć się z bazą danych

Spring Repositories

- Abstrakcja pozwalająca na ominięcie większości kodu związanego z infrastrukturą i **Entity Manager**em
- Oparta na rozszerzaniu interfejsów, a następnie generowaniu ich implementacji przez **Spring Data**
- Interfejsy dostarczają podstawowe metody pozwalające na interakcję z bazą danych takie jak:
  - findAll()
  - findById()
  - save() - używane zarówno dla operacji **INSERT** i **UPDATE**
  - deleteById()




```
public interface SimpleEntityRepository extends JpaRepository<Long, SimpleEntity> {
}
```

Spring Repositories

- Pozwalają na dopisanie własnych deklaracji metod. Na podstawie nazw tych metod zostaną wygenerowane odpowiednie zapytania
- W przypadku bardziej skomplikowanego zapytania można użyć adnotacji **@Query** pozwalającej na użycie **JPQL**. Wspierane są zarówno **native** jak i **named query**.
- Paginacja i sortowanie wyników zapewnione są odpowiednio przez interfejsy **Pageable** i **Sort**.
- Sygnatura metody jest dynamiczna co oznacza, że możemy komponować nasze zapytania z różnych wspieranych obiektów, a Spring postara się wykryć co chcemy osiągnąć

```
public interface SimpleEntityRepository extends JpaRepository<SimpleEntity, Long> {

    List<SimpleEntity> findAllByAgeGreaterThan(int age, Pageable pageable);

    @Query(value = "select s from SimpleEntity s where s.id > :age")
    List<SimpleEntity> selectEntitiesBasedOnAge(int age);
}
```

Spring Repositories

- Na podstawie typu zwracanego Spring domyśla się co chcemy wyciągnąć z bazy danych.
- Przykładowe zwracane typy:
  - **Optional** - zwrócona (bądź nie) zostanie pojedyncza encja
  - **List** - zwrócone zostaną wszystkie encje
  - **Page** - zostanie zwrócona tylko jedna strona wyników (domyślnie o wielkości 20)
  - **Stream** - pozwala na streamowanie i dobieranie rezultatów w trakcie ich procesowania. Pod spodem wykorzystany zostaje **ScrollableResultSet**. Należy pamiętać, aby taki Stream zamknąć po zakończeniu procesowania.
  - **CompletableFuture** - zapytanie do bazy zostanie wykonane w osobnym wątku

Spring Data Specification

- Rozszerzenie koncepcji **Criteria API**
- Pozwala na zdefiniowanie małych, prostych warunków a następnie na ich podstawie komponowanie znacznie bardziej złożonych
- Aby repozytorium mogło korzystać z **Specification** musi dodatkowo rozszerzać interfejs **JpaSpecificationExecutor**

```
public interface SimpleEntityRepository extends JpaRepository<SimpleEntity, Long>, JpaSpecificationExecutor<SimpleEntity> {
```

```
@GetMapping("/adults-from-gdansk")
public List<SimpleEntity> getAdultsFromGdansk() {
    return simpleEntityRepository.findAll(isAdult().and(isFromGdansk()));
}
```

```
public class SimpleEntityConditions {

    private static final String GDANSK_CITY = "Gdańsk";

    public static Specification<SimpleEntity> isFrom(String city) {
        return (root, query, cb) -> cb.equal(root.get(SimpleEntity_.address).get(SimpleEmbeddableAddress_.city), city);
    }

    public static Specification<SimpleEntity> isFromGdansk() {
        return isFrom(GDANSK_CITY);
    }

    public static Specification<SimpleEntity> isAdult() {
        return (root, query, cb) -> cb.greaterThan(root.get(SimpleEntity_.age), y: 17);
    }
}
```

OSIV - kontrowersje

- **Open Session in View** - tryb działania JPA, który tworzy EntityManagerem przed rozpoczęciem obsługi requesta i zamyka go dopiero po wyrenderowaniu odpowiedzi
- Pozwala na dostęp do lazy relacji poza transakcją
- Przez część ekspertów od **Hibernate** uważany za Anti Pattern
- W Spring Boot domyślnie włączony
- Może powodować re użycie Entity pomiędzy transakcjami
- Więcej informacji:
  - https://vladmihalcea.com/the-open-session-in-view-anti-pattern/
  - https://github.com/spring-projects/spring-boot/issues/7107
  - https://blog.maczkowski.dev/2020/02/wzorzec-open-session-in-view-w-spring.html