



# Spring



Placeholder text for the right side of the slide, consisting of multiple lines of blurred content.

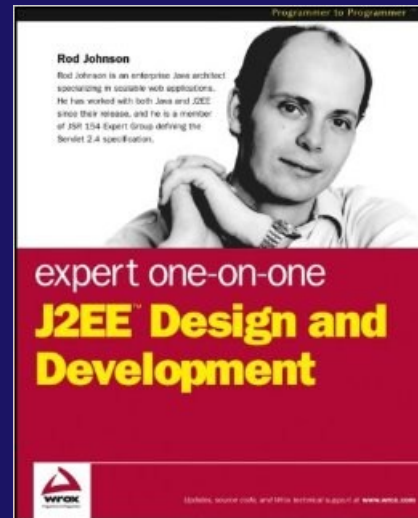
# Spring

Spring to framework a obecnie cały ekosystem umożliwiający tworzenie aplikacji webowych z wykorzystaniem języka Java.

Jest alternatywą dla JEE (Java Enterprise Edition)

Pomysł na Springa powstał podczas pisania książki Expert One-on-One J2EE Design and Development napisanej przez Roda Johnsona w 2002 roku.

Głównym celem Springa jest umożliwienie tworzenia złożonych systemów w łatwy sposób, z pominięciem modeli programowania wymuszających historyczne rozwiązania.



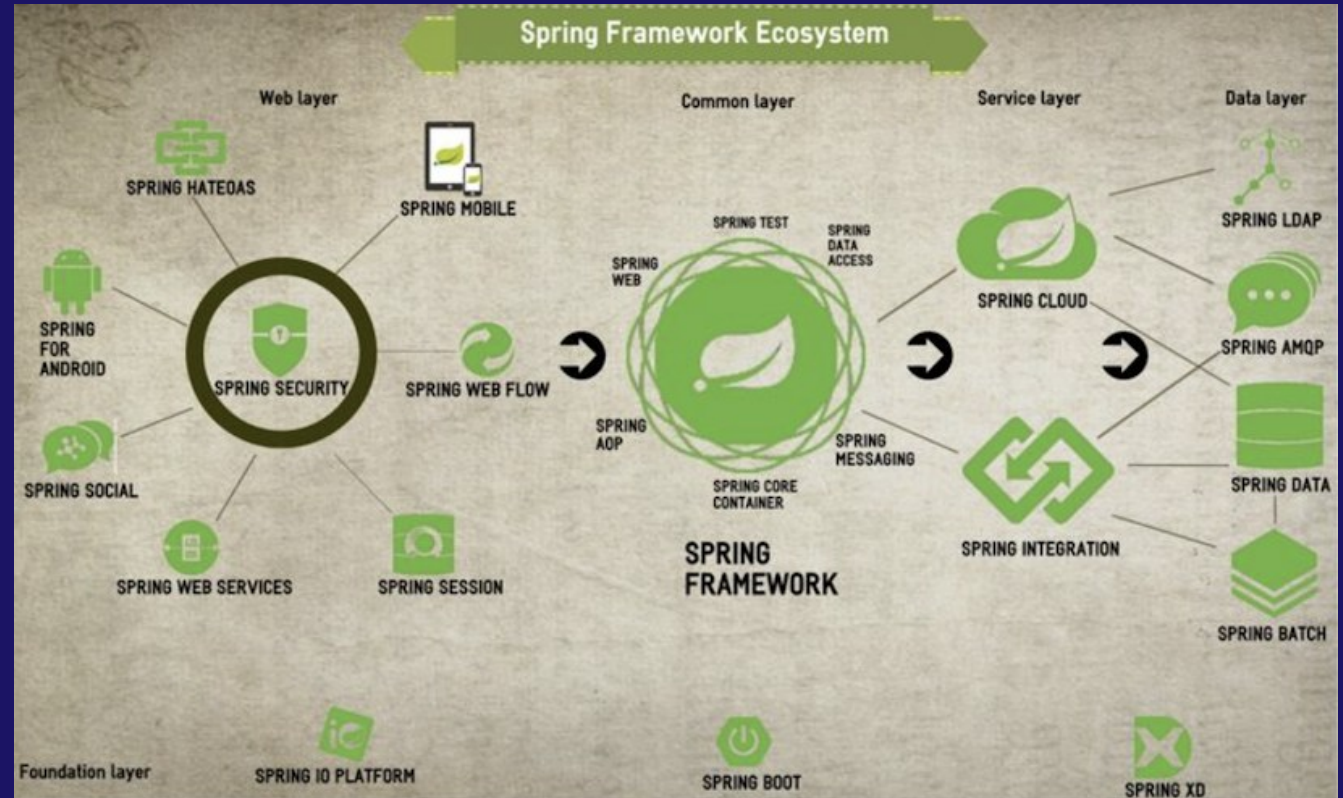
# Spring

---

Spring składa się obecnie z wielu osobnych projektów i modułów – wybieramy te, które są nam potrzebne w naszym projekcie.



# Spring Framework

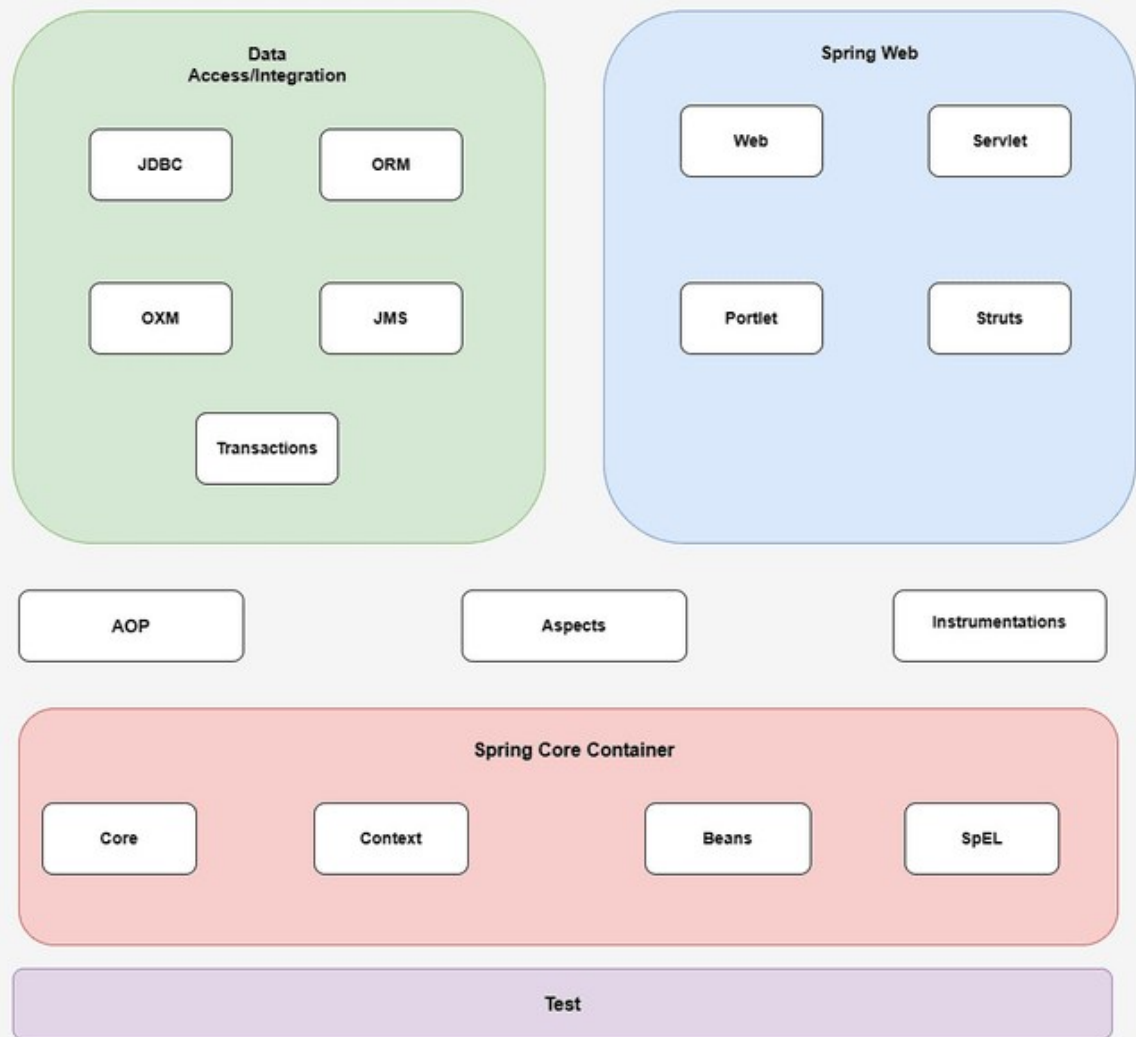


# Core

**Core** – odnosi się do podstawowego modułu kontenera wstrzykiwania zależności.

`org.springframework.beans` i `org.springframework.context` tworzą fundamenty dla zarządzania obiektami i kontekstem w aplikacjach Spring

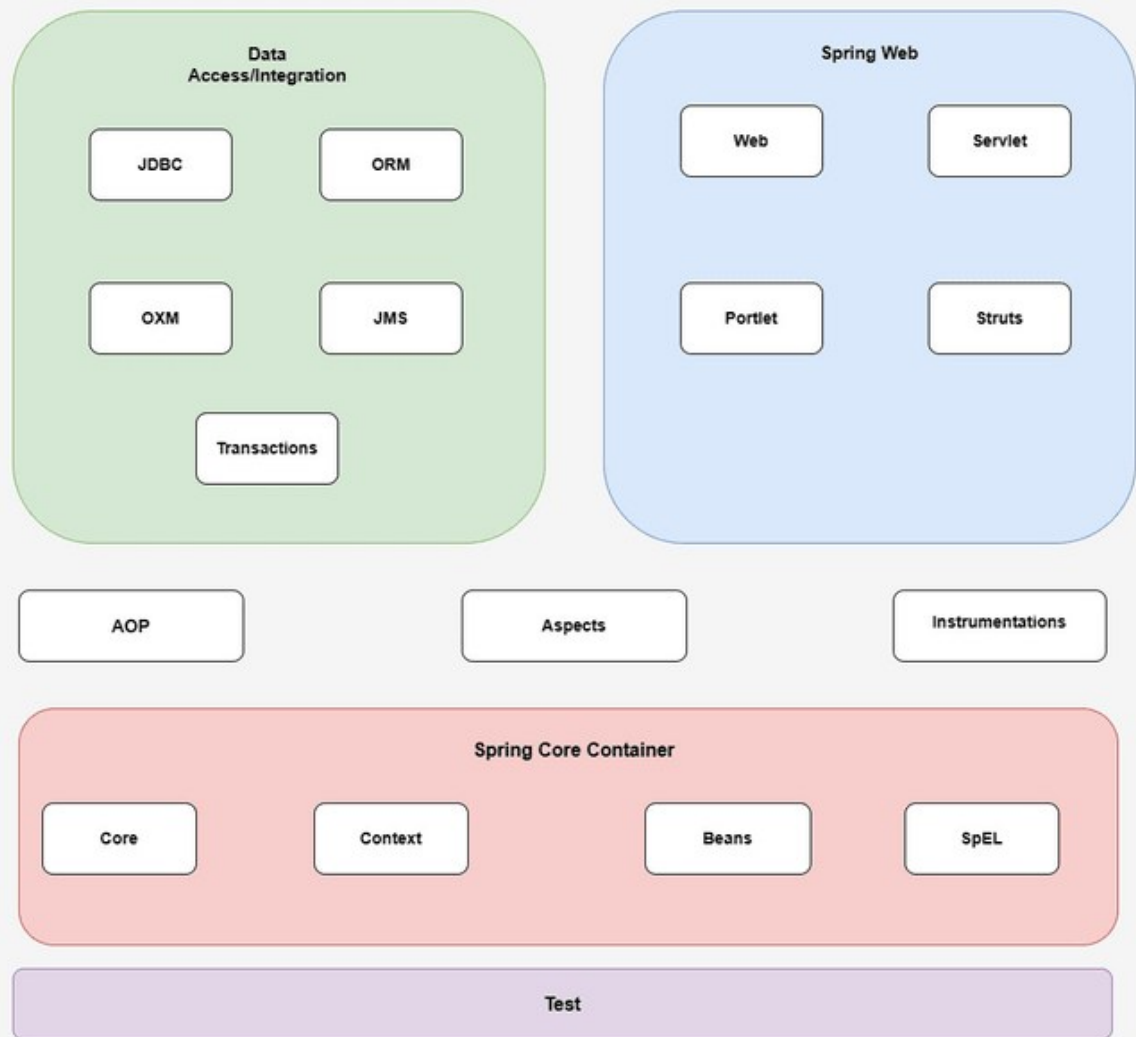
## Spring Framework



# Context

Kontekst aplikacji Spring zarządza instancjami beanów, które są tworzone zgodnie z definicjami zadeklarowanymi przez programistę w konfiguracji aplikacji (XML, adnotacje, Java config). Kontekst zarządza cyklem życia beanów, wstrzykując zależności, zarządzając ich tworzeniem.

## Spring Framework



# Context

Przykład użycia kontekstu aplikacji  
Spring MVC w aplikacji webowej.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         id="WebApp_ID" version="4.0">
  <display-name>mvceexample</display-name>
  <!-- Ustawiamy Dispatcher Servlet na ten który udostępnia nam Spring
Framework -->
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <!-- Ustawiamy kontekst aplikacji: -->
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/mvccontext.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- mapujemy url: -->
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

## Context

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/mvc ">
  <context:component-scan base-package="org.example"/>
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

Przykład użycia kontekstu aplikacji  
Spring MVC w aplikacji webowej.

Servlet jest podstawowym elementem aplikacji biznesowych tworzonych w języku Java, która jest zdolna do przetwarzania żądań HTTP. Servlety wykorzystywane są zarówno w aplikacjach Javy EE jak i tych tworzonych w Spring MVC.





## Context

Aplikacja napisana z wykorzystaniem SpringMVC musi zostać umieszczona w Kontenerze aplikacji webowej.

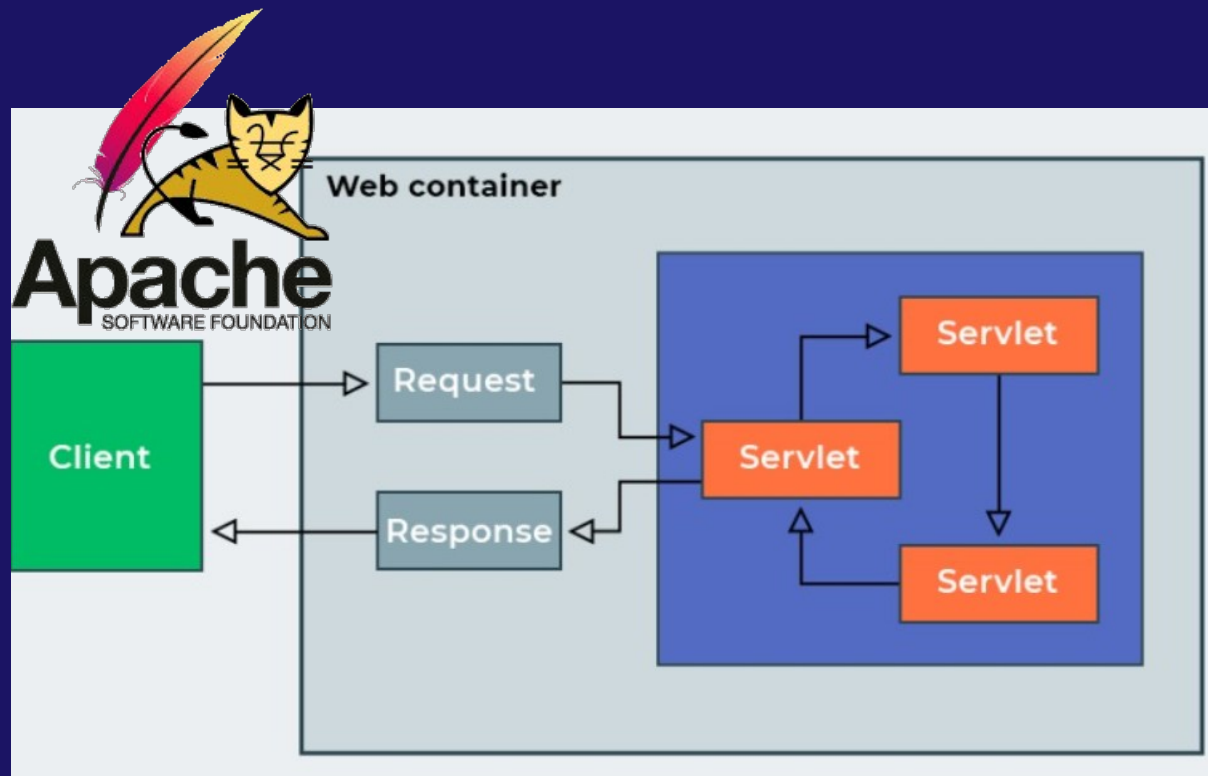
**Funkcje kontenera:**

**1.przetwarza żądanie**

2.Dynamicznie generuje strony HTML z plików JSP.

3.Szyfruje/odszyfrowuje wiadomości HTTPS.

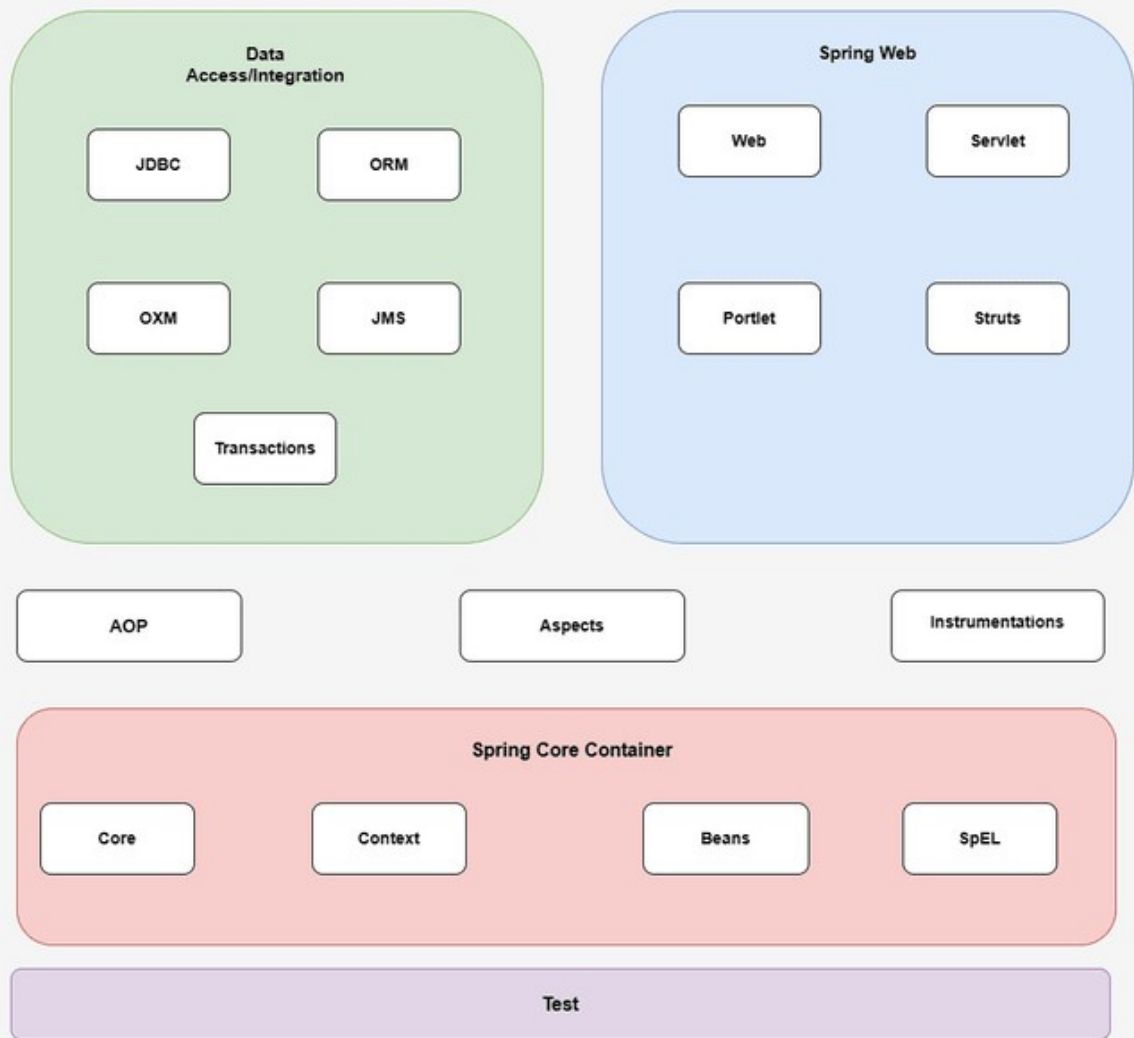
4.Zapewnia ograniczony dostęp do administrowania servletami.



# Data Access

Umożliwia programistom korzystanie z interfejsów API persystencji, takich jak JDBC i Hibernate

## Spring Framework

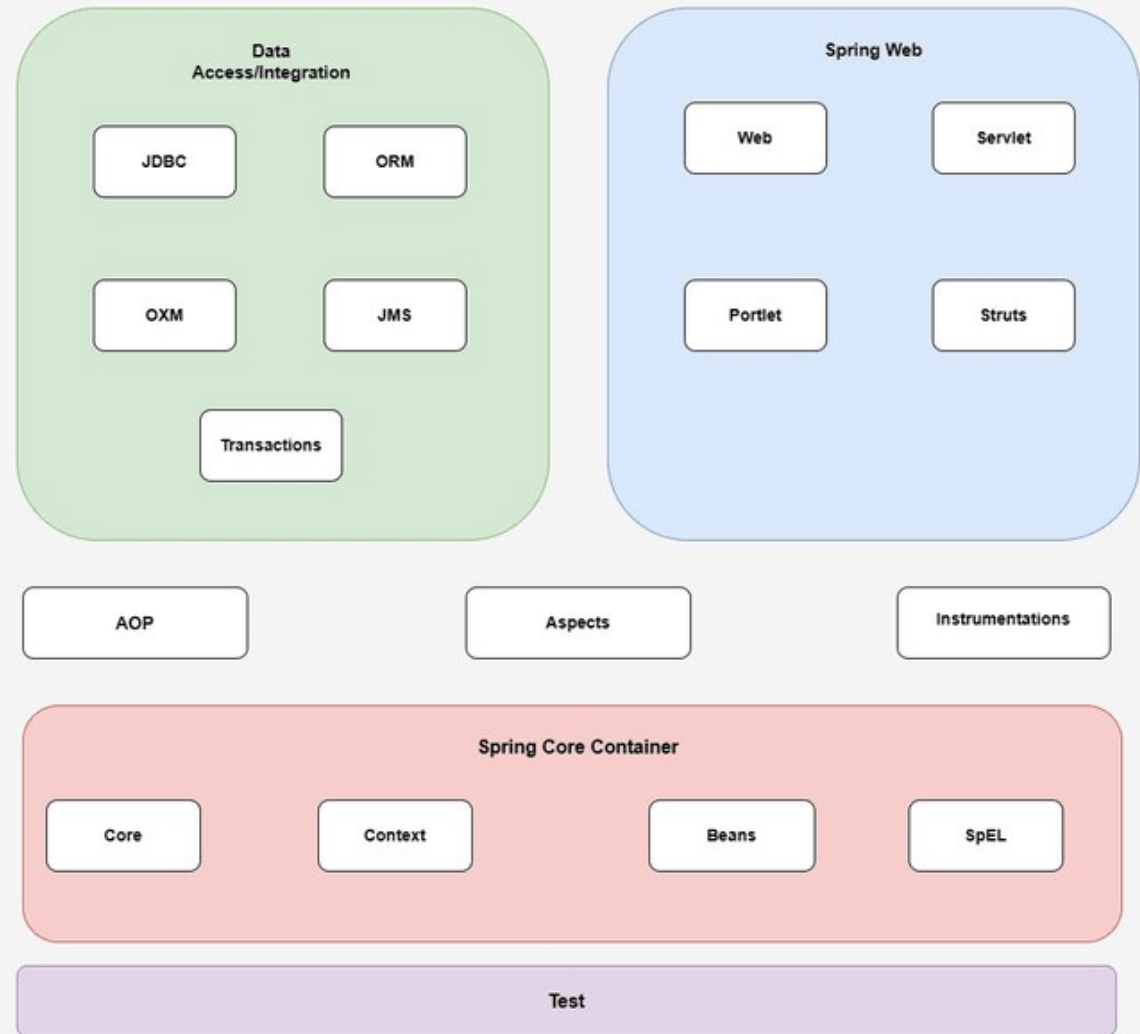


# Spring Web

Spring MVC Umożliwia budowanie aplikacji internetowych w oparciu o architekturę MVC. Wszystkie żądania użytkownika przechodzą najpierw przez kontroler, a następnie są przesyłane do różnych widoków wraz z danymi z modelu, takich jak stron JSP, lub Thymeleaf.

Spring Web umożliwia także tworzenie aplikacji REST(Representational State Transfer).

## Spring Framework



# Projekty Springa

Spring Framework  
Spring Security  
Spring Boot



## Spring Boot

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.



## Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.



## Spring Data

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.



## Spring Cloud

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.



## Spring Cloud Data Flow

Provides an orchestration service for composable data microservice applications on modern runtimes.



## Spring Security

Protects your application with comprehensive and extensible authentication and authorization support.



## Spring Authorization Server

Provides a secure, light-weight, and customizable foundation for building OpenID Connect 1.0 Identity Providers and OAuth2 Authorization Server products.



## Spring for GraphQL

Spring for GraphQL provides support for Spring applications built on GraphQL Java.



## Spring Session

Provides an API and implementations for managing a user's session information.

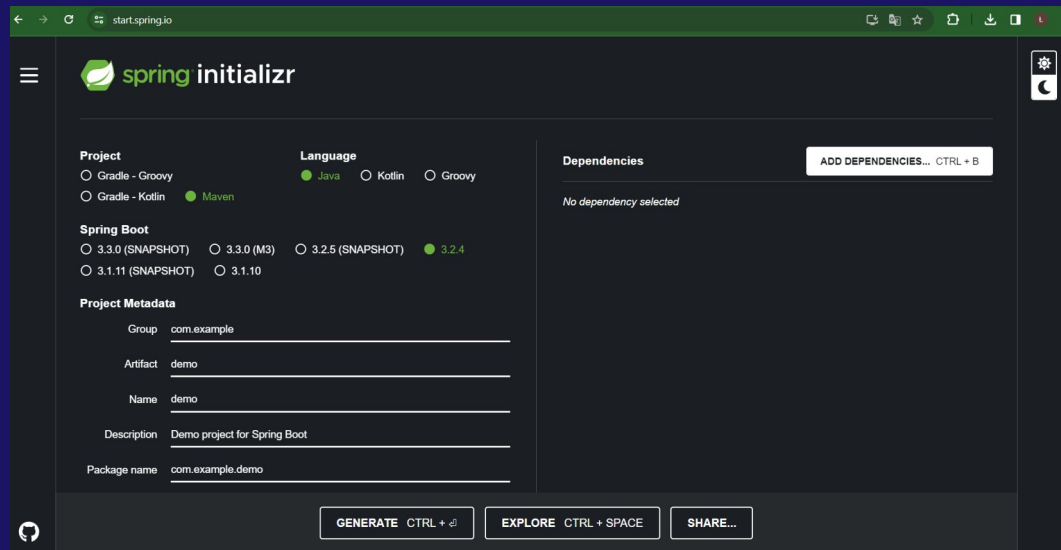


## Spring Integration

Supports the well-known Enterprise Integration Patterns through lightweight messaging and declarative adapters.

# Projekty Springa

Spring jest lekki i modułowy.  
Obecnie zależności ekosystemu można „wyklikać” używając SpringInitializr:



The screenshot shows the Spring Initializr web application in a browser window. The interface is dark-themed and contains the following sections:

- Project:** Radio buttons for `Gradle - Groovy`, `Gradle - Kotlin`, and `Maven` (selected).
- Language:** Radio buttons for `Java` (selected), `Kotlin`, and `Groovy`.
- Spring Boot:** Radio buttons for `3.3.0 (SNAPSHOT)`, `3.3.0 (M3)`, `3.2.5 (SNAPSHOT)`, `3.2.4` (selected), `3.1.11 (SNAPSHOT)`, and `3.1.10`.
- Project Metadata:** Text input fields for `Group` (com.example), `Artifact` (demo), `Name` (demo), `Description` (Demo project for Spring Boot), and `Package name` (com.example.demo).
- Dependencies:** A section with the text "No dependency selected" and a button "ADD DEPENDENCIES... CTRL + B".

At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE...".

# Spring Boot

Spring Boot jest projektem w ekosystemie Spring, ułatwiającym tworzenie aplikacji Spring, minimalizując potrzebę konfiguracji i zarządzania zależnościami.

The screenshot shows the Spring Initializr web application interface. The browser address bar displays 'start.spring.io'. The interface is divided into several sections:

- Project:** Includes radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', and 'Maven' (selected).
- Language:** Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
- Spring Boot:** Includes radio buttons for versions: '3.5.0 (SNAPSHOT)', '3.5.0 (RC1)', '3.4.6 (SNAPSHOT)', '3.4.5' (selected), '3.3.12 (SNAPSHOT)', and '3.3.11'.
- Project Metadata:** A form with fields for 'Group' (com.umcsuser), 'Artifact' (car-rent), 'Name' (car-rent), 'Description' (Car Rent project), and 'Package name' (com.umcsuser.car-rent). It also includes 'Packaging' options (Jar selected, War) and 'Java' version options (24, 21, 17 selected).
- Dependencies:** A list of dependencies with an 'ADD DEPENDENCIES... CTRL + B' button. The listed dependencies are:
  - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
  - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
  - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
  - PostgreSQL Driver** (SQL): A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
  - Spring Boot DevTools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
  - Validation** (JIO): Bean Validation with Hibernate validator.

At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and an ellipsis menu.

# Spring Boot

## Cechy:

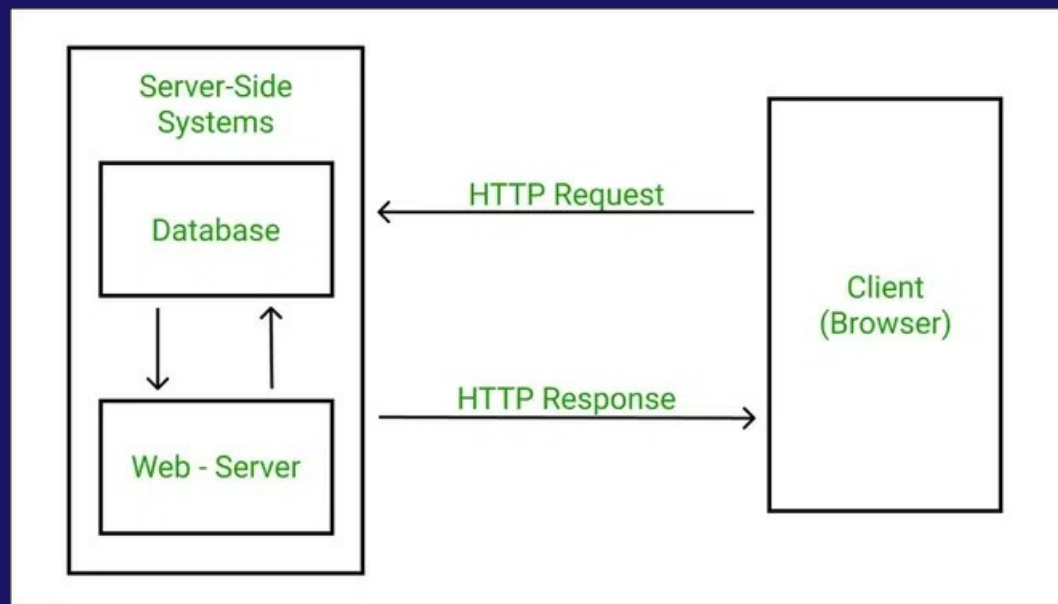
- Automatyczna konfiguracja
- posiada gotowe zestawy zależności „starters”
- wbudowany kontener aplikacji webowych.

```
@SpringBootApplication
@RestController
public class SimpleApp {
    public static void main(String[] args) {
        SpringApplication.run(SimpleApp.class,
args);
    }
    @GetMapping("/hello")
    public String helloWorld() {
        return String.format("Hello world");
    }
}
```

<https://bykowski.pl/spring-i-spring-boot/>

# Protokół HTTP(Hypertext Transfer Protocol)

HTTP jest protokołem warstwy aplikacji, który działa na podstawie modelu klient-serwer. Klient (np. przeglądarka internetowa) wysyła żądanie HTTP do serwera, a serwer odpowiada na to żądanie, przesyłając odpowiednie dane.





# Protokół HTTP(Hypertext Transfer Protocol)

## Żądanie:

**Metoda** - określa rodzaj żądania, na przykład **GET**, **POST**, **PUT**, **DELETE**.

**Adres URL** - określa lokalizację zasobu, do którego odnosi się żądanie.

**Nagłówki** - zawierają dodatkowe informacje o żądaniu, takie jak typ danych, język, dane uwierzytelniające itp.

**Treść** - aktualne dane przesyłane w odpowiedzi.

## Odpowiedź:

**Kod stanu** - informuje o rezultacie żądania

**Nagłówki** - zawierają dodatkowe informacje o odpowiedzi, takie jak typ danych, długość treści, dane uwierzytelniające itp.

**Treść** - aktualne dane przesyłane w odpowiedzi.

# Protokół HTTP

## Kody odpowiedzi:

- 1xx - Informacyjne
- 2xx – Sukces
- 3xx – Przekierowania
- 4xx - Błędy klienta
- 5xx - Błędy serwera



# Aplikacja wypożyczalni

Spring boot zapewni nam odpowiednie wersje kompatybilnych ze sobą modułów.

Używając java17 należy użyć kompatybilnej dla niej wersji Spring Boot Starter Parent.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.4.5</version>
  <relativePath/>
</parent>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <annotationProcessorPaths>
          <path>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.38</version>
          </path>
          <path>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <version>${project.parent.version}</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>
```

# Aplikacja wypożyczalni

Wszystkie bean'y będziemy tworzyć przez adnotacje w już istniejących klasach (nowych)

Konfiguracja również wykona się automatycznie, na podstawie ustawień w pliku application.properties

```
import org.hibernate.SessionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan("org.example")
//@EnableScheduling
public class AppConfigConfiguration {
    @Bean
    public SessionFactory sessionFactory() {
        return new org.hibernate.cfg.Configuration().configure().buildSessionFactory()
    }
}
```

```
spring.application.name=car-rent
server.port=8080
spring.datasource.url=${DB_CONNECT_URL}
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

# Integracja z aplikacją wypożyczalni

Klasa, która odpala nam aplikację SpringBoot.

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

# Konfiguracja

Automatycznie konfiguruje aplikację springa – konfiguruje EntityManagerFactory, skanuje też komponenty (@Service, @Repository, @RestController, @Entity itp.) w pakiecie głównym i jego podpakietach.

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

## Dodatkowe klasy

Będziemy także potrzebowali  
3 nowych typów klas:

Controller(**@RestController**)

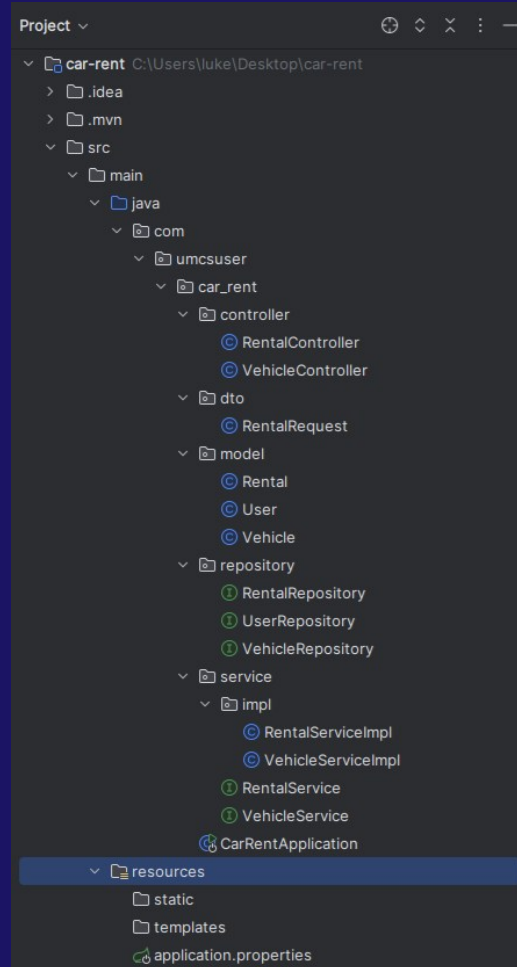
Dto

Service(**@Service**)

Klasy **Service** – powinny zawierać logikę odseparowaną od innych warstw aplikacji, takich jak kontrolery i dostęp do danych

**RestController** – obsługa żądań i konwersja odpowiedzi do formatu JSON

**Dto** – obiekt do przenoszenia danych, w prostych przypadkach można użyć encji



# @Repository

Użyjemy nowego typu Repozytorium – gdzie interfejs będzie automatycznie implementowany:

```
import  
org.springframework.data.jpa.repository.  
JpaRepository;
```

Z opcjonalną adnotacją:

**@Repository**.

Spring skanuje kontekst aplikacji w poszukiwaniu beanów, które pasują do parametrów konstruktora w klasie, gdzie użyto **@Autowired**. Jeśli znajdzie pasujące beany, automatycznie je wstrzykuje.

W nowym repozytorium jest to zbędne!

Stare repozytorium:

```
@Repository  
public class UserDao implements IUserRepository  
{  
    SessionFactory sessionFactory;  
  
    @Autowired  
    private UserDao(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
    //...  
}
```

Nowe Repozytorium:

```
@Repository  
public interface UserRepository extends JpaRepository<User, String> {  
    // Methods findAll(), findById(),  
    //save(), deleteById() from JpaRepository.  
    Optional<User> findByLogin(String login);  
}
```



# @Repository

## Repozytorium Rental:

```
@Repository
public interface RentalRepository extends JpaRepository<Rental, String> {
    // Methods findAll(), findById(), save(), deleteById() from JpaRepository.
    Optional<Rental> findByVehicleIdAndReturnDateIsNull(String vehicleId);
    Optional<Rental> findByVehicleIdAndUserIdAndReturnDateIsNull(String vehicleId, String userId);
    boolean existsByVehicleIdAndReturnDateIsNull(String vehicleId);
    @Query("SELECT r.vehicle.id FROM Rental r WHERE r.returnDate IS NULL")
    Set<String> findRentedVehicleIds();
}
```

## Repozytorium Vehicle:

```
@Repository
public interface VehicleRepository extends JpaRepository<Vehicle, String> {
    // Methods findAll(), findById(), save(), deleteById() from JpaRepository.
    List<Vehicle> findByIsActiveTrue();
    Optional<Vehicle> findByIdAndIsActiveTrue(String id);
    List<Vehicle> findByIsActiveTrueAndIdNotIn(Set<String> rentedVehicleIds);
}
```

# @RestController

Następnie należy stworzyć klasę kontrolera. Np. `VehicleController` z addnotacją `@RestController` – klasa zajmie się obsługą żądań związanych z pojazdami.

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

    private final VehicleService vehicleService;

    @Autowired
    public VehicleController(VehicleService vehicleService) {
        this.vehicleService = vehicleService;
    }

    //...
}
```

## @Controller

Klasa Service dotycząca pojazdów będzie zawierała metody o analogicznej nazwie co te w Repozytorium Będzie pobierało odpowiednie dane od kontrolera, wywoływało metody na repozytorium i przekazywało dane z powrotem do kontrolera.

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

    private final VehicleService vehicleService;

    @Autowired
    public VehicleController(VehicleService vehicleService) {
        this.vehicleService = vehicleService;
    }

    //...
}
```

Do wymiany danych użyjemy DTO lub encji.

`@RequestMapping("/api/vehicles")`

W klasie Oznacza wywołanie metod w odpowiedzi na konkretne żądanie zaczynające się od „/api/vehicles”. Plus następne części url dodane przy metodach.

## @Controller->@Service->@Repository->@Service->@Controller

Metoda getAllVehicles() wykona na userService metodę findAll()  
Następnie VehicleService wykona metodę findAll() na repozytorium

Kontroler przekazuje kolekcję obiektów przekonwertowanych na format json.

Z adnotacją @Transactional, Spring automatycznie:

otwiera transakcję na początku metody,  
zamyka (zatwierdza lub wycofuje) transakcję po zakończeniu metody (commit lub rollback).

readOnly = true - w tym wypadku transakcja tylko do odczytu

```
@GetMapping // Mapowanie GET na główny URL /api/vehicles
public List<Vehicle> getAllVehicles() {
    return vehicleService.findAll();
}

@Service
public class VehicleServiceImpl implements VehicleService {

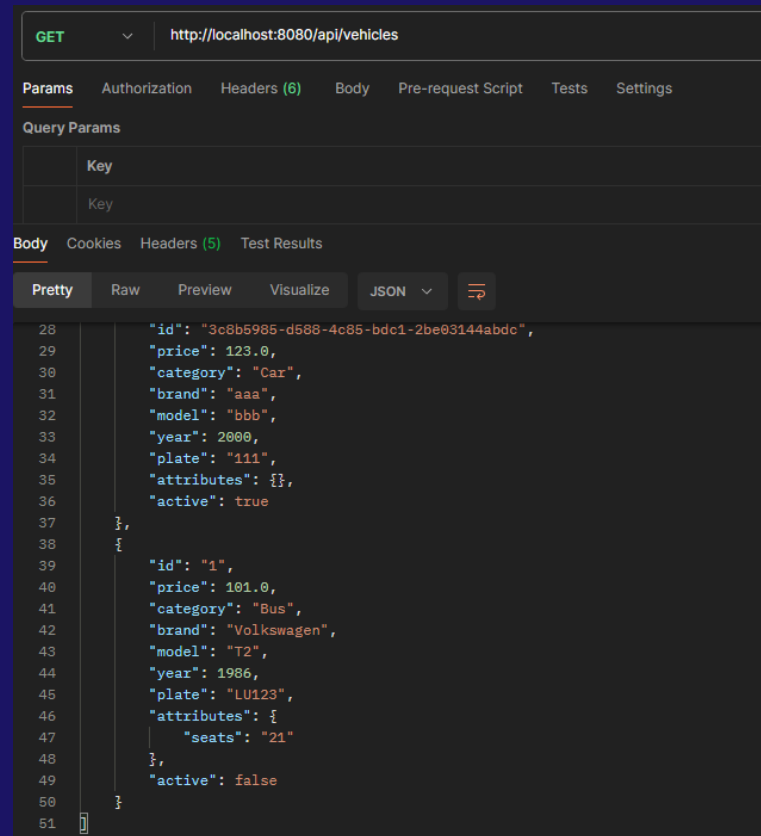
    private final VehicleRepository vehicleRepository;
    private final RentalRepository rentalRepository;
    @Autowired
    public VehicleServiceImpl(VehicleRepository vehicleRepository,
                             RentalRepository rentalRepository) {
        this.vehicleRepository = vehicleRepository;
        this.rentalRepository = rentalRepository;
    }

    @Override
    @Transactional(readOnly = true)
    public List<Vehicle> findAll() {
        return vehicleRepository.findAll();
    }
}
```

## @Service

Metoda getAllVehicles() wykona na userService metodę findAll()  
Następnie VehicleService wykona metodę findAll() na repozytorium.

Kontroler przekazuje kolekcję obiektów przekonwertowanych na format json.



The screenshot shows a REST client interface with a GET request to `http://localhost:8080/api/vehicles`. The response is displayed in JSON format, showing a list of two vehicles. The first vehicle is a Car with ID `3c8b5985-d588-4c85-bdc1-2be03144abdc`, price 123.0, brand 'aaa', model 'bbb', year 2000, plate '111', and is active. The second vehicle is a Bus with ID `1`, price 101.0, brand 'Volkswagen', model 'T2', year 1986, plate 'LU123', 21 seats, and is not active.

```
28      "id": "3c8b5985-d588-4c85-bdc1-2be03144abdc",
29      "price": 123.0,
30      "category": "Car",
31      "brand": "aaa",
32      "model": "bbb",
33      "year": 2000,
34      "plate": "111",
35      "attributes": {},
36      "active": true
37    },
38    {
39      "id": "1",
40      "price": 101.0,
41      "category": "Bus",
42      "brand": "Volkswagen",
43      "model": "T2",
44      "year": 1986,
45      "plate": "LU123",
46      "attributes": {
47        "seats": "21"
48      },
49      "active": false
50    }
51  ]
```

## @Controller

Do reprezentowania całej odpowiedzi HTTP, która obejmuje zarówno dane (ciało odpowiedzi), jak i status HTTP, oraz opcjonalnie dodatkowe nagłówki HTTP można użyć:

`ResponseEntity<Vehicle>`

Poza danymi przekazujemy status 200 OK, lub 404 jak nie ma pojazdu.

```
@GetMapping("/{id}")
public ResponseEntity<Vehicle> getVehicleById(@PathVariable String id) {
    logger.info("Request received for vehicle with ID: {}", id);
    return vehicleService.findById(id)
        .map(vehicle -> {
            return ResponseEntity.ok(vehicle);
            // 200 OK z obiektem Vehicle
        })
        .orElseGet(() -> {
            return ResponseEntity.notFound().build();//404
        });
}
```

## Tworzenie nowego pojazdu:

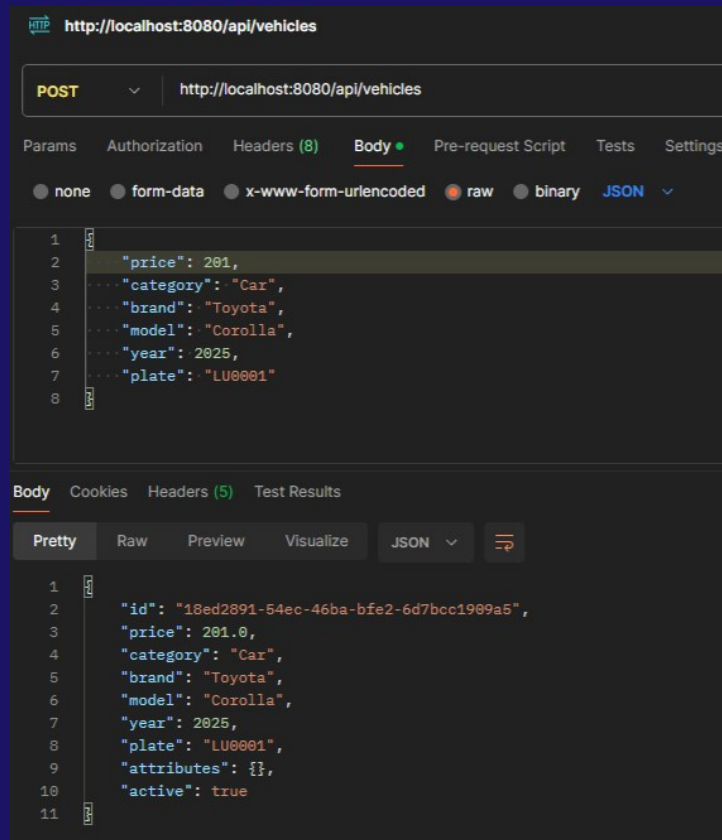
```
@PostMapping
public ResponseEntity<Vehicle> addVehicle(@RequestBody Vehicle vehicle) {
    try {
        Vehicle savedVehicle = vehicleService.save(vehicle);
        return ResponseEntity.status(HttpStatus.CREATED).body(savedVehicle);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



```
@Override
@Transactional
public Vehicle save(Vehicle vehicle) {
    if (vehicle.getId() == null || vehicle.getId().isBlank()) {
        vehicle.setId(UUID.randomUUID().toString());
        vehicle.setActive(true);
    }
    Vehicle savedVehicle = vehicleRepository.save(vehicle);
    return savedVehicle;
}
```



```
@Repository
public interface VehicleRepository
//...save from JpaRepository
```



# Wypożyczenie pojazdu z DTO

**Dto** – obiekt do przenoszenia danych , w wypadku wypożyczenia pojazdu prześlemy vehicleId i userId

```
public class RentalRequest {  
    public String vehicleId;  
    public String userId;  
}
```

W RentalController:

```
@PostMapping("/rent")  
public ResponseEntity<Rental> rentVehicle(@RequestBody RentalRequest rentalRequest) {  
    if (rentalRequest.vehicleId == null || rentalRequest.userId == null) {  
        return ResponseEntity.badRequest().build();  
    }  
  
    try {  
        Rental rental = rentalService.rent(rentalRequest.vehicleId, rentalRequest.userId);  
        return ResponseEntity.status(HttpStatus.CREATED).body(rental);  
    } catch (Exception e) {  
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();  
    }  
}
```



# Wypożyczenie pojazdu z DTO

Dto – obiekt do przenoszenia danych , w wypadku wypożyczenia pojazdu przekażemy vehicleId i userId

## W RentalService:

```
@Override
@Transactional
public Rental rent(String vehicleId, String userId) {
    if (!vehicleService.isAvailable(vehicleId)) {
        throw new IllegalStateException("Vehicle " + vehicleId + " is not available for rent.");
    }
    Vehicle vehicle = vehicleRepository.findById(vehicleId)
        .orElseThrow(() -> new EntityNotFoundException("Vehicle consistency error. ID: " + vehicleId));
    User user = userRepository.findById(userId)
        .orElseThrow(() -> {
            return new EntityNotFoundException("User not found with ID: " + userId);
        });

    Rental newRental = Rental.builder()
        .id(UUID.randomUUID().toString())
        .vehicle(vehicle)
        .user(user)
        .rentDate(LocalDate.now())
        .returnDate(null)
        .build();
    Rental savedRental = rentalRepository.save(newRental);
    return savedRental;
}
```

Uwaga! W SpringBoot można z powodzeniem użyć LocalDateTime w encji Rental!

```
@Column(name = "rent_date", nullable = false)
private LocalDateTime rentDate;
```

```
@Column(name = "return_date")
private LocalDateTime returnDate;
```

Konwersja istniejących danych w bazie:

```
ALTER TABLE rental
ALTER COLUMN rent_date TYPE TIMESTAMPTZ WITHOUT TIME ZONE
USING rent_date::timestamp without time zone,
ALTER COLUMN return_date TYPE TIMESTAMPTZ WITHOUT TIME ZONE
USING return_date::timestamp without time zone;
```

# Wypożyczenie pojazdu z DTO

POST http://localhost:8080/api/rentals/rent

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON

```
1 123
2 "vehicleId": "2",
3 "userId": "fc6e600b-e89e-42e2-b2b7-ca18edb68924"
4 123
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 123
2 "id": "d26b684b-3e02-42bb-a3ac-e204a19f9bb1",
3 "vehicle": {
4   "id": "2",
5   "price": 200.0,
6   "category": "Motorcycle",
7   "brand": "Honda",
8   "model": "CBR600",
9   "year": 2016,
10  "plate": "LU456",
11  "attributes": {
12    "drive": "chain",
13    "licence_category": "A"
14  },
15  "active": true
16 },
17 "user": {
18   "id": "fc6e600b-e89e-42e2-b2b7-ca18edb68924",
19   "login": "lukasz",
20   "password": "$2a$10$7MUYZR8FP8ebLzfqwMK85ejBke/TNp/Yme9NrC0XxRy.NAFde2vEm",
21   "role": "ADMIN"
22 },
23 "rentDate": "2025-04-30T01:11:58.4782335",
24 "returnDate": null
25 123
```

# Zadanie:

## 1. Implementacja serwisów:

```
public interface VehicleService {  
    List<Vehicle> findAll();  
    List<Vehicle> findAllActive();  
    Optional<Vehicle> findById(String id);  
    Vehicle save(Vehicle vehicle);  
    List<Vehicle> findAvailableVehicles();  
    List<Vehicle> findRentedVehicles();  
    boolean isAvailable(String vehicleId);  
    // "it should be soft delete"  
    void deleteById(String id);  
}
```

```
public interface RentalService {  
    boolean isVehicleRented(String vehicleId);  
    Optional<Rental> findActiveRentalByVehicleId(String vehicleId);  
    Rental rent(String vehicleId, String userId);  
    boolean returnRental(String vehicleId, String userId);  
    List<Rental> findAll();  
}
```

## 2. Użycie serwisów w kontrolerach:

VehicleController

RentalController

# **Dziękuję za uwagę!**

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.