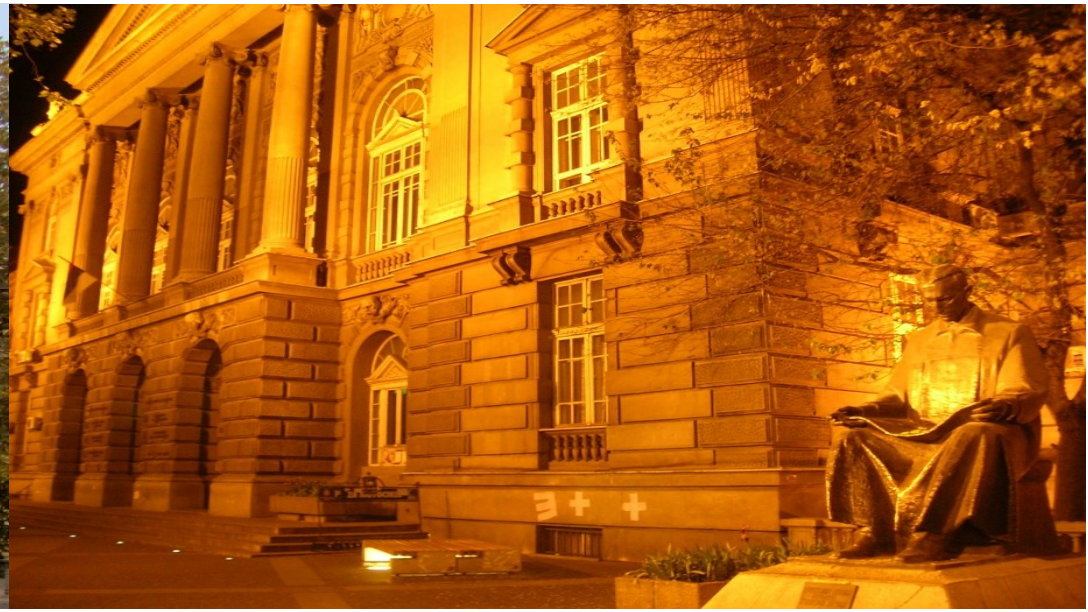
Univerzitet u Beogradu – Elektrotehnički fakultet

Spring Boot



Predavač: Prof. dr Dražen Drašković

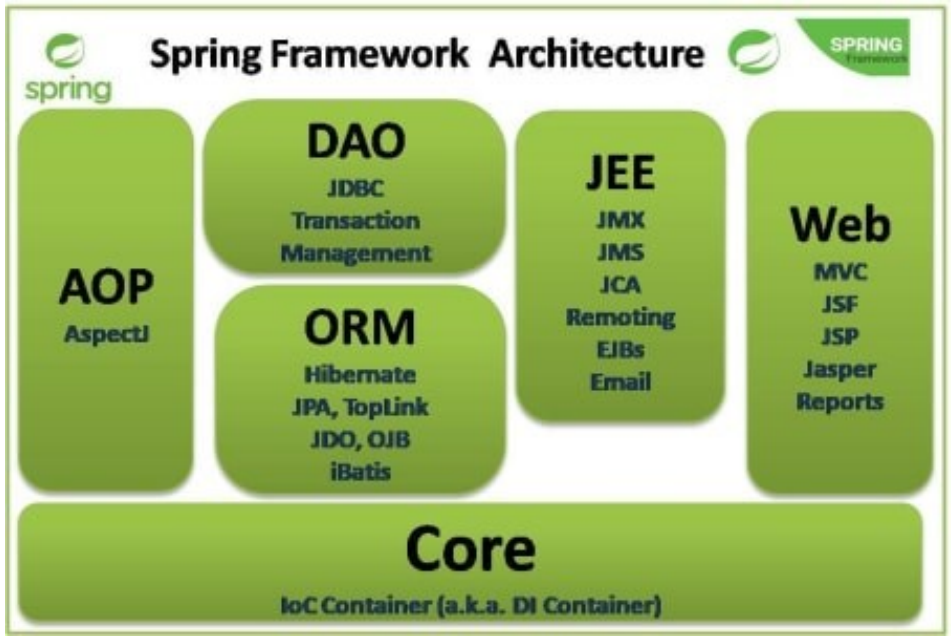
Beograd, april 2024. godine

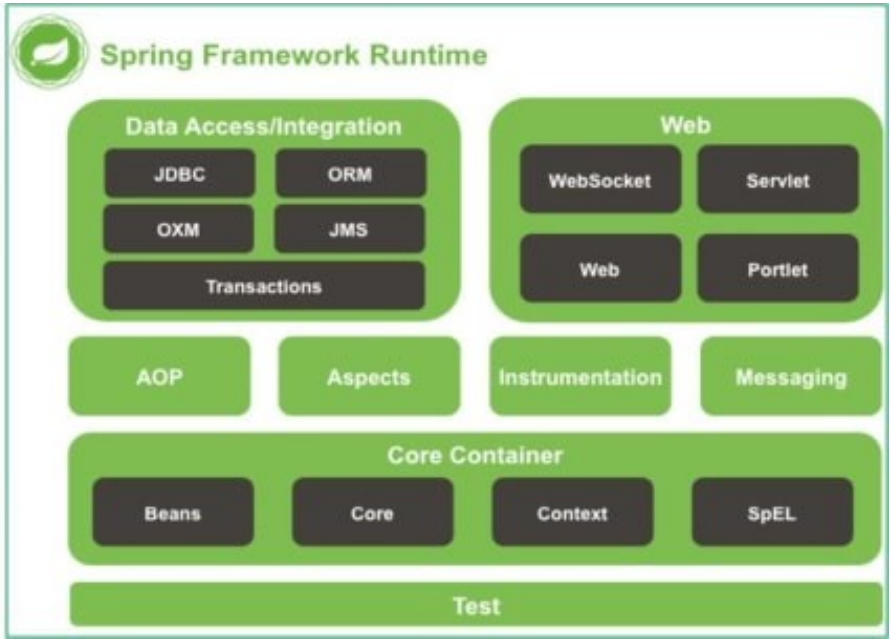
Sadržaj

- Kako pojednostaviti razvoj Spring aplikacija?
- Osnovne karakteristike Spring Boot
- Podešavanje radnog prostora
- Arhitektura slojeva u Spring Boot aplikaciji
- Povezivanje Spring Boot sa bazama podataka i korišćenje JPA
- Testiranje Spring Boot aplikacija

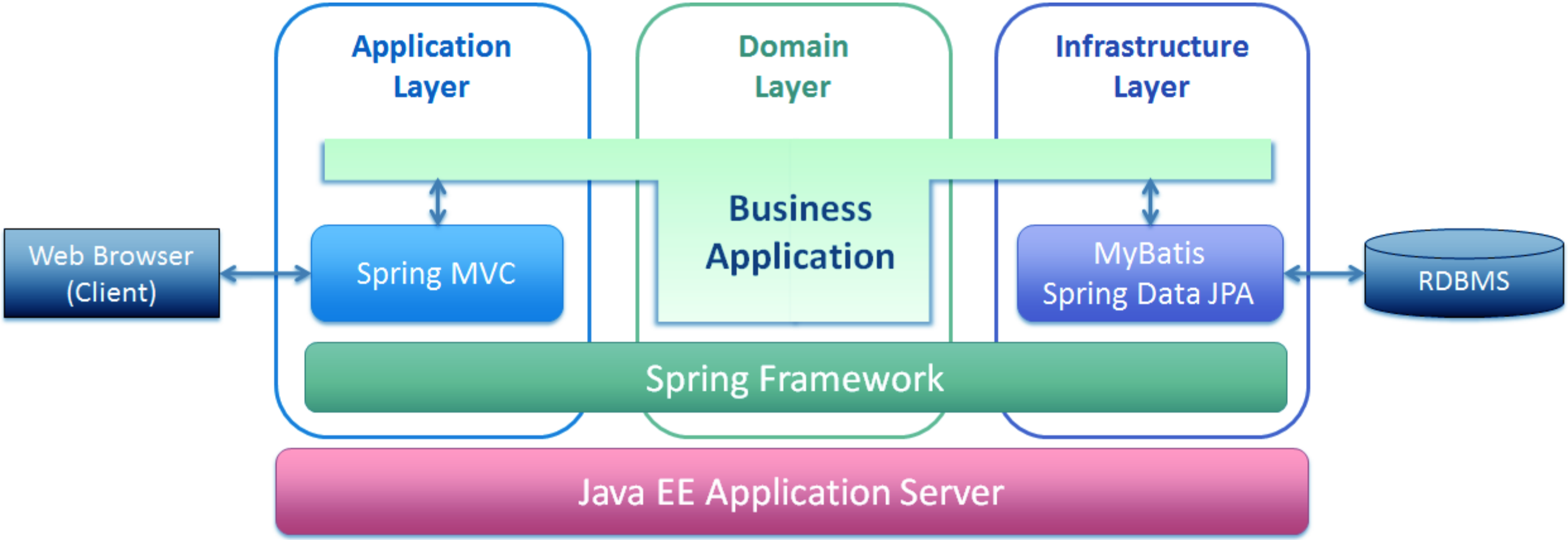
Spring Framework arhitektura

- Aspektno-orijentisano programiranje
- MVC - HTTP/servlet-baziran radni okvir
- Upravljanje transakcijama
- Jezgro kontejner - pruža ubrizgavanje zavisnosti (DI/IoC), sadrži BeanFactory,...
- Kontekst aplikacije - modul koji pruža različite usluge na nivou servisa, raspoređivanja, e-poštu, itd.
- Svaka EE aplikacija komunicira sa bazom: Spring DAO obezbeđuje apstrakciju preko JDBC
- Spring ORM - alati za objektno-relaciono mapiranje

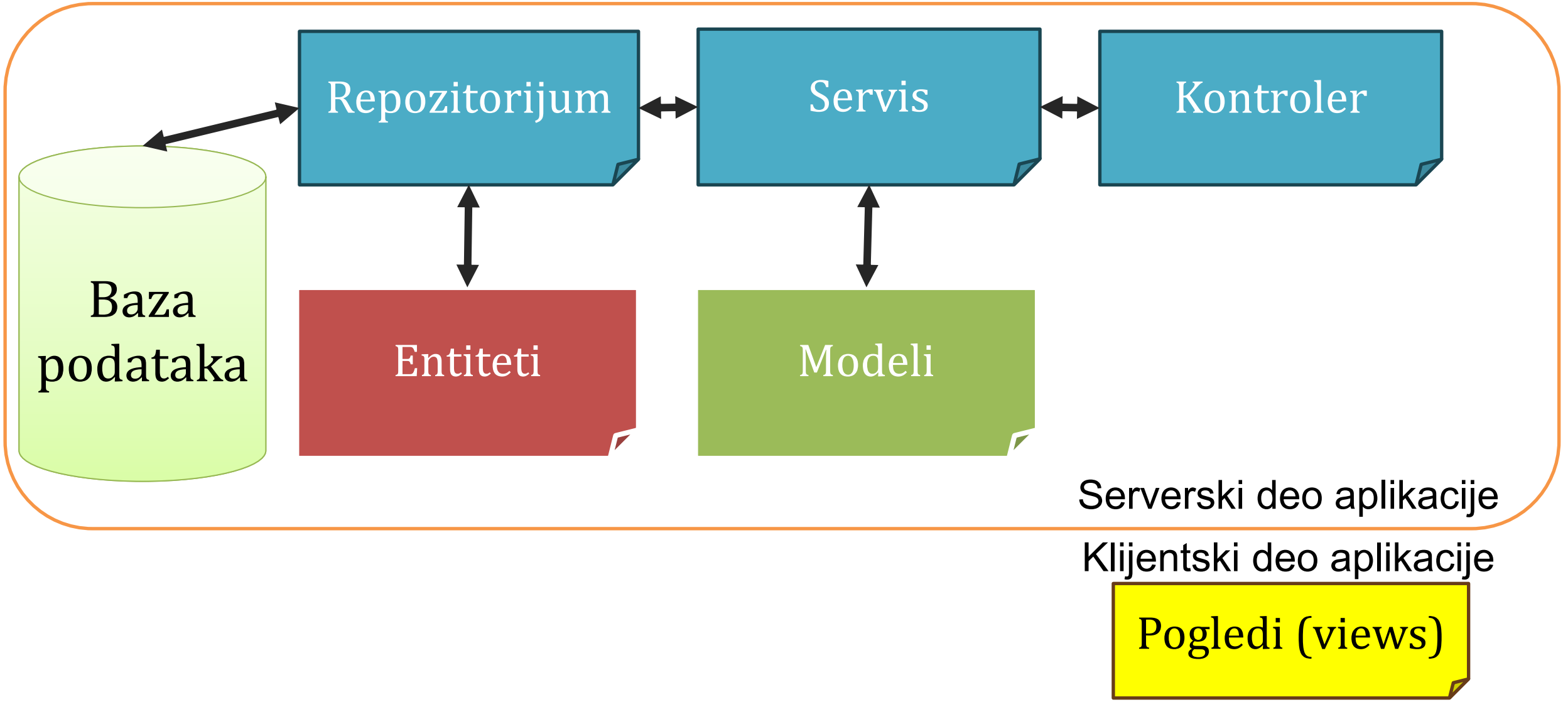




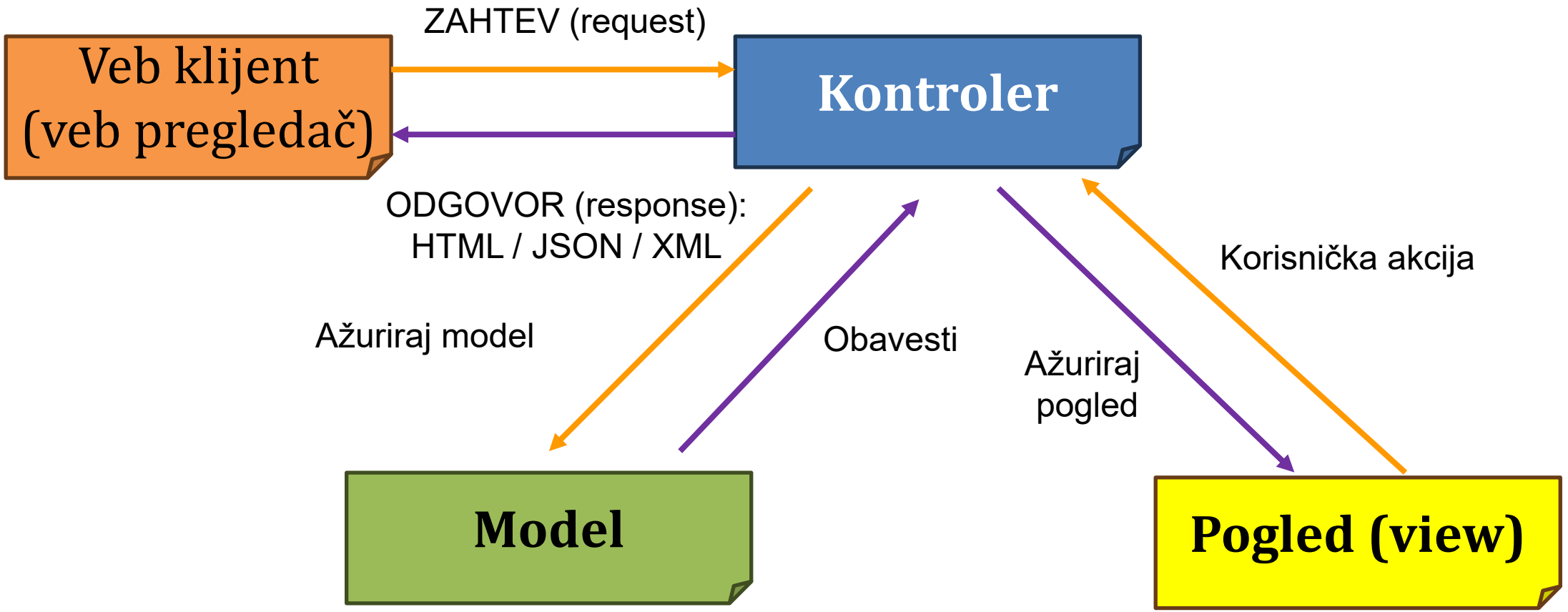
Modularnost Spring aplikacije



Arhitektura aplikacije Spring



MVC kontrola toka



Spring i Spring Boot

- Spring je počeo kao alternativa Java EE (Enterprise Edition / J2EE) - umesto Java Beans, koristiti injektiranu zavisnost i aspektno-orijentisano programiranje
- Lak za pisanje komponenti, težak za konfigurisanje
- Početak sa XML fajlovima, zatim anotacije (od Spring 2.5), pa Java bazirano konfigurisanje (od Springa 3.0)
- Upravljanje transakcijama i Spring MVC zahtevalo eksplicitnu konfiguraciju
- Upravljanje zavisnostima nezahvalan zadatak - koje biblioteke i koje verzije?
- Spring Boot - nudi novu paradigmu za razvoj Spring aplikacija, sa minimalnim naporom (minimalno konfigurisanje)
- Spring Boot nam donosi: konvenciju za konfigurisanje; standardizaciju za mikroservise; integrisani server za razvoj; podršku za Cloud; prilagođavanje i podrška za *3rd party* biblioteke.

Zašto nam je neophodan Spring Boot?

- Glavna prednost Spring Boot:
  konfiguriše resurse na osnovu onoga što nađe u Classpath.
- Npr. ukoliko vaš Maven POM sadrži JPA zavisnosti i MySQL drajver, tada će Spring Boot podesiti aplikaciju za rad sa MySQL bazom. Ukoliko dodate i veb zavisnosti, onda će podrazumevano biti konfigurisana Spring MVC arhitektura.
- Ukoliko se ništa ne definiše u POM fajlu, Spring Boot će konfigurisati podrazumevano Hibernate kao JPA provajder sa *HyperSQL DataBase*.
- Glavni ciljevi uvođenja Spring Boot:
  - Da se obezbedi brži razvoj Spring aplikacija
  - Da se lako iskoriste podrazumevane konfiguracije parametara aplikacije
  - Da obezbedi veći broj nefunkcionalnih zahteva koji su veoma uobičajeni za projekte velikih razmera (npr. ugrađeni serveri, bezbednost aplikacije, metrika, eksterna konfiguracija, itd.)

Spring Boot pokretanje

- Realizuju se *stand-alone* aplikacije, zasnovane na Spring tehnologiji, koje se lako pokreću
- Pokretanje slično kao kod Javinih aplikacija: java –jar ili preko tradicionalnog WAR
- Postoji i CLI preko koga možemo pokretati „Spring skripte“

Hello World

- Primer aplikacije za Groovy-baziran kontroler/klasu
- Nema konfiguracije
- Nema web.xml

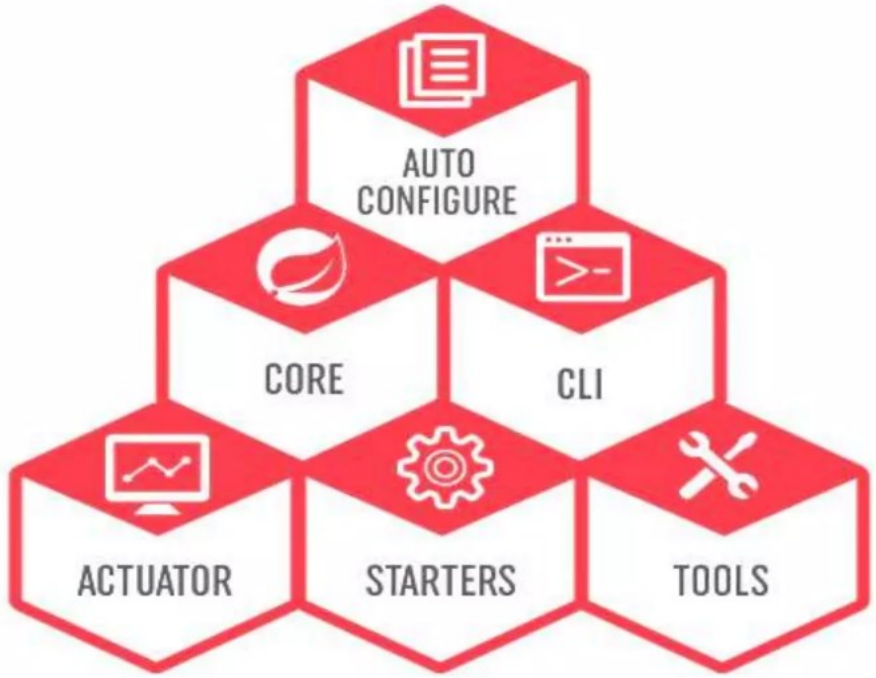
- Pokretanje iz Spring Boot CLI:
  $ spring run HelloController.groovy

- Ovo je samo primer da može i sa Groovy, ali mi ćemo raditi sa programskim jezikom Java.

```
@RestController
class HelloController {

  @RequestMapping("/")
  def hello() {
    return "Hello World"
  }

}
```

Osnovne komponente Spring Boot

- Automatska konfiguracija - automatsko obezbeđivanje konfiguracije
- Jezgro / Početne zavisnosti - kažete koju funkcionalnost želite i osigurano je da će se biblioteke dodati pri izgradnji
- Interfejs komandne linije (CLI) - pisanje kompletne aplikacije samo sa kodom aplikacije (bez tradicionalne izgradnje projekta)
- Aktuatori, starteri, alati



Automatska konfiguracija

- Modul koji radi auto konfiguraciju širokog spektra Spring projekata.
- Detektuje postojanje određenog radnog okvira (eng. framework), kao što su: Spring Batch, Spring Data JPA, Hibernate, JDBC.
- Kada detektuje, pokušaće da autokonfiguriše taj radni okvir sa nekim razumnim podrazumevanim vrednostima, koje se mogu lako zameniti (override) konfiguracijom u datoteci *application.properties/yml*.
- * YAML (Yet Another Markup Language) – *human-readable data serialization language*

  Kao i JSON, ovo je noviji jezik, alternativa XML.

```
- name: create users
  hosts: all
  tasks:
    - user:
      name: "{{ item.name }}"
      state: present
      groups: "{{ item.groups }}"
    with_items:
      - { name: 'linda', groups: 'wheel' }
      - { name: 'lisa', groups: 'root' }
```

Spring Boot Core i CLI

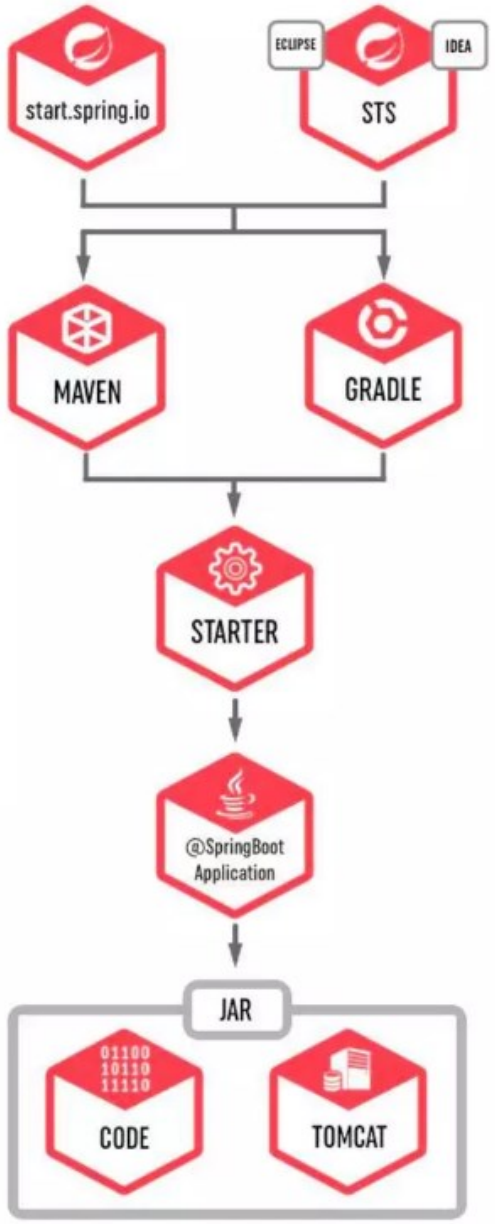
- Sprint Core – baza za druge module, ali pruža neke funkcionalnosti koje mogu da se koriste samostalno, npr. koristeći argumente komandne linije i YAML fajlova, kao svojstva Spring okruženja i automatski vezujući svojstva okruženja za svojstva Spring bean-ova (sa validacijom).
- Spring CLI – interfejs komandne linije za pokretanje ili zaustavljanje kreiranja jedne Spring Boot aplikacije.

Aktuatori, starteri i alati

- Aktuatori - kada se projekat doda, omogući će određene osobine okruženja (bezbednost, metrika, stranica sa podrazumevanim greškama) u vašoj aplikaciji.
- Koristi se da otkrije određene radne okvire/osobine u vašoj aplikaciji.
- Primer: Korišćenjem aktuatora možete pronaći sve REST servise koji su definisani u veb aplikaciji.
- Starteri - različiti kratki startni projekti koji se mogu uključiti kao zavisnost u vaš Maven/Gradle build fajl.
- Alati za izgradnju Maven i Gradle, kao i prilagođeni Spring Boot Loader (korišćen u izvršnom JAR/WAR) su uključeni u projekat.

Kako koristiti Spring Boot?

- Korišćenjem start.spring.io (*Initializr*) ili korišćenjem STS (*Spring Tool Suite*) podrške dostupne u alatima IntelliJ IDEA ili Eclipse ili VS Code, i onda odabrati sve Spring Boot Startere.
- Odabere se da li se koristi Maven ili Gradle kod izgradnje.
- Ukoliko se koristi start.spring.io, tada se preuzme ZIP i konfiguriše radni prostor. Sa druge strane, korišćenjem alata (IDE) će se automatski kreirati zahtevani fajl u radnom prostoru.
- Za kreiranje JAR fajla, može se koristiti mvn clean package ili korisiti IntelliJ IDEA /Eclipse. JAR podrazumeva integrisanje sa Tomcat serverom.



spring initializr

Project

- (●) Gradle - Groovy
- ( ) Gradle - Kotlin
- ( ) Maven

Language

- (●) Java
- ( ) Kotlin
- ( ) Groovy

Spring Boot

- ( ) 3.2.0 (SNAPSHOT)
- ( ) 3.2.0 (RC2)
- ( ) 3.1.6 (SNAPSHOT)
- (●) 3.1.5
- ( ) 3.0.13 (SNAPSHOT)
- ( ) 3.0.12
- ( ) 2.7.18 (SNAPSHOT)
- ( ) 2.7.17

Project Metadata

Group: com.example

Artifact: demo

Name: demo

Dependencies

ADD DEPENDENCIES... CTRL + B

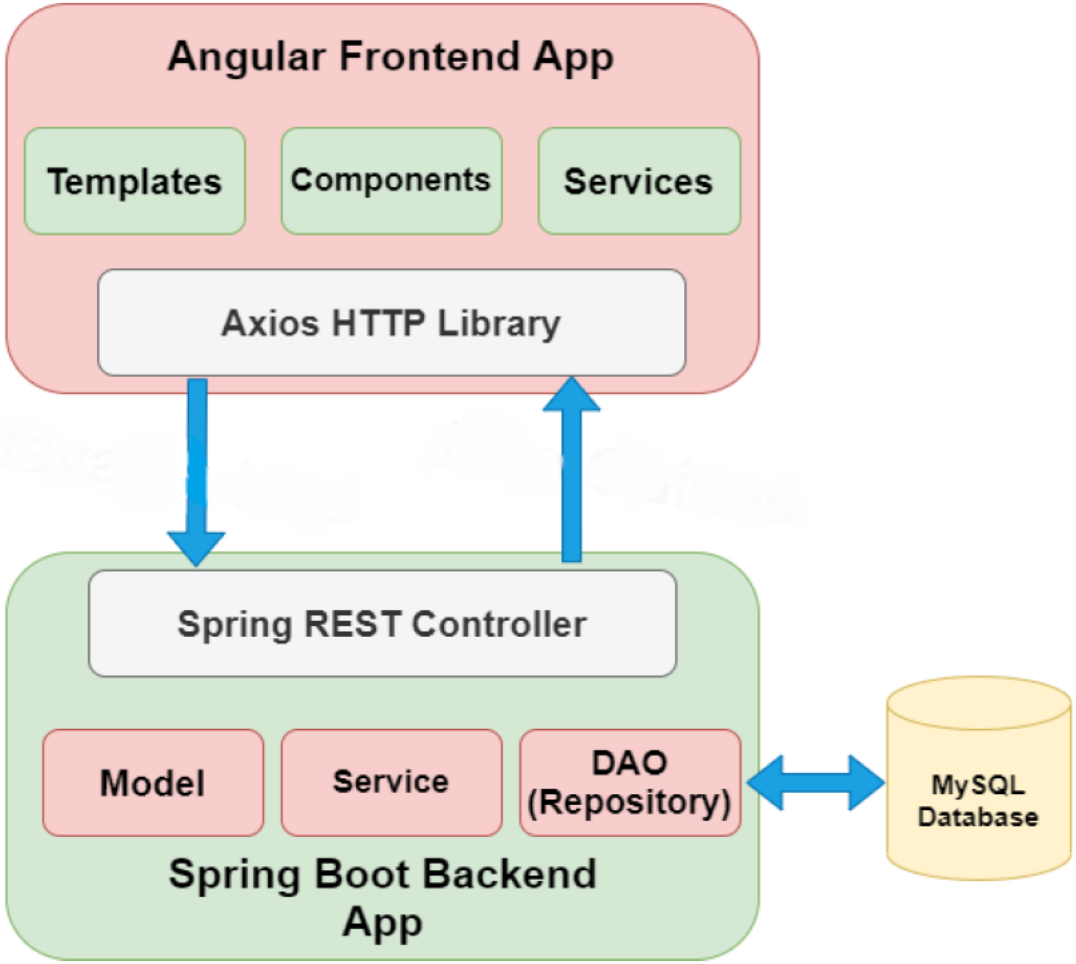
No dependency selected

GENERATE CTRL + ⏎ | EXPLORE CTRL + SPACE | SHARE...

Koraci manuelne inicijalizacije

- Posetiti: https://start.spring.io.
- Servis povlači sve neophodne zavisnosti koje želite i postvlja inicijalni setap.
- Bira se jezik koji želite (Java/Kotlin/Groovy) i
  tip izgradnje aplikacije (Gradle ili Maven).
- U zavisnostima odabrati: Spring Web.
- Pritisnuti Generate
- Preuzeti rezultujući ZIP fajl, koji je generisana veb aplikacija.

Šta mi želimo?
Arhitektura punog steka – Angular + Spring + MySQL



Spring Rest kontroleri

- Prihvataju HTTP zahteve sa klijentske strane (*frontend*) i upućuju ih odgovarajućim servisima.
- Uključuje procesiranje zahteva za podacima, autentifikaciju i autorizaciju korisnika i druge akcije.
- Anotacija: @RestController

Primer kontrolera

```
package com.example.springboot;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

        @GetMapping("/")
        public String index() {
                return "Greetings from Spring Boot!";
        }

}
```

@RestController – za prihvatanje veb zahteva koristiće Spring MVC.

@GetMapping mapira na index() metod.

Kada se pozove iz veb pregledača metoda vraća tekst.

To je zato što @RestController kombinuje @Controller i @ResponseBody, dve anotacije koje kao rezultat vraćaju podatke, a ne prikaz.

src/main/java/com/example/springboot/Application.java

Application class

```
package com.example.springboot;

import java.util.Arrays;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Application {
	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}
	@Bean
	public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
		return args -> {
			System.out.println("Let's inspect the beans provided by Spring Boot:");
			String[] beanNames = ctx.getBeanDefinitionNames();
			Arrays.sort(beanNames);
			for (String beanName : beanNames) {
				System.out.println(beanName);
			}
		};
	}

}
```

Anotacija @SpringBootApplication

- Ona dodaje sledeće:
- @Configuration: Označava klasu kao izvor definicija bean-a za kontekst aplikacije.
- @EnableAutoConfiguration: Govori Spring Boot-u da treba da doda bean-ove zasnovane na podešavanjima classpath-a, drugim bean-ovima ili različitim podešavanjima osobina. Na primer: ako je spring-webmvc na classpath-u, ova anotacija označava da je aplikacija tipa veb aplikcije i aktivira ključna ponašanja, kao što je podešavanje *DispatcherServlet* (frontend kontroler).
- @ComponentScan: Traži Spring-u da pronađe druge komponente, konfiguracije i servise u paketu.
- Metoda main() koristi Spring Boot's SpringApplication.run() metod da pokrene aplikaciju.
- Postoji i metoda *CommandLineRunner* označena kao @Bean, koja se takođe startuje, i preuzima sve bean-ove koje je kreirala aplikacija ili su deo Spring Boot.

Anotacija @RequestMapping

- @RequestMapping – koristi se da mapira HTTP zahtev u metodu MVC/REST kontrolera. Kod Spring MVC aplikacija, *DispatcherServlet* je odgovoran za rutiranje dolazećeg HTTP zahteva u metodu kontrolera.
- Ova anotacija se može primeniti na nivo klase ili nivo metode u kontroleru.
- Na nivou klase, ova anotacija mapira specifičnu putanju zahteva ili obrazac (pattern) na kontroler. Moguće je dodati anotacije na nivou metoda da bi mapiranje označili specifičnijim.
- Primer:

```
// Annotation
@RequestMapping("/hello")
// Method
public String helloWorld() {
   return "Hello World!";
}
```

Ova anotacija veliki broj opcionih elemenata:
consumes, header, method, name, params, path, produces, **value**.

Mapiranja

- **@GetMapping:** Mapira zahteve **HTTP GET** u specifičnu metodu za hvatanje zahteva.
  Koristi se da kreira krajnju tačku veb servisa koji služi za dohvatanje podataka, sa klijentske strane.
  Koristi se umesto: **@RequestMapping(method = RequestMethod.GET)**
- **@PostMapping:** Mapira zahteve **HTTP POST** u specifičnu metodu za hvatanje zahteva.
  Koristi se da kreira krajnju tačku veb servisa koji kreira.
  Koristi se umesto: **@RequestMapping(method = RequestMethod.POST)**
- **@PutMapping:** Mapira zahteve **HTTP PUT** u specifičnu metodu za hvatanje zahteva.
  Koristi se da kreira krajnju tačku veb servisa koji kreira ili ažurira.
  Koristi se umesto: **@RequestMapping(method = RequestMethod.PUT)**
- **@DeleteMapping:** Mapira zahteve **HTTP DELETE** u specifičnu metodu za hvatanje zahteva.
  Koristi se da kreira krajnju tačku veb servisa koji briše resurse.
  Koristi se umesto: **@RequestMapping(method = RequestMethod.DELETE)**
- **@PatchMapping:** Mapira zahteve **HTTP PATCH** u specifičnu metodu za hvatanje zahteva.
  Koristi se umesto: **@RequestMapping(method = RequestMethod.PATCH)**

@RequestBody i @ResponseBody

- Anotacija **@RequestBody** koristi se za vezivane zahteva (HttpRequest) sa objektom prenosa u parametru metode (ili domenskim objektom), omogućavajući automatsku deserijalizaciju dolaznog tela HttpRequest na Java objekat.
- Podrazumevano tip koji je označen anotacijom @RequestBody mora da odgovara JSON objektu, koji šalje naš kontroler na klijentskoj strani.
- Anotacija **@ResponseBody** govori kontroleru da je vraćeni objekat automatski serijalizovan u JSON ili XML format, i vraćen nazad u HttpResponse objekat.

@PathVariable i @RequestParam

- Anotacija **@PathVariable** se koristi za preuzimanje podataka sa URL putanje.
- Definisanjem džoker znakova (placeholder) u mapirajućoj URL adresi zahteva, mogu se povezati ti znakovi sa parametrima metoda anotiranih u @PathVariable.
- Ovo omogućava pristup dinamičkim vrednostima u URL i njihovo korišćenje.
- Primer: /users/123, čime prosleđujemo jedinstveni broj indeksa metodi, koja će preuzeti nakon toga neke podatke o tom korisniku (studentu)
- Anotacija **@RequestParam** dozvoljava da izvučemo podatke iz parametara upita u URL zahtevu. Parametri upita su ključ-vrednost parovi, koji se ugrađuju u URL putanju nakon oznake ?
- Ovo je korisno kada treba proslediti dodatne informacije ili filtere vašim krajnjim tačkama API
- Primer: /users/search?name=Milovan

Korišćenje @PathVariable

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

    @GetMapping("/{userId}")
    public ResponseEntity<User> getUserDetails(@PathVariable Long userId) {
        // Implementation to fetch user details based on the provided userId
        // ...
        return ResponseEntity.ok(user);
    }
}
```

Korišćenje @RequestParam

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

        @GetMapping("/search")
        public ResponseEntity<List<User>> searchUsers(@RequestParam("name")
String name) {
                // Implementation to search users based on the provided name
                // ...
                return ResponseEntity.ok(users);
        }
}
```

Više parametara u @RequestParam

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

        @GetMapping("/search")
        public ResponseEntity<List<User>> searchUsers(
                @RequestParam(value = "name", required = false, defaultValue = "John") String name,
                @RequestParam(value = "age", required = false, defaultValue = "18") int age) {

                // Implementation to search users based on the provided name and age
                // ...
                return ResponseEntity.ok(users);
        }
}
```

Možemo imati podrazumevanu (default) vrednost, ukoliko Query parametar nije definisan u URL putanji.

Још неka важна правила конверзије

- Конверзија типа података: Spring Boot омогућава аутоматску конверзију типа података за @PathVariable и @RequestParam параметре. Улазни подаци захтева могу да се конвертују у потребне типове података (String, int, boolean,...). Уколико конверзија не успе, биће ухваћен изузетак, омогућавајући нам да лако рукујемо грешком.
- Више параметара: Можете користити више парамерата и код @PathVariable и @RequestParam у једној методи која треба да издвоји више вредности из URL путање и параметара упита. Ово омогућава да користимо више података из једног захтева.
- Валидација података: Можете користити валидацију над екстрахованим подацима из захтева коришћењем анотација за валидацију из пакета javax.validation или коришћењем прилагођене логике валидације. Ово помаже да подаци испуњавају одређене критеријуме, пре него што буду даље процесирани.

Још неka важна правила конверзије (2)

- Obrasci (patterns) promenljivih putanja: Anotacija @PathVariable podržava promenljive obrasce unutar URL putanje. Ovo može biti korisnio za rukovanje dinamičkim i složenim URL strukturama.
- Kolekcije parametara upita: Ako očekujete više vrednosti iz upita, možete koristiti tipove List ili Array za odgovarajući parametar metode označen sa @RequestParam. Spring Boot će automatski uvezati sve vrednosti za parametar kolekcije.
- Opcione promenljive putanje: Promenljive putanje možete učiniti opcionim, time što ćete obezbediti podrazumevanu vrednost ili koristiti tip Optional<T> kao parametar metode. Ovo omogućava da rukujete slučajevima u kojima određene promenljive putanje mogu ili ne moraju biti prisutne u zahtevu.
- Rukujte uvek potencijalnim izuzecima i scenarijima sa greškama kada ekstrahujete podatke iz zahteva. Spring Boot pruža nekoliko različitih mehanizama.

@RequestHeader

- Koristi se da bi se dohvatili detalji iz zaglavlja HTTP zahteva.
- Ova anotacija se koristi kao parametar metode.
- Opcioni elementi ove anotacije su: **name**, **required**, **value**, **defaultValue**
- Za svaki detalj u zaglavlju, treba da se koristi odvojena anotacija (dozvoljeno je korišćenje anotacije više puta u metodi).
- Primer sa string tipom u zaglavlju i definisanim imenom:

```
@GetMapping("/greeting")
public ResponseEntity<String> greeting(@RequestHeader(HttpHeaders.ACCEPT_LANGUAGE)
String language) {
    // code that uses the language variable
    return new ResponseEntity<String>(greeting, HttpStatus.OK);
}
```

@RequestHeader

- Primer sa celim brojem:

```
@GetMapping("/double")
public ResponseEntity<String> doubleNumber(@RequestHeader("my-number") int myNumber) {
    return new ResponseEntity<String>(String.format("%d * 2 = %d",
     myNumber, (myNumber * 2)), HttpStatus.OK);
}
```

- Ako nismo sigurni da je zaglavlje prisutno ili nam je potrebno više njih nego što želimo u potpisu naše metode, tada možemo koristiti istu anotaciju, ali bez određenog imena.
- Koju promenljivu onda koristiti?
- Ima više načina: Map, MultiValueMap, HttpHeaders objekat

@RequestHeader

Primeri:

```
@GetMapping("/listHeaders")
public ResponseEntity<String> listAllHeaders(
 @RequestHeader Map<String, String> headers) {
  headers.forEach((key, value) -> {
     LOG.info(String.format("Header '%s' = %s", key, value));
  });

  return new ResponseEntity<String>(
   String.format("Listed %d headers", headers.size()),
HttpStatus.OK);
}
```

```
@GetMapping("/multiValue")
public ResponseEntity<String> multiValue(
 @RequestHeader MultiValueMap<String, String> headers) {
  headers.forEach((key, value) -> {
     LOG.info(String.format(
      "Header '%s' = %s", key,
       value.stream().collect(Collectors.joining("|"))));
  });

  return new ResponseEntity<String>(
   String.format("Listed %d headers", headers.size()),
HttpStatus.OK);
}
```

CORS (*Cross-Origin Resource Sharing*) konfiguracija

- RESTful veb servis će uključiti CORS kontrolu pristupa zaglavlja u odgovoru, ukoliko dodate anotaciju @CrossOrigin:

```
@CrossOrigin(origins = "http://localhost:8080")
@GetMapping("/greeting")
public Greeting greeting(@RequestParam(required = false, defaultValue = "World") String name)
{
        System.out.println("==== get greeting ====");
        return new Greeting(counter.incrementAndGet(), String.format(template, name));
```

- Anotacija @CrossOrigin dozvoljava deljenje resursa sa više izvora samo za ovu označenu metodu. Podrazumevano, dozvoljeni su svi izvori, sva zaglavlja i HTTP metode naznačene u @RequestMapping anotaciji.

@Required

- Primenjuje se na setter metodu u bean-u.
- Ukazuje da anotirani bean mora biti popunjen u vreme konfigurisanja potrebnim svojstvom, u suprotnom će se baciti izuzetak *BeanInitilizationException.*
- *Primer:*

```
public class Machine
{
        private Integer cost;
        @Required
        public void setCost(Integer cost)   {
                this.cost = cost;
        }
        public Integer getCost() {
                return cost;
        }
}
```

@Autowired

- Anotacija koja služi za injektiranja drugih bean-ova sa kojima sarađujemo u naš bean.
- Nakon što omogućimo injektiranje, možemo koristiti automatsko uvezivanje na svojstvima (properties), setter metodama i konstruktorima.
- *Primer*:

```
@Component
public class Customer  {
        private Person person;
        @Autowired
        public Customer(Person person) {
                this.person=person;
        }
}
```

@Configuration

- Anotacija klasnog nivoa. Klasa koja ima ovu oznaku koristi se od strane Spring kontejnera kao izvor definisanja bean-a.
- *Primer*:

```
@Configuration
public class Vehicle  {
    @Bean
    Vehicle engine()  {
            return new Vehicle();
    }
}
```

@ComponentScan

- Anotacija kada u određenom bean-u želimo da skeniramo paket.
- Koristi se zajedno sa anotacijom @Configuration.
- @ComponentScan osigurava da se sve klase označene sa @Component, kao i njihovi derivati (uključujući @Repository) pronađu i registruju kao Spring bean-ovi.
- @ComponentScan je automatski uključen u @SpringBootApplication.
- *Primer*:

```
@ComponentScan(basePackages = "rs.ac.bg.etf")
@Configuration
public class ScanComponent
{
    // ...
}
```

@Bean

- Anotacija na nivou metoda. Predstavlja alternativu za <bean> oznaku u XML fajlu.
- Ovaj element govori metodi da će njom upravljati Spring Container.
- *Primer*:

```
@Bean
public BeanExample beanExampleMethod()
{
    return new BeanExample ();
}
```

@Component

- Za razliku od prethodnih anotacija, koji su pripadali Core Spring anotacijama, u nastavu navodimo stereotipove koji grade Spring aplikaciju.
- @Component je anotacija na nivou klase. Koristi se za označavanje klase kao beana.
- Javina klasa označena sa @Component se pronalazi u classpath-u.
- Radni okvir preuzima i konfiguriše ga u kontekstu aplikacije kao Spring Bean.
- *Primer*:

```
@Component
public class Student
{
      .......
}
```

@Controller

- Anotacija na nivou klase. Označava klasu kao hvatač za veb zahtev (request) i često se koristi da prikaže veb stranice.
- Podrazumevano vraća string, koji označava rutu na koju se redirektujemo.
- Kombinuje se sa anotacijom @RequestMapping.
- *Primer*:

```
@Controller
@RequestMapping("books")
public class BooksController  {
    @RequestMapping(value = "/{name}", method = RequestMethod.GET)
    public Employee getBooksByName() {
        return booksTemplate;
    }
}
```

Razlike @Controller i @RestController

- U Spring MVC, obe anotacije se koriste da definišu veb kontrolere u MVC uzorku. Kontroler je odgovoran za prihvatanje HTTP zahteva i vraćanje HTTP odgovora klijentu.
- Kod Spring Boot, @Controller se koristi da kreira veb kontroler koji vraća poglede (*views*) u vidu HTTP odgovora (*response*),
  dok se @RestController koristi da kreira veb servise (REST API) koji vraćaju JSON ili XML podatke.
- @RestController je kombinacija @Controller i @ReponseBody sa ciljem da napravimo REST API u Spring Boot (uveden je od Spring 3.4 verzije).

Servisi @Service

- Sadrže poslovnu logiku aplikacije i koriste anotaciju @Service.
- Koristi se na klasnom nivou i definiše klasu sa poslovnom logikom.
- Ovde se obavlja obrada podataka, validacija, nekad i komunikacija sa bazom podataka.

```
package rs.ac.bg.etf;
@Service
public class TestService  {
    public void serviceMethod()  {
    //poslovna logika
    }
}
```

Repozitorijumi @Repository

- Repozitorijumi se koriste za interakciju sa bazom podataka.
- Anotacija @Repository prikazuje da je klasa tipa repozitorijuma.
- Ovaj mehanizam služi za enkapsuliranje skladišta, preuzimanje i pretraživanje objekata, kroz kolekcije objekata, odnosno repozitorijumi omogućavaju dohvatanje, ažuriranje i brisanje podataka u bazi.
- Predstavlja specijalizaciju anotacije @Component, koja omogućava da se klase automatski detektuju kroz skeniranje classpath.

```
package rs.ac.bg.etf;
@Repository
public class TestRepository  {
    public void delete()  {
    //persistence code
    }
}
```

Repozitorijumi u strukturi projekta

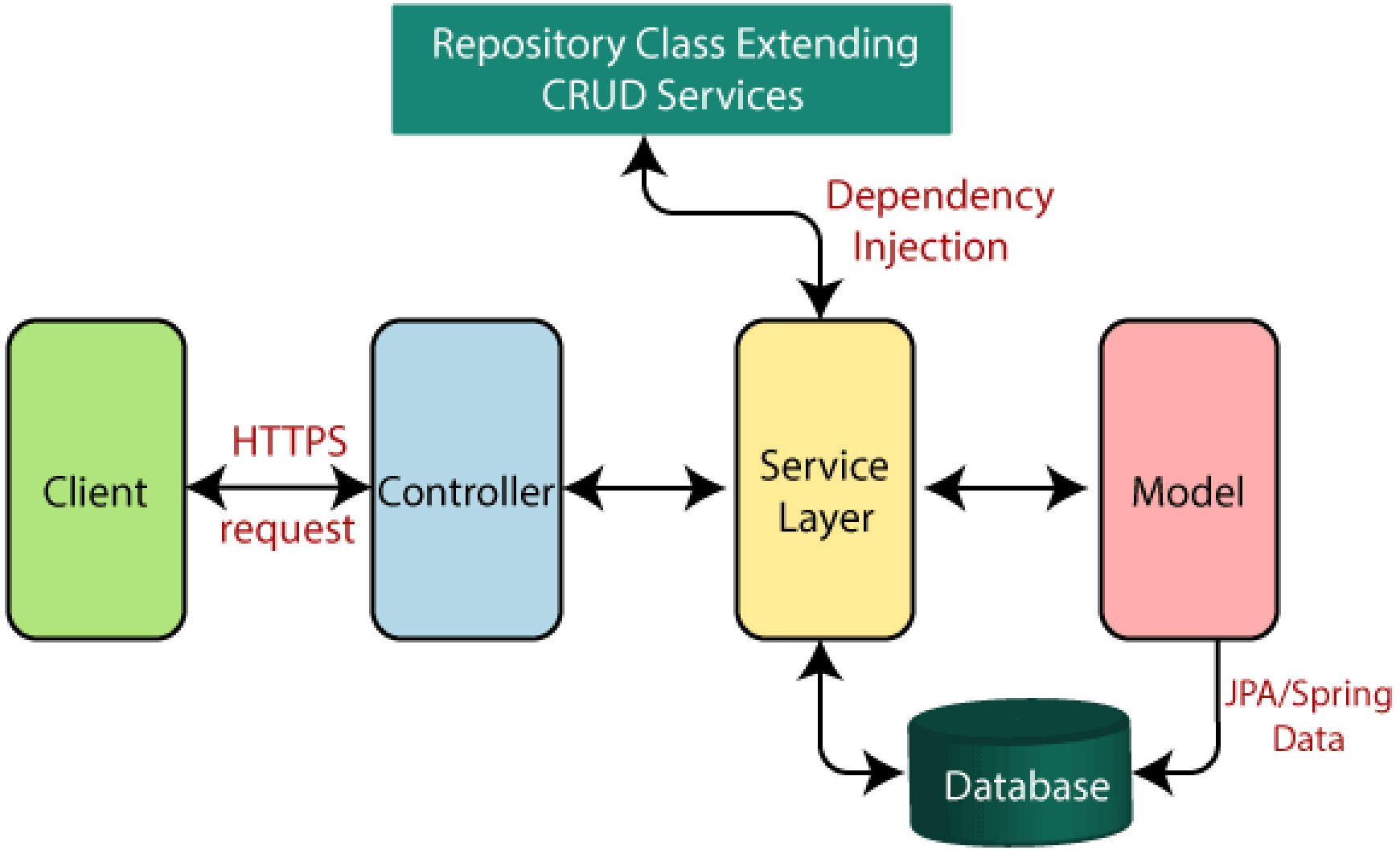
```
build.gradle
...
src
├── main
│   ├── java
│   │   └── com
│   │       └── zetcode
│   │           ├── Application.java
│   │           ├── controller
│   │           │   └── MyController.java
│   │           ├── model
│   │           │   └── Country.java
│   │           ├── repository
│   │           │   └── CountryRepository.java
│   │           └── service
│   │               ├── CountryService.java
│   │               └── ICountryService.java
│   └── resources
│       ├── application.yml
│       ├── import.sql
│       ├── static
│       │   └── index.html
│       └── templates
│           └── showCountries.ftlh
└── test
    ├── java
    └── resources
```

@Query

- Koriste anotaciju za upite @Query:

```
@Query(value = "SELECT b FROM Diplomski b WHERE b.idDiplomski=?1")
Optional<Diplomski> findDiplomskiById(Integer id);

@Transactional
@Modifying
@Query(value = "DELETE FROM Diplomski b WHERE b.idDiplomski=?1")
void deleteDiplomskiById(Integer id);
```

Spring Boot arhitektura sa bazom



Maven - alat za upravljanje zavisnostima

- Prednosti alata za upravljanje zavisnostima:
  - Pružaju centralizovane informacije o zavisnostima, uz definisanje verzije Spring Boot na jednom mestu. Veoma je korisno ukoliko menjamo verzije radnog okvira.
  - Izbegava nepodudaranje različitih verzija Spring Boot biblioteka.
  - Potrebno je samo da napišemo ime biblioteke, sa specificiranjem verzije (korisno za projekte sa više modula).
- Maven projekat nasleđuje osobine iz početnog projekta ***spring-boot-starter-parent***:
  - Podrazumevanu Java kompajler verziju
  - UTF-8 enkodovanje
  - Odeljak sa zavisnostima iz spring-boot-dependency-pom
  - Zavisnosti nasleđene iz POM fajla
  - Resource filtering + Plugin configuration

spring-boot-starter-parent (pom.xml)

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.3</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

- Verzija Jave (ako želimo da menjamo):

```
<properties>
        <java.version>1.8</java.version>
</properties>
```

spring-boot-starter-parent (pom.xml)

- Dodavanje Maven dodatka i pakovanje u JAR fajl:

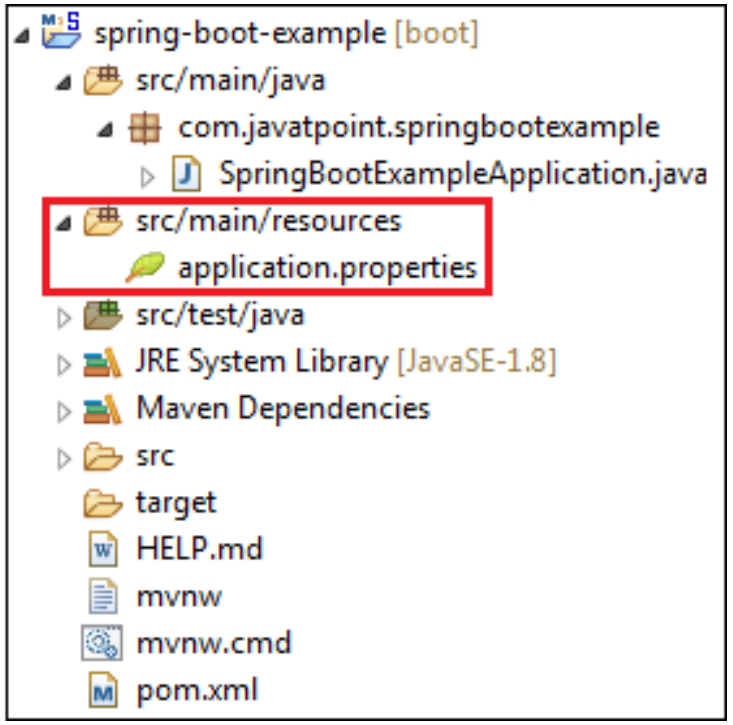
```
<build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
</build>
```

Spring Boot bez Parent POM

```
<dependencyManagement>
    <dependencies>
        <dependency><!-- Import dependency management from Spring Boot -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.2.2.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Spring Boot Application Properties

- Radni okvir Spring Boot dolazi sa mehanizmom izgradnje za konfigurisanje aplikacije korišćenjem fajla **application.properties** (lokacija: **src/main/resources**)
- Spring Boot pruža nekoliko osobina koje se mogu konfigurisati i neke od njih imaju podrazumevane (default) vrednosti.
- Spring Boot dozvoljava definisanje nekog posebnog parametra koji želite, ako je potrebno.
- Korišćenjem ovog fajla:
  - konfiguriše se Spring Boot radni okvir
  - našoj aplikaciji definišemo specifične osobine



Primer applicaton.properties fajla

```
spring.datasource.url=jdbc:mysql://localhost:3306/rti_katedra
spring.datasource.username=root
spring.datasource.password=sifra123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
server.error.include-stacktrace=always
spring.main.allow-circular-references=true

spring.servlet.multipart.max-file-size=2MB
spring.servlet.multipart.max-request-size=2MB
```

Kategorije Spring Boot osobina (property)

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties
15. DevTools Properties
16. Testing Properties

Osobine aplikacije (1)

Property	Default Values	Description
Debug	false	It enables debug logs.
spring.application.name		It is used to set the application name.
spring.application.admin.enabled	false	It is used to enable admin features of the application.
spring.config.name	application	It is used to set config file name.
spring.config.location		It is used to config the file name.
server.port	8080	Configures the HTTP server port
server.servlet.context-path		It configures the context path of the application.
logging.file.path		It configures the location of the log file.
spring.banner.charset	UTF-8	Banner file encoding.
spring.banner.location	classpath:banner.txt	It is used to set banner file location.
logging.file		It is used to set log file name. For example, data.log.

Osobine aplikacije (2)

Property	Default Values	Description
spring.application.index		It is used to set application index.
spring.application.name		It is used to set the application name.
spring.application.admin.enabled	false	It is used to enable admin features for the application.
spring.config.location		It is used to config the file locations.
spring.config.name	application	It is used to set config the file name.
spring.mail.default-encoding	UTF-8	It is used to set default MimeMessage encoding.
spring.mail.host		It is used to set SMTP server host. For example, smtp.example.com.
spring.mail.password		It is used to set login password of the SMTP server.
spring.mail.port		It is used to set SMTP server port.
spring.mail.test-connection	false	It is used to test that the mail server is available on startup.
spring.mail.username		It is used to set login user of the SMTP server.

Osobine aplikacije (3)

Property	Default Values	Description
spring.main.sources		It is used to set sources for the application.
server.address		It is used to set network address to which the server should bind to.
server.connection-timeout		It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection.
server.context-path		It is used to set context path of the application.
server.port	8080	It is used to set HTTP port.
server.server-header		It is used for the Server response header (no header is sent if empty)
server.servlet-path	/	It is used to set path of the main dispatcher servlet
server.ssl.enabled		It is used to enable SSL support.
spring.http.multipart.enabled	True	It is used to enable support of multi-part uploads.
spring.servlet.multipart.max-file-size	1MB	It is used to set max file size.
spring.mvc.async.request-timeout		It is used to set time in milliseconds.

Osobine aplikacije (4)

Property	Default Values	Description
spring.mvc.date-format		It is used to set date format. For example, dd/MM/yyyy.
spring.mvc.locale		It is used to set locale for the application.
spring.social.facebook.app-id		It is used to set application's Facebook App ID.
spring.social.linkedin.app-id		It is used to set application's LinkedIn App ID.
spring.social.twitter.app-id		It is used to set application's Twitter App ID.
security.basic.authorize-mode	role	It is used to set security authorize mode to apply.
security.basic.enabled	true	It is used to enable basic authentication.
Spring.test.database.replace	any	Type of existing DataSource to replace.
Spring.test.mockmvc.print	default	MVC Print option
spring.freemaker.content-type	text/html	Content Type value
server.server-header		Value to use for the server response header.
spring.security.filter.dispatcher-type	async, error, request	Security filter chain dispatcher types.
spring.security.oauth2.client.registration.*		OAuth client registrations.

Starteri

- Spring Boot pruža veći broj startera (početnih podešavanja), koji omogućavaju lakši i brži razvoj. Spring Boot Starteri su deskriptori zavisnosti.
- U ovom radnom okviru, svi starteri imaju slično imenovanje: **spring-boot-starter-*** (gde * označava određeni tip aplikacije)
- Na primer, ako želimo da koristimo Spring i JPA za pristup bazama podataka, mi ćemo uključiti zavisnost **spring-boot-starter-data-jpa** u naš **pom.xml** fajl.
- Takođe, mogu biti uključeni starteri (pokretači) sa neke treće strane.
- Starter treće strane počinje imenom projekta. Na primer ako je projekat treće strane imenovan sa etfbgd, onda će zavinost biti: etfbgd-spring-boot-starter

Spring Boot starteri

Naziv	Opis
spring-boot-starter-thymeleaf	It is used to build MVC web applications using Thymeleaf views.
spring-boot-starter-data-couchbase	It is used for the Couchbase document-oriented database and Spring Data Couchbase.
spring-boot-starter-artemis	It is used for JMS messaging using Apache Artemis.
spring-boot-starter-web-services	It is used for Spring Web Services.
spring-boot-starter-mail	It is used to support Java Mail and Spring Framework's email sending.
spring-boot-starter-data-redis	It is used for Redis key-value data store with Spring Data Redis and the Jedis client.
spring-boot-starter-web	It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container.
spring-boot-starter-data-gemfire	It is used to GemFire distributed data store and Spring Data GemFire.
spring-boot-starter-activemq	It is used in JMS messaging using Apache ActiveMQ.
spring-boot-starter-data-elasticsearch	It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch.
spring-boot-starter-integration	It is used for Spring Integration.
spring-boot-starter-test	It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito.

Spring Boot starteri (2)

Naziv	Opis
spring-boot-starter-jdbc	It is used for JDBC with the Tomcat JDBC connection pool.
spring-boot-starter-mobile	It is used for building web applications using Spring Mobile.
spring-boot-starter-validation	It is used for Java Bean Validation with Hibernate Validator.
spring-boot-starter-hateoas	It is used to build a hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS.
spring-boot-starter-jersey	It is used to build RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web.
spring-boot-starter-data-neo4j	It is used for the Neo4j graph database and Spring Data Neo4j.
spring-boot-starter-data-ldap	It is used for Spring Data LDAP.
spring-boot-starter-websocket	It is used for building the WebSocket applications. It uses Spring Framework's WebSocket support.
spring-boot-starter-aop	It is used for aspect-oriented programming with Spring AOP and AspectJ.
spring-boot-starter-amqp	It is used for Spring AMQP and Rabbit MQ.
spring-boot-starter-data-cassandra	It is used for Cassandra distributed database and Spring Data Cassandra.

Spring Boot starteri (3)

Naziv	Opis
spring-boot-starter-social-facebook	It is used for Spring Social Facebook.
spring-boot-starter-jta-atomikos	It is used for JTA transactions using Atomikos.
spring-boot-starter-security	It is used for Spring Security.
spring-boot-starter-mustache	It is used for building MVC web applications using Mustache views.
spring-boot-starter-data-jpa	It is used for Spring Data JPA with Hibernate.
spring-boot-starter	It is used for core starter, including auto-configuration support, logging, and YAML.
spring-boot-starter-groovy-templates	It is used for building MVC web applications using Groovy Template views.
spring-boot-starter-freemarker	It is used for building MVC web applications using FreeMarker views.
spring-boot-starter-batch	It is used for Spring Batch.
spring-boot-starter-social-linkedin	It is used for Spring Social LinkedIn.
spring-boot-starter-cache	It is used for Spring Framework's caching support.
spring-boot-starter-data-solr	It is used for the Apache Solr search platform with Spring Data Solr.
spring-boot-starter-data-mongodb	It is used for MongoDB document-oriented database and Spring Data MongoDB.

Spring Boot starteri (4)

Naziv	Opis
spring-boot-starter-jooq	It is used for jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc.
spring-boot-starter-jta-narayana	It is used for Spring Boot Narayana JTA Starter.
spring-boot-starter-cloud-connectors	It is used for Spring Cloud Connectors that simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku.
spring-boot-starter-jta-bitronix	It is used for JTA transactions using Bitronix.
spring-boot-starter-social-twitter	It is used for Spring Social Twitter.
spring-boot-starter-data-rest	It is used for exposing Spring Data repositories over REST using Spring Data REST.

Produkcioni i tehnički starteri

Naziv	Opis
spring-boot-starter-actuator	It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application.
spring-boot-starter-remote-shell	It is used for the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5.

Naziv	Opis
spring-boot-starter-undertow	It is used for Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat.
spring-boot-starter-jetty	It is used for Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat.
spring-boot-starter-logging	It is used for logging using Logback. Default logging starter.
spring-boot-starter-tomcat	It is used for Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web.
spring-boot-starter-log4j2	It is used for Log4j2 for logging. An alternative to spring-boot-starter-logging.

Spring Boot Starter za veb razvoj

- Spring veb koristi Spring MVC, REST i Tomcat, kao podrazumevani ugrađeni veb server. Ubacivanjem spring-boot-starter-web zavisnosti, tranzitivno se povlače sve zavisnosti neophodne za razvoj veb aplikacija:
  - org.springframework.boot:spring-boot-starter
  - org.springframework.boot:spring-boot-starter-tomcat
  - org.springframework.boot:spring-boot-starter-validation
  - com.fasterxml.jackson.core:jackson-databind
  - org.springframework:spring-web
  - org.springframework:spring-webmvc
- Primer:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

Automatska konfiguracija

- *Spring-boot-starter-web* konfiguriše sledeće stvari:
  - Dispatcher Servlet
  - Stranicu sa greškom (*error page*)
  - Web JARs za upravljanje statičkim zavisnostima
  - Ugrađeni servlet kontejner
- Svaka Spring Boot aplikacija ima ugrađen server za podizanje aplikacije.
- Podrazumevani ugrađeni server je **Tomcat**. Postoje podrška za još 2 servera: Jetty Server i Undertow Server.
- Kada se koristi drugi server, Tomcat mora da se isključi, da ne bi došlo do konflikta između dva veb servera.

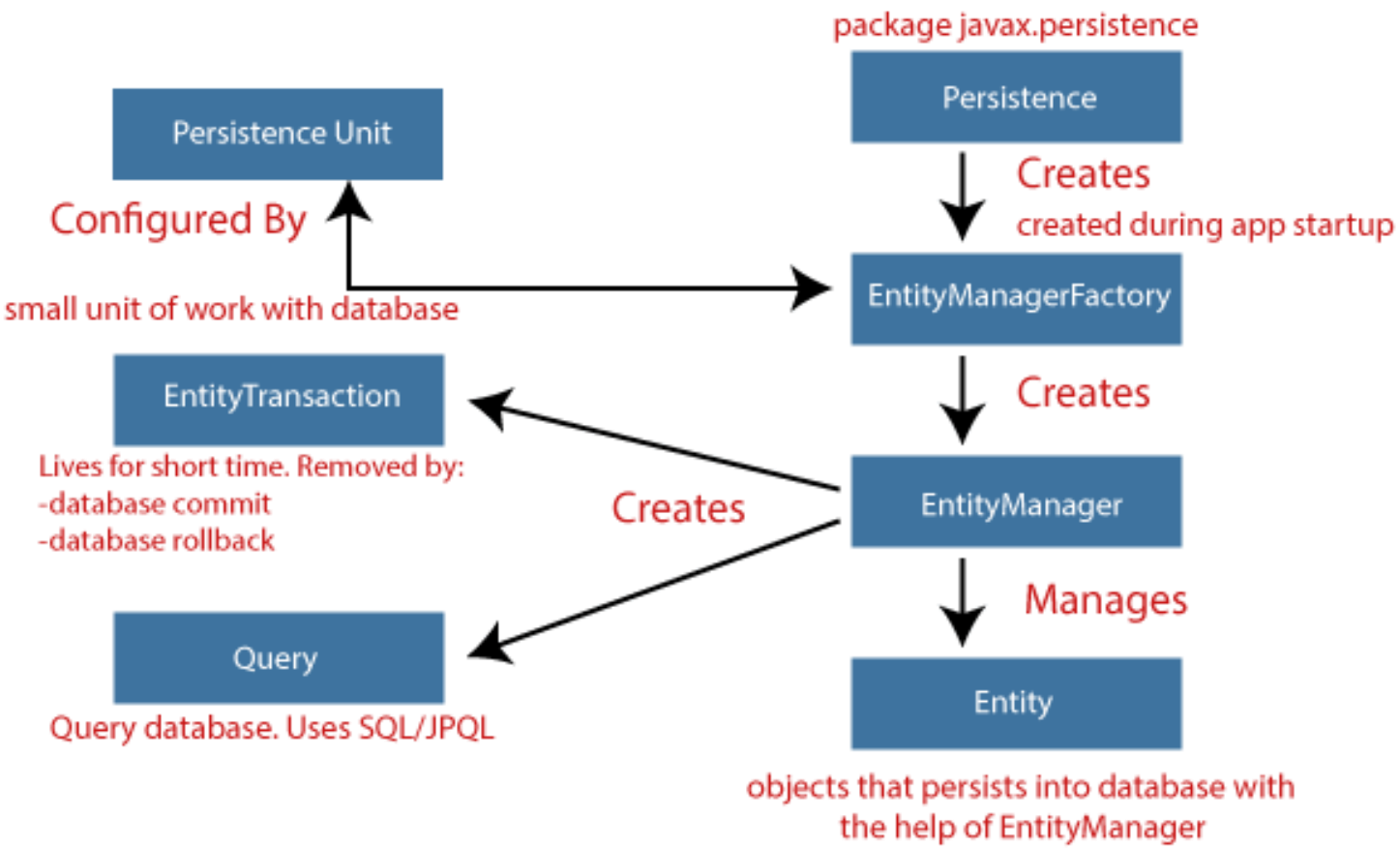
Zamena servera u podešavanjima

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
                <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

JPA – Java Persistence API

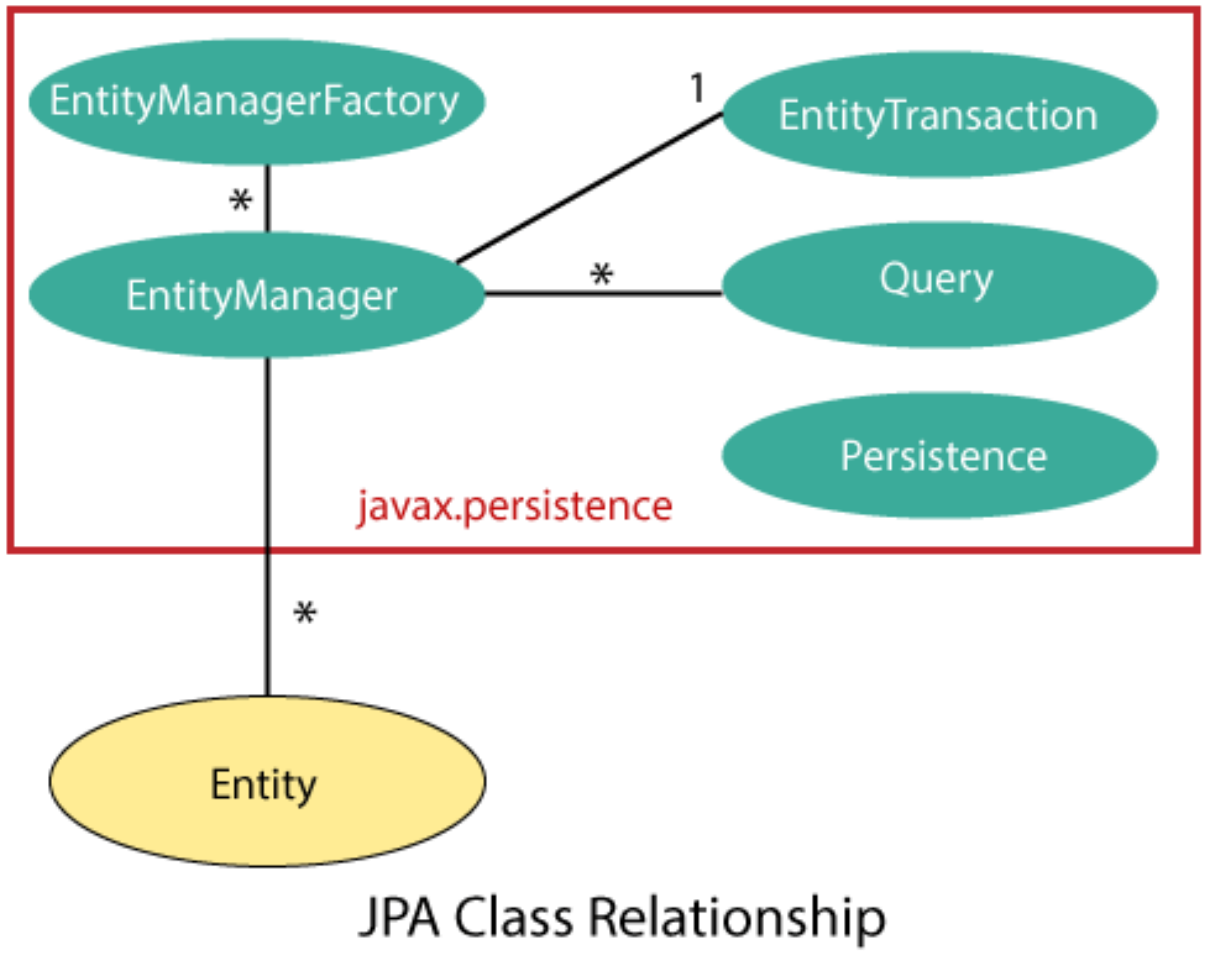
- Za komunikaciju sa relacionom MySQL bazom možemo koristiti *Java Persistence API* (JPA), koji omogućava mapirane Java objekata na entitete u bazi, čime se olakšava pristup i manipulacija podacima.
- Sve operacije sa bazom podataka, kao što su čitanje, ažurirane, brisanje i dodavanje obavljaće se preko JPA.
- Koristi se: platformski nezavistan, objektno orijentisani upitni jezik JPQL (*Java Persistent Query Language*).
- Prednosti JPA:
  - JPA izbegava pisanje DDL zasnovan na SQL dijalektu, umesto toga mapira u XML ili koristi Java anotacije. Takođe, JPA izbegava pisanje DML zasnovanom na specifičnom dijalektu SQL.
  - JPA omogućava da čuvamo i učitavamo Java objekte i grafove bez ikakvog DML jezika.
  - Kada izvršavamo JPQL upite, on omogućava da izrazimo upite u obliku Java entiteta, a ne kroz prirodne SQL tabele i kolone.

Arhitektura Java Persistence API



- Persistence - klasa koja sadrži statičke metode za dobijanje EntityManagerFactory instance.
- EntityManagerFactory – klasa fabrika EntityManager, koja kreira i upravlja sa više instanci EntityManager.
- EntityManager – interfejs koji kontroliše operacije perzistencije nad objektima. Radi za Query instancu.
- Entity – to su perzistencioni objekti koji čuvaju zapis u bazi podataka.
- Persistence Unit – definiše skup svih entitetskih klasa. U aplikaciji, EntityManager instance upravljaju tim.
- EntityTransaction – ima 1-1 vezu za EntityManager.
- Query – interfejs koji implementira svaki JPA potrošač.

Relacije između JPA klasa i interfejsa



JPA Class Relationship

- Relacija između EntityManager i EntityTransaction je 1-1. Postoji po jedna instanca EntityTransaction za svaku EntityManager operaciju.
- Relacija između EntityManagerFactory i EntityManager je 1-prema-više. Postoji jedna klasa fabrika za intance EntityManager.
- Veza između EntityMaanger i Query je 1-prema-više. Može se izvršiti više upita nad jednom instancom EntityManager klase.
- Veza između EntityManager i Entity je 1-prema-više. Instanca EntityManager može upravljati sa više entiteta.

Spring Data JPA

- Spring Data je izvorni Spring projekat visokog nivoa.
- Svrha: objediniti i omogućiti jednostavan pristup različitim vrstama skladišta, i relacionim bazama i NoSQL skladištima podataka, kroz DAL.
- Cilj: Implementacijom nove aplikacije, treba da se fokusiramo na poslovnu logiku, ne na tehničku složenost i šablonski programski kod.
- Spring Data JPA dodaje sloj na vrh JPA i on koristi sve osobine definisane kroz JPA specifikaciju: entitet, mapiranje asocijacija, mogućnosti izgadnje upita kroz JPA.
- Spring Data JPA dodaje svoje osobine kao što su implementacija šablona repozitorijuma (bez koda) i kreiranje upita baze podataka iz naziva metode.
- Spring Data JPA obrađuje većinu kompleksnog pristupa bazama kroz JDBC i ORM (objektno relaciono mapiranje).

Spring Data JPA - osobine

- Tri glavne prednosti su:
  - **Repozitorijum bez koda -** Najpopularniji projektni obrazac za perzistenciju**.** Omogućava nam da implementiramo naš kod na višem nivou apstrakcije**.**
  - **Redukovanje šablonskog koda -** Obezbeđuje podrazumevanu implementaciju za svaki metod preko interfejsa repozitorijuma**.** To znači da nemamo potrebu implementirati operacije čitanja i pisanja**.**
  - **Generisani upiti –** Generisanje upita za bazu na osnovu imena metoda**.** Ako upit nije previše kompleksan, treba da definišemo metod na interfejsu našeg repozitorijuma sa imenom koji počinje sa **findBy.** Nakon definisanja metoda, Spring parsira naziv metode i kreira upit:

```
public interface EmployeeRepository extends CrudRepository<Employee, Long>
{
    Employee findByName(String name);
}
```

JPQL – Java Persistence Query Language

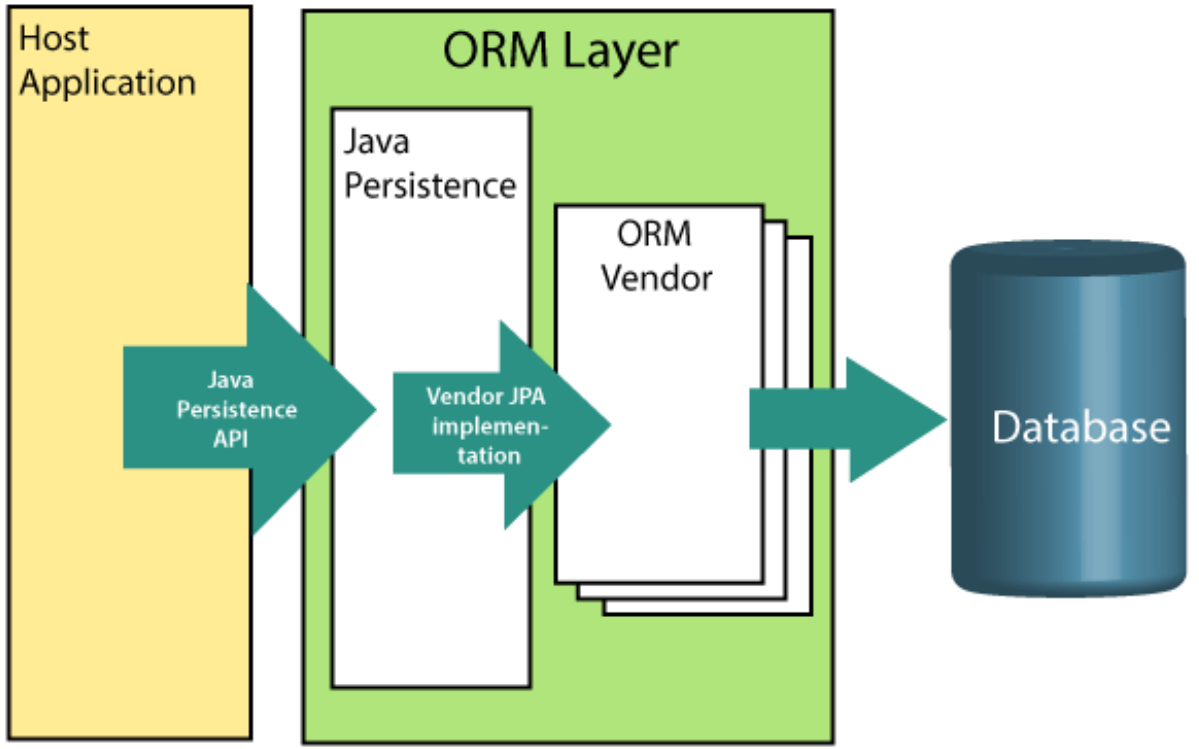
- Spring generiše JPQL upite, zasnovane na imenu metode.
- Upit se izvodi iz potpisa metode. On postavlja vrednost parametra, izvršava upit i vraća rezultat.
- Postoje još neke karakteristike ovog jezika kao što su:
  - Može da integriše poseban kod za repozitorijum.
  - Podržava transparentnu reviziju i apstrakciju objektno-relacionog mapiranja.
  - Implementira osnovnu domensku klasu koja obezbeđuje osnovna svojstva.
  - Podržava nekoliko modula, kao što su: Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra, itd.

Spring Data JPA repozitorijumi

- *CrudRepository* – nudi standarde operacije *Create*, *Read*, *Update*, *Delete*. Sadrži metode kao što su: findOne( ), findAll( ), save( ), delete( ), itd.
- *PagingAndSortingRepository* – proširuje CrudRepozitory i dodaje findAll( ) metode, uz omogućavanje sortiranja i preuzimanja podataka koje ćemo izdeliti (paginacija).
- *JpaRepository* - JPA specifičan repozitorijum, koji proširuje oba repozitorijuma (*CrudRepository* i *PagingAndSortingRepository*) i dodaje specifične metode, kao što je *flush( )* da bi se pokrenula operacija flush za perzistiranje konteksta.
- Hibernate je implementacija JPA, i predstavlja jedan od najpopularnijih ORM radnih okvira. JPA je samo API koji definiše specifikaciju.
- Uz pomoć Hibernate vezujemo objekte sa tabelama. On osigurava da se podaci čitaju iz baze/čuvaju u bazi, na osnovu mapiranja. On pruža takođe dodatne funkcije na vrhu JPA.
- Postoje i druge implementacije JPA, kao što su EclipseLink, DataNucleus, itd.

ORM

- ORM – mapiranje Javinih objekata u tabele baze podataka.
- ORM mapiranje radi kao most između relacionih baza podataka (tabele i zapisi) i Java aplikacija (klase i objekti).
- ORM sloj je sloj za prilagođavanje koji prilagođava jezik objektnih grafova u strukturalni jezik SQL.



Spring Boot aktuatori (pokretači)

- Spring Boot aktuatori su potprojekti koji uključuju određeni broj dodatnih funkcija koji nam pomažu da nadgledamo i upravljamo Spring Boot aplikacijom.
- Aktuatori sadrže krajnje tačke aktuatora (mesta gde resursi žive).
- Možemo koristiti HTTP i JMX krajnje tačke za upravljanje i nadgledane Spring Boot aplikacije.
- Ako želimo da dobijemo funkcionalnosti spremne za produkcionu aplikaciju, tada koristimo Spring Boot aktuatore.
- Tri glavne karakteristike aktuatora su:
  - krajnje tačke (eng. *endpoints*),
  - metrike (eng. *metrics*),
  - revizije (eng. *audit*).

Karakteristike aktuatora

- Krajnje tačke aktuatora omogućavaju nadgledanje i interakciju sa aplikacijom.
- Spring Boot pruža brojne ugrađene krajnje tačke, ali možemo dodati i sopstvene krajnje tačke. Mi utičemo na uključivanje ili isključivanje krajnje tačke individualno.
- Većina aplikacija bira HTTP, gde je ID krajnje tačke zajedno sa prefiksom /actuator, mapiran u URL adresi. Na primer: krajnja tačka /health pruža osnovne informacije o stanju aplikacije, pa mu se pristupa preko: /actuator/health
- Spring Boot Aktuator obezbeđuje dimenzionalne metrike, uz pomoć Mikrometra.
- Mikrometar, integrisan u Spring Boot, je biblioteka za instrumentaciju, koja pokreće isporuku aplikativnih metrika iz Springa.
- Mikrometar obezbeđuje interfejse, neutralne za proizvođača, kao što su: timers, gauges, counters, long task timers, itd.
- Spring Boot pruža fleksibilan okvir revizije (događaji se objavljuju u *AuditEventRepository*). Događaji autentifikacije se automatski objavljuju.

Omogućavanje Spring Boot aktuatora

- Injektiranjem zavisnosti spring-boot-starter-acutator, u pom.xml fajlu:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

Spring Boot aktuator - krajnje tačke

ID	Korišćenje	Default
actuator	It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath.	True
auditevents	It exposes audit events information for the current application.	True
autoconfig	It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied.	True
beans	It is used to display a complete list of all the Spring beans in your application.	True
configprops	It is used to display a collated list of all @ConfigurationProperties.	True
dump	It is used to perform a thread dump.	True
env	It is used to expose properties from Spring's ConfigurableEnvironment.	True
flyway	It is used to show any Flyway database migrations that have been applied.	True
health	It is used to show application health information.	False
info	It is used to display arbitrary application info.	False

Spring Boot aktuator - krajnje tačke (2)

ID	Korišćenje	Default
loggers	It is used to show and modify the configuration of loggers in the application.	True
liquibase	It is used to show any Liquibase database migrations that have been applied.	True
metrics	It is used to show metrics information for the current application.	True
mappings	It is used to display a collated list of all @RequestMapping paths.	True
shutdown	It is used to allow the application to be gracefully shutdown.	True
trace	It is used to display trace information.	True

Spring Boot aktuator podešavanja

- Spring Boot omogućava sigurnost za sve krajnje tačke aktuatora.
  Koristi autenftifikaciju koja daje korisnički ID kao korisnika i nasumično generisanu lozinku.
- Mi možemo pristup krajnjim tačkama ograničiti, tako što ćemo prilagoditi sigurnost osnovne autentifikacije krajnjim tačkama.
- Primer:

```
management.security.enabled=true
management.security.roles=ADMIN
security.basic.enabled=true
security.user.name=admin
security.user.passowrd=admin
```

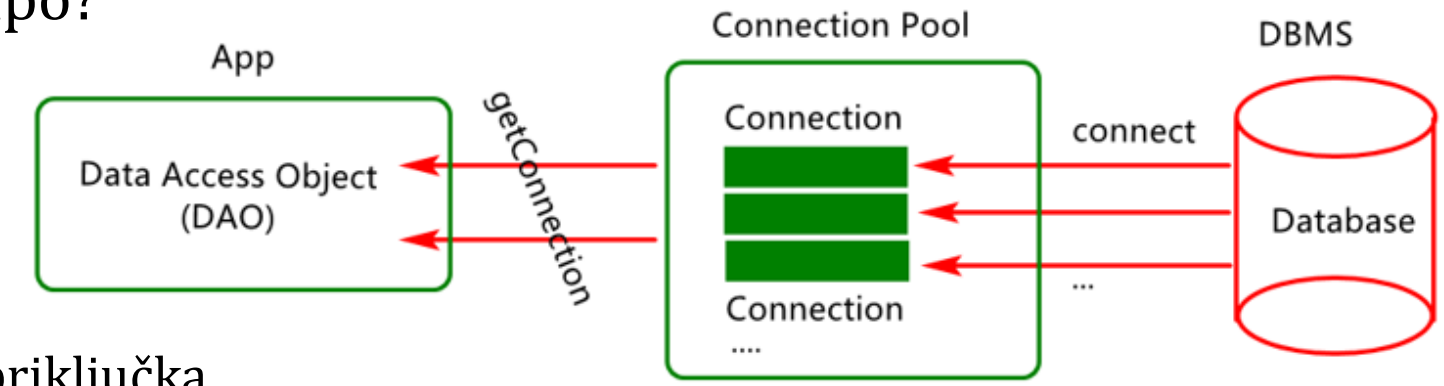
Spring Boot JDBC

- Spring Boot obezbeđuje starter i biblioteke za povezivanje aplikacije sa JDBC.
- U Spring Boot JDBC, bean-ovi koji se povezuju sa bazom, kao što su *DataSource*, *JdbcTemplate*, *NamedParameterJdbcTemplate*, se autokonfigurišu i kreiraju tokom pokretanja. Možemo automatski povezati (sa @Autowired) ove klase, ako želimo da ih koristimo.
- Primer:

```
@Autowired
JdbcTemplate jdbcTemplate;
@Autowired
private NamedParameterJdbcTemplate jdbcTemplate;
```

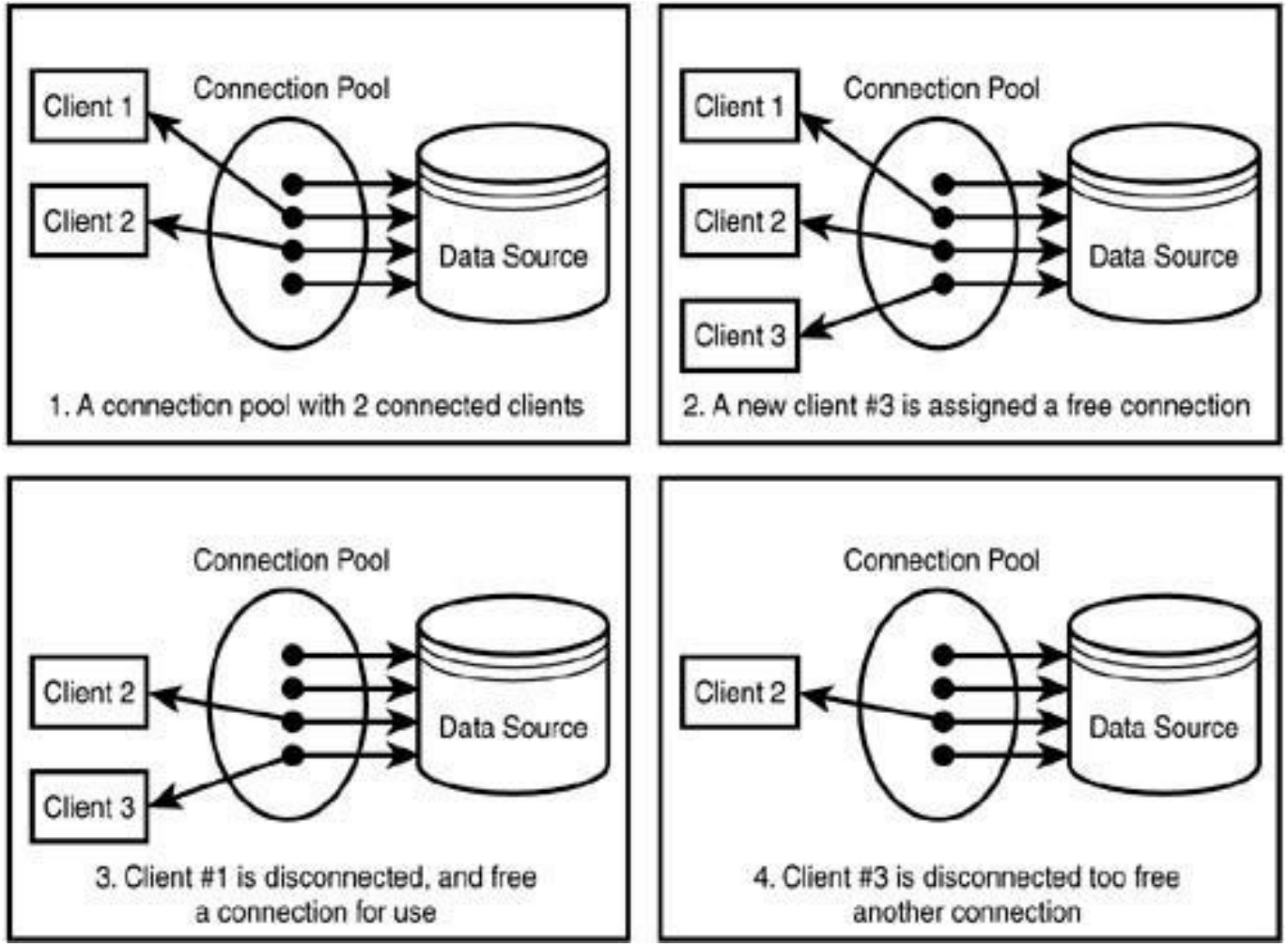
Udruživanje (*pooling*) JDBC konekcija

- Udruživanje JDBC konekcija je mehanizam koji upravlja višestrukim zahtevima za povezivanje sa bazom podataka.
- Olakšava ponovnu upotrebu veze, memoriše keš svih veza sa bazama (skup veza).
- Modul za udruživanje veza održava kao sloj na vrhu bilo kog standardnog JDBC drajverskog proizvoda.
- U fajlu application.properties konfiguriše se DataSource i mehanizam udruživanja Zašto je kreiranje konekcije skupo?
  - Otvaranje konekcije ka bazi
  - Autentifikacija korisnika
  - Kreiranje TCP priključka za čitanje/upisivanje podataka
  - Slanje/primanje podataka preko priključka
  - Zatvarane konekcije
  - Zatvaranje TCP priključka



Šta je uloga udruživanja (*pooling*)?

- Povećava brzinu pristupa podacima i smanjuje broj veza sa bazom podataka za aplikaciju.
- Poboljšava performanse aplikacije.
- Glavni zadaci su:
  - Upravljanje dostupnom konekcijom
  - Dodeljivanje nove konekcije
  - Zatvaranje konekcije



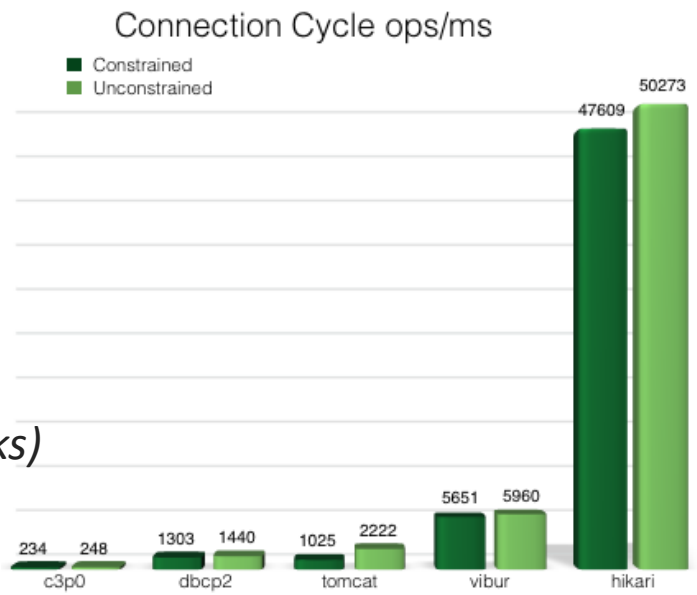
Radni okviri za udruživanje: karakteristike

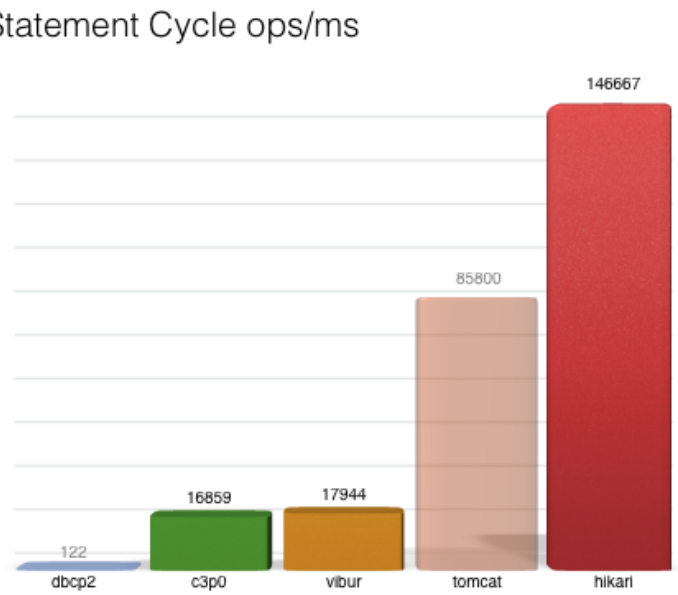
- Postoji veći broj radnih okvira, koje biramo na osnovu karakteristika.
- Pouzdanost:
  - Lako se konfiguriše.
  - Obratite pažnju na otvorene defekte u biblioteci.
  - Voditi računa o problemima sa zastojima (*deadlocks*).
- Performanse:
  - Obratite pažnju na podešavanja i okruženje za testiranje.
  - Rezultati testiranja u velikoj meri zavise od podešavanja konfiguracije.
- Podrška:
  - Dobra dokumentacija.
  - Velika aktivna zajednica i široka upotreba.
- Najpopularniji radni okviri: *Tomcat JDBC* i *HikariCP*.

Radni okvir HikariCP

- Najbrži mehanizam za upravljanje konekcijama.
- Prednosti radnog okvira:
  - dizajniran da bude deadlock-free;
  - može sam otkriti curenje konekcija;
  - pruža dobre podrazumevane vrednosti za konfiguraciju;
  - pronalazi dobru ravnotežu između nepreopterećenih korisnika sa mnogo konfiguracije i mnogo funkcionalne konfiguracije;
  - ultra laan (samo 130Kb);
  - odlični rezultati testova upoređivanja (benchmark);
  - mali broj defekata (bagova).
- Nema formule za max skup

  *no_connections = ((2 * core_count) + no_of_disks)*





Primer sa Hikari ili Tomcat mehanizmom

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.zaxxer</groupId>
      <artifactId>HikariCP</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>10.1.7</version>
</dependency>
```

- Način da *HikariCP* zamenimo sa *Tomcat* mehanizmom.
- Ne pišemo klasu @Configuration i programski definešemo @Bean sa izvorom podataka.
- Spring Boot može autokonfigurisati H2 unutarmemorijsku bazu:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
    <scope>runtime</scope>
 </dependency>
```

- Alternativno, preskakanje algoritma za pretraživanja skupa konekcija:

```
//u fajlu application.properties:
spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource
// other spring datasource properties
```

Dodatna svojstva u Tomcat mehanizmu

- Za optimizaciju performansi i ispunjavanje nekih specifičnih zahteva možemo dodati u fajlu *application.properties*, još neka svojstva:

```
spring.datasource.tomcat.initial-size=15
spring.datasource.tomcat.max-wait=20000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.max-idle=15
spring.datasource.tomcat.min-idle=8
spring.datasource.tomcat.default-auto-commit=true
```

Primer bez mehanizma upravljanja konekcijama

```
//DB.java
@Configuration
public class DB {

   @Bean
   public static DataSource source(){
      DriverManagerDataSource ds = new DriverManagerDataSource();
      ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
      ds.setUrl("jdbc:mysql://localhost:3306/mojabaza");
      ds.setUsername("root");
      ds.setPassword("");

      return ds;
   }
}
```

- Realizujemo klasu sa @Configuration anotacijom i metodu koja implementira izvor podataka.
- Postavljamo drajver koji koristimo, URL putanju do instance baze, parametre autentifikacije.
- U sloju podataka, repozitorijumske klase imaju metode koje izvršavaju upite nad bazom i koriste ovu konekciju.

Prednosti Spring Boot u odnosu na Spring

Spring Boot	Spring
Neophodna samo zavisnost **spring-boot-starter-jdbc**	U Spring JDBC, više zavisnosti je neophodno da se konfiguriše kao što su **spring-jdbc** i **spring-context**
Automatski konfiguriše *Datasource bean*, ako se ne održava eksplicitno. Ako ne želimo da koristimo bean, mi ćemo postaviti svojstvo **spring.datasource.initialize** na vrednost **false**.	In Spring JDBC, it is necessary to create a database bean either using **XML** or **javaconfig**.
Nije potrebno da registrujemo Template bean-ove, jer on automatski registruje bean-ove.	Moraju se zasebno registrovati Template bean-ovi kao što su **PlatformTransactionManager, JDBCTemplate, NamedParameterJdbcTemplate**
Sve skripte za inicijalizaciju baze podataka su memorisane u .SQL fajlu, i automatski se izvršavaju.	Ukoliko bilo koja skripta za inicijalizaciju (kao što su brisanje ili kreiranje tabele) je kreirana u SQL fajlu, ta informacija je potrebna da se eksplicitno stavi u konfiguraciju.

Razlike JDBC i Hibernate

JDBC	Hibernate
JDBC je tehnologija.	Hibarnate je radni okvir za objektno-relaciono mapiranje (ORM).
Korisnik je odgovoran za otvaranje i zatvaranje konekcija.	Sistem u realnom vremenu brine o otvaranju i zatvaranju konekcija.
Nije podržano lenjo učitavanje (*lazy loading*).	Podržano je lenjo učitavanje, koje daje bolje performanse.
Ne podržava asocijacije (konekcije između dve odvojene klase).	Podržane su asocijacije.

Spring Boot CRUD operacije

- CRUD – *Create*, *Read*/*Retrieve*, *Update*, *Delete*;
  4 osnovne funkcije za perzistiranje skladišta podataka:
  - CREATE operacija: Izvršava INSERT naredbu za kreiranje novog zapisa.
  - READ operacija: Čita zapis tabele na osnovu ulaznog parametra.
  - UPDATE operacija: Izvršava naredbu za ažuriranje tabele, na osnovu ulaznog parametra.
  - DELETE operacija: Briše specifičan red u tabeli, na osnovu ulaznog parametra.
- CRUD je orijentisan na podatke i standardizovanu upotrebu HTTP akcija:
  - POST: kreiranje novog resursa
  - GET: čitane resursa
  - PUT: ažuriranje postojećeg resursa
  - DELETE: brisanje resursa
- Unutar baze podataka, svaka od ovih operacija se direktno mapira u niz komandi.
- Odnos sa RESTful API je malo složeniji.

Spring Boot CRUD operacije

- Postoji dosta opcija za izvršavanje CRUD operacija.
- Jedna od najefikasnijih: kreiranje skupa uskladištenih (*stored*) procedura u SQL.
- Svako slovo CRUD može se mapirati u SQL naredbu i HTTP metodu.

Operacija	SQL upit	HTTP radnja	RESTful veb servis
Create	INSERT	PUT/POST	POST
Read	SELECT	GET	GET
Update	UPDATE	PUT/POST/PATCH	PUT
Delete	DELETE	DELETE	DELETE

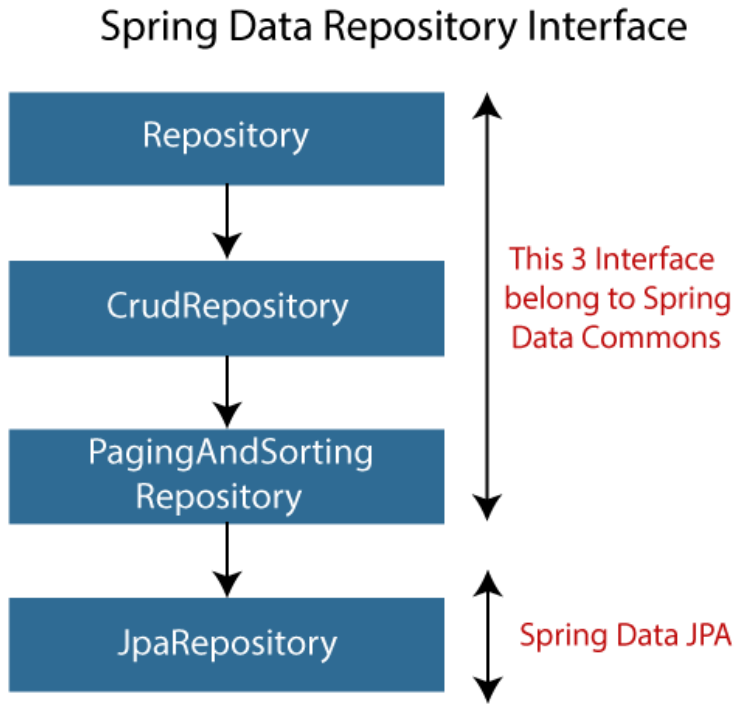
Spring Boot CRUD repozitorijum

- Spring Boot obezbeđuje interfejs, nazvan **CrudRepository**, koji ima ugrađene metode za CRUD operacije. On je izveden iz *Repository* interfejsa.
- Definisan u paketu: **org.springframework.data.repository**.
- Sintaksa: **public interface** CrudRepository<T,ID> **extends** Repository<T,ID>
  T - tip domena kojim skladište upravlja;
  ID – tip identifikatora entiteta kojim skladište upravlja.
- Primer:

```
public interface StudentRepository
    extends CrudRepository<Student, Integer> {
}
```

- JPA repozitorijum pruža JPA metode kao što su *flushing*, perzistiranje konteksta, brisanje zapisa u grupama.
- Primer:

```
public interface BookDAO extends JpaRepository { }
```



Razlike između repozitorijuma CRUD i JPA

- Interfejsi omogućavaju Springu da pronađe interfejs repozitorijuma i kreira proksi objekte do njih.
- Interfejsi pružaju metode koje nam omogućavaju da izvršimo neke uobičajne operacije.

CrudRepository	JpaRepository
Ne pruža nijednu metodu za paginaciju i sortiranje.	Proširuje *PagingAndSortingRepository*, koji pruža sve metode za implementiranje paginacije.
Radi kao interfejs markera.	Proširuje oba repozitorijuma: *CrudRepository* i *PagingAndSortingRepository*.
Pruža samo osnovne CRUD funkcije, kao što su: **findById(), findAll(),** itd.	Priža neke dodatne metode, zajedno sa metodama iz PagingAndSortingRepository i CrudRepository. Primer: **flush(), deleteInBatch().**
Koristimo kada nam nisu neophodne funkcije koje nam nudi *JpaRepository* i *PagingAndSortingRepository*.	Koristimo kada želimo da implementiramo paginaciju i funkcionalnost sortiranja u aplikaciji.

Primer sa CRUD operacijama

- Primer sa jednim modelom, i po jednim kontrolerom, servisom i repozitorijumom.

```
package rs.ac.bg.etf.models;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table
public class Books {
        //Defining book id as primary key
        @Id
        @Column
        private int bookid;                                    }
        @Column
        private String bookname;
        @Column
        private String author;
        @Column
        private int price;
        public int getBookid(){  return bookid;  }
        public void setBookid(int bookid){  this.bookid = bookid;  }
        public String getBookname(){ return bookname; }
        public void setBookname(String bookname){
                this.bookname = bookname;      }
        public String getAuthor(){  return author;  }
        public void setAuthor(String author){ this.author = author; }
        public int getPrice(){  return price;  }
        public void setPrice(int price){  this.price = price;  }
```

Primer sa CRUD operacijama (2)

```
package rs.ac.bg.etf.controllers;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import rs.ac.bg.etf.models.Books;
import rs.ac.bg.etf.services.BooksService;
 @RestController
public class BooksController {
        @Autowired
        BooksService booksService;
        @GetMapping("/book")
        private List<Books> getAllBooks(){
                return booksService.getAllBooks();  }
@GetMapping("/book/{bookid}")
private Books getBooks(@PathVariable("bookid") int bookid){
  return booksService.getBooksById(bookid); }
@DeleteMapping("/book/{bookid}")
  private void deleteBook(@PathVariable("bookid") int bookid){
                        booksService.delete(bookid);
  }
@PostMapping("/books")
  private int saveBook(@RequestBody Books books){
                booksService.saveOrUpdate(books);
                return books.getBookid();
  }
@PutMapping("/books")
  private Books update(@RequestBody Books books){
                booksService.saveOrUpdate(books);
                return books;
  }
}
```

Primer sa CRUD operacijama (3)

```
package rs.ac.bg.etf.services;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import rs.ac.bg.etf.models.Books;
import rs.ac.bg.etf.repository.BooksRepository;
@Service
public class BooksService {
  @Autowired
  BooksRepository booksRepository;
  //dohvatanje svih zapisa koriscenjem ugradjene metode iz CrudRepository
  public List<Books> getAllBooks(){
    List<Books> books = new ArrayList<Books>();
    booksRepository.findAll().forEach(books1 -> books.add(books1));
    return books;
  }
  public Books getBooksById(int id){
    return booksRepository.findById(id).get();  }
  //snimanje zapisa koriscenjem metode save() iz
  //CrudRepository
  public void saveOrUpdate(Books books){
    booksRepository.save(books);
  }
  public void delete(int id){
    booksRepository.deleteById(id);
  }
  public void update(Books books, int bookid){
    booksRepository.save(books);
  }
}
```

```
package rs.ac.bg.etf.repository;
import
      org.springframework.data.repository.CrudRepository;
import rs.ac.bg.etf.models.Books;
public interface BooksRepository
                  extends CrudRepository<Books, Integer>  {
}
```

Validacija za RESTful servise

- Korišćenje Java Validation API:

```
package rs.etf.server.main.user;
import java.net.URI;
import java.util.List;
import javax.validation.Valid;
  ...
@RestController
public class UserResource {
  @Autowired
  private UserDaoService service;
  @GetMapping("/users")
  public List<User> retriveAllUsers(){
    return service.findAll();  }
  @GetMapping("/users/{id}")
  public User retriveUser(@PathVariable int id){
    User user= service.findOne(id);
    if(user==null) throw new UserNotFoundException("id: "+ id);
              return user;
 }
```

```
@DeleteMapping("/users/{id}")
  public void deleteUser(@PathVariable int id){
    User user= service.deleteById(id);
    if(user==null) throw new UserNotFoundException("id: "+ id);
  }
  @PostMapping("/users")
  public ResponseEntity<Object> createUser(@Valid
                                    @RequestBody User user){
    User sevedUser=service.save(user);
    URI location=ServletUriComponentsBuilder.
                  fromCurrentRequest().path("/{id}").
                  buildAndExpand(sevedUser.getId()).toUri();
    return ResponseEntity.created(location).build();
  }
}
```

Validacija za RESTful servise (2)

```
package rs.etf.server.main.user;
import java.util.Date;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
public class User {
  private Integer id;
  @Size(min=5,
        message= "Ime treba da ima najmanje 5 karaktera")
  private String name;
  @Past
  private Date dob;
  //default constructor
  protected User(){ }
   public User(Integer id, String name, Date dob){
            super();
            this.id = id;
            this.name = name;
            this.dob = dob;
  }
  public Integer getId(){  return id;  }
   public void setId(Integer id){ this.id = id; }
   public String getName(){ return name; }
   public void setName(String name){ this.name = name; }
   public Date getDob(){ return dob; }
   public void setDob(Date dob){ this.dob = dob; }
   @Override
   public String toString(){
      //return "User [id="+id+", name="+name+", dob="+dob + "]";
      return String.format("User [id=%s, name=%s, dob=%s]", id,
                                    name, dob);
   }
}
```

Validacija za RESTful servise (3)

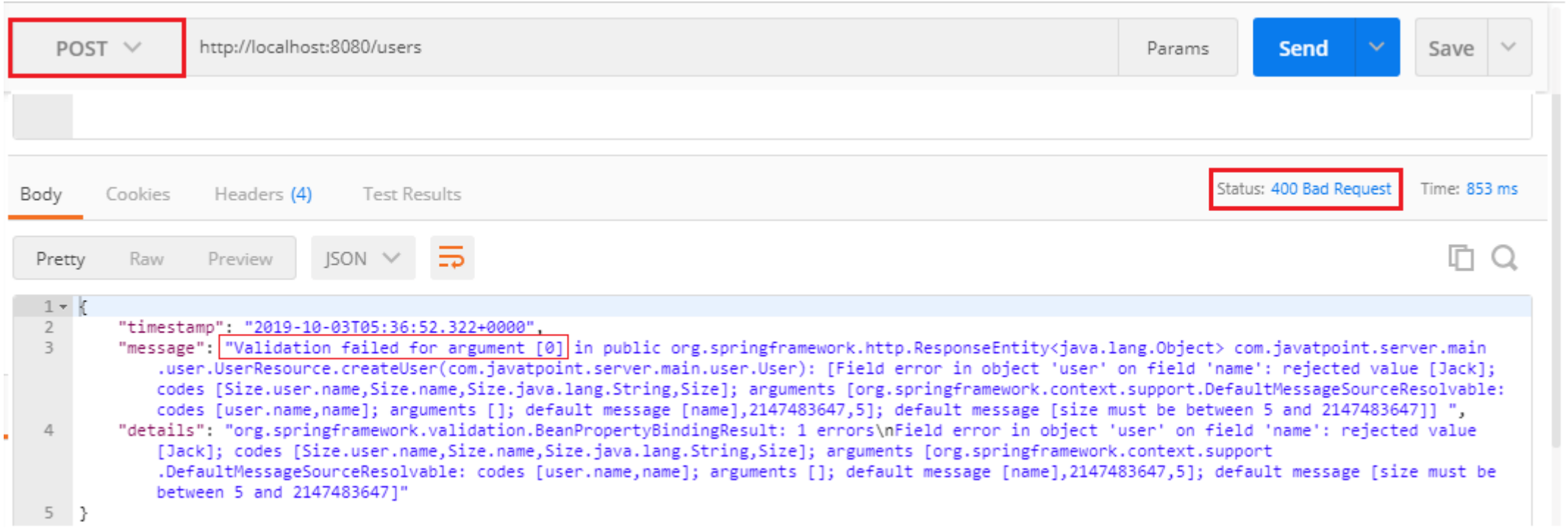
```
package rs.etf.server.main;
import java.util.Date;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
import com.javatpoint.server.main.exception.ExceptionResponse;
import com.javatpoint.server.main.user.UserNotFoundException;

@ControllerAdvice                                   //Definisemo obradu izuzetka za sve izuzetke
@RestController
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler  {
```

Validacija za RESTful servise (4)

```
@ExceptionHandler(Exception.class)
public final ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)  {
  //definisanje structure odgovora za izuzetak
  ExceptionResponse exceptionResponse= new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));
  return new ResponseEntity(exceptionResponse, HttpStatus.INTERNAL_SERVER_ERROR); //vracanje strukture odgovora i statusa
}
@ExceptionHandler(UserNotFoundException.class)
public final ResponseEntity<Object> handleUserNotFoundExceptions(UserNotFoundException ex, WebRequest request) {
  //definisanje strukture odgovora za izuzetak
  ExceptionResponse exceptionResponse= new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));
  return new ResponseEntity(exceptionResponse, HttpStatus.NOT_FOUND); //vracanje strukture odgovora i statusa
}
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers,
                                                              HttpStatus status, WebRequest request)   {
  ExceptionResponse exceptionResponse= new ExceptionResponse(new Date(), ex.getMessage(), ex.getBindingResult().toString());
  return new ResponseEntity(exceptionResponse, HttpStatus.BAD_REQUEST); //vracanje strukture odgovora i statusa
  }
}
```

Uhvaćene greške

- POST zahtev poslat iz Postman alata:



Internacionalizacija (I18N)

- Proces dizajniranja veb aplikacije ili servisa tako da se pruži automatska podrška za različite zemlje, različite jezike, bez unošenja velikih promena unutar aplikacije.
- Lokalizacija se izvršava dodavanjem komponenti specifičnih za prevedeni tekst, podataka koji opisuju lokalno ponašanje, itd.
- Priža punu integraciju u klase i pakete koji pružaju funkcionalnost koja zavisi od jezika ili kulture.
- Java pruža osnovu za internacionalizaciju i za desktop i za serverske aplikacije.

- Dve stvari treba da se konfigurišu da bi servis bio internacionalan:
  - LocaleResolver
  - ResourceBundleMessageSource

Internacionalizacija - podrška

- Predstavljanje teksta: Java je zasnovana na Unicode skupu znakova, i nekoliko biblioteka implementira Unicode standard.
- Identifikacija i lokalizacija: *Locale* u Javi su identifikatori, koji se koriste za zahtevanje ponašanja specifičnog za lokalizaciju.
  Klasa *ResourceBundle* podržava lokalizaciju i obezbeđuje pristup lokalnim specifičnim objektima, uključujući i nizove.
- Rukovanje datumom i vremenom: Java nudi različite kalendare.
  Podržava konverziju u/iz kalendarski nezavisnih objekata datuma.
  Podržava sve vremenske zone na svetu.
- Obrada teksta: uključuje analizu znakova, mapiranje velikih i malih slova, poređenje stringova, razvijanje teksta u reči, formatiranje brojeva, datuma i vrednosti vremena u nizove ili njihovo raščlanjivanje iz stringova.
- Kodiranje znakova: podrška za pretvarane između Unicode i drugih kodiranja.

Internacionalizacija

- Podrazumevana vrednost za *Locale* je *Locale.US*.
- Ova vrednost se dobija uvek ukoliko nije definisana lokacija.
- Svojstva se čuvaju u objektu *ResourceBundle*.
- *ResourceBundleMessageSource* je koncept u Spring MVC za dohvatanje svojstava.
- Nakon toga koristi se *MessageSource* i zaglavlje *Accept-Language*.
- Konfigurisanje bean-a na podrazumevanu vrednost:

```
@Bean
public  LocaleResolver localeResolver() {
  SessionLocaleResolver localeResolver = new SessionLocaleResolver();
  localeResolver.setDefaultLocale(Locale.US);
  return localeResolver;
}
```

Fajlovi sa svojstvima

- messages.properties:

```
good.morning.message=Good Morning
good.evening.message=Good Evening
```

- messages_fr.properties

```
good.morning.message=Bonjour
good.evening.message=Bonne soirée
```

- messages_de.properties

```
good.morning.message=Guten Morgen
good.evening.message=Good Abend
```

Konfigurisanje aplikacije

```
package rs.etf.server.main;
import java.util.Locale;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
@SpringBootApplication
public class RestfulWebServicesApplication{
public static void main(String[] args){
        SpringApplication.run(RestfulWebServicesApplication.class, args);
}
```

```
//konfiguracija podrazumevanog Locale
@Bean
public  LocaleResolver localeResolver(){
        SessionLocaleResolver localeResolver =
                new SessionLocaleResolver();
        localeResolver.setDefaultLocale(Locale.US);
        return localeResolver;
}

//konfiguracija ResourceBundle
@Bean
public ResourceBundleMessageSource messageSource(){
        ResourceBundleMessageSource
messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("messages");
        return messageSource;
 }
}
```

Upotreba u kontroleru

```
package rs.etf.server.main.helloworld;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;
import java.util.Locale;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.i18n.LocaleContextHolder;
@Configuration
@RestController
public class HelloWorldController {
  @Autowired
  private MessageSource messageSource;
  @GetMapping(path="/hello-world")
  public String helloWorld(){
    return "Hello World";
  }
```

```
  @GetMapping(path="/hello-world-bean")
  public HelloWorldBean helloWorldBean(){
    return new HelloWorldBean("Hello World");
  }
  @GetMapping(path="/hello-world/path-variable/{name}")
  public HelloWorldBean helloWorldPathVariable(@PathVariable
String name) {
  return
    new HelloWorldBean(String.format("Hello World, %s",name));
}
  //internacionalizacija
  @GetMapping(path="/hello-world-internationalized")
  public String helloWorldInternationalized(@RequestHeader(
          name="Accept-Language", required=false) Locale locale) {
   return messageSource.getMessage("good.morning.message",
                                   null, LocaleContextHolder.getLocale());
  }
}
```

Konfigurisanje aplikacije (2)

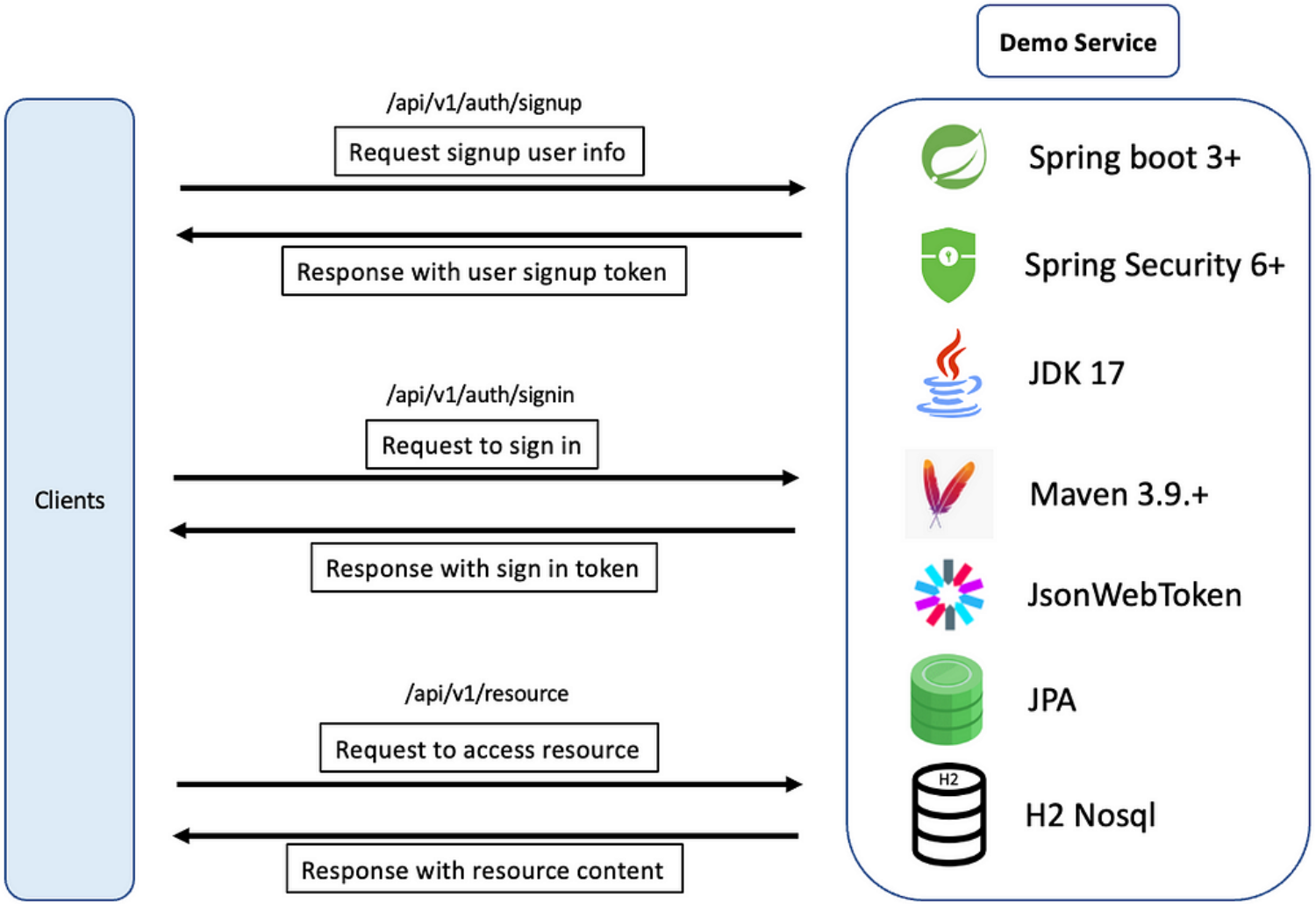
```
package rs.etf.server.main;
import java.util.Locale;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver;
@SpringBootApplication
public class RestfulWebServicesApplication{
public static void main(String[] args){
        SpringApplication.run(RestfulWebServicesApplication.class, args);
}
```

Prednost AcceptHeaderLocaleResolver je što ne moramo da konfigurišemo zaglavlje zahteva kao parametar, u svakom metodu kontrolera.

```
@Bean
public  LocaleResolver localeResolver(){
  AcceptHeaderLocaleResolver localeResolver =
                    new AcceptHeaderLocaleResolver();
  localeResolver.setDefaultLocale(Locale.US);
  return localeResolver;
 }
}
//konfiguracija ResourceBundle
// može i definisanjem u application.properties:
//spring.messages.basename=messages
@Bean
public ResourceBundleMessageSource messageSource(){
        ResourceBundleMessageSource
messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("messages");
        return messageSource;
 }
}
```

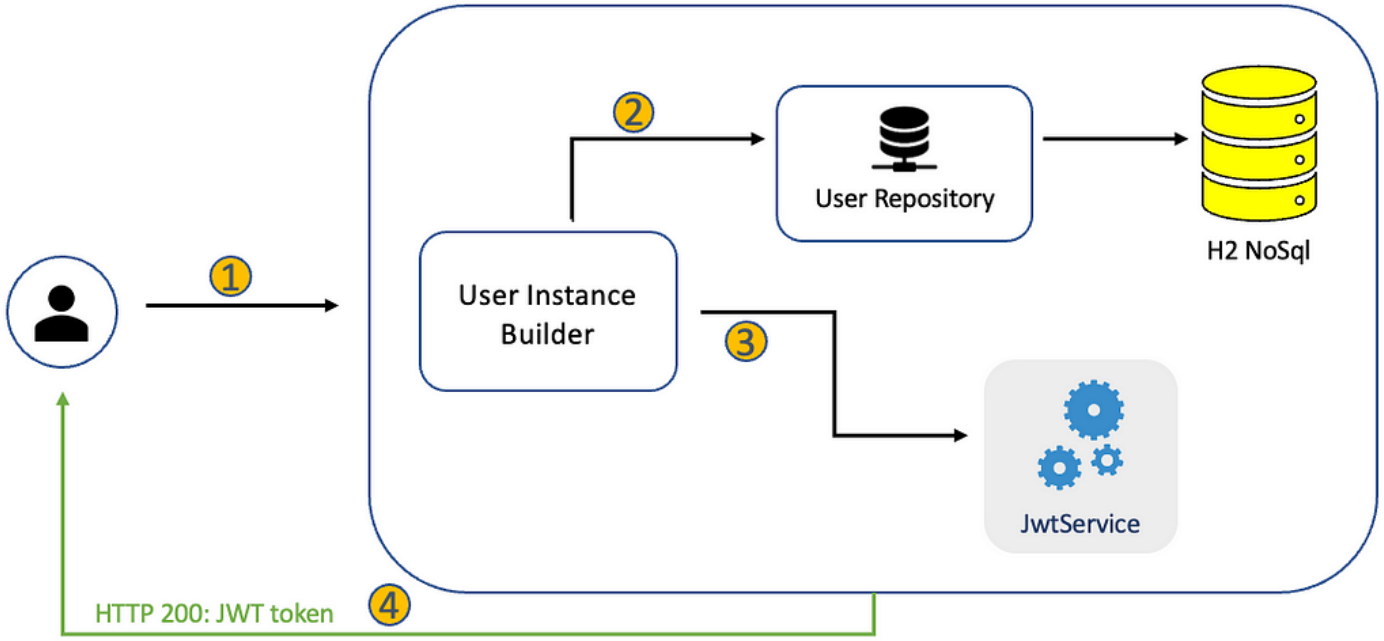
JWT autentifikacija i autorizacija

- Postoji više načina za autentifikaciju naših RESTful veb servisa.
- U osnovnoj autentifikaciji šaljemo korisničko ime i lozinku, kao deo našeg zahteva.
- Kod malo naprednijih oblika autentifikacije, ne šalje se stvarna lozinka serveru, već se šalje sažetak.
- Najnapredniji oblici autentifikacije danas su OAuth i OAuth2.

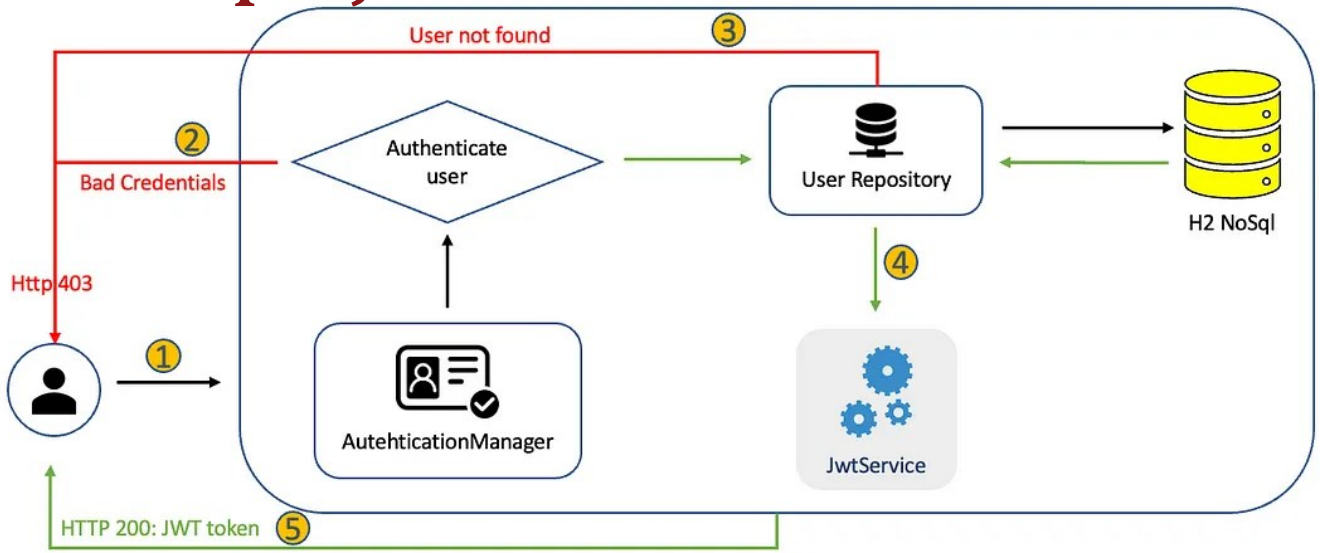


Dijagram toka registracije korisnika

- 1) Proces počinje kada korisnik podnese zahtev servisu.
- Korisnički objekat se generiše iz podataka zahteva, pri čemu se lozinka kodira pomoću *PasswordEncoder*.
- 2) Korisnički objekat se čuva u bazi podataka koristeći *UserRepository*, koji koristi *Spring Data JPA*.
- 3) *JwtService* se poziva da generiše JWT za objekat *User*.
- 4) JWT se enkapsulira u JSON odgovor i potom se vraća korisniku.

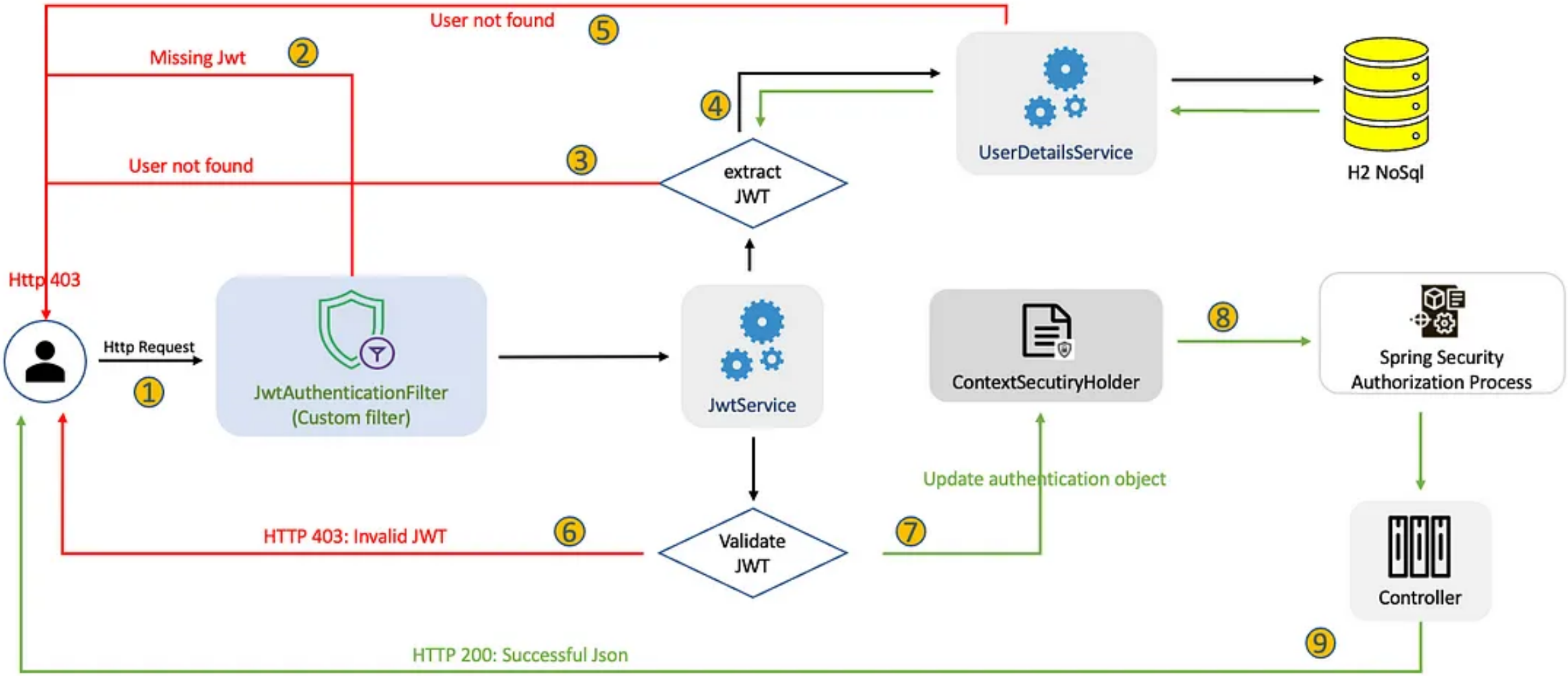


Dijagram toka prijave u sistem



- 1) Proces počinje kada korisnik šalje zahtev servisu.
  Autentifikacioni objekat se tada generiše, da dostavi korisničko ime i lozinku.
- 2) AuthenticationManager je zadužen za autentifikaciju autent. objekta, i hvatanje svih neophodnih zahteva.
  Ako su korisnički ime ili lozinka nekorektni, izuzetak će biti uhvaćen, a odgovor HTTP statusa 403 će biti vraćen korisniku.
- 3) Nakon uspešne autentifikacije, pokušava se dohvatiti korisnik iz baze podataka. Ukoliko korisnik ne postoji, odgovor HTTP status 403 se vraća korisniku.
- 4) Nakon što imamo korisničke informacije, generišemo JWT iz servisa.
- 5) JWT se enkapsulira u JSON odgovor, koji se vraća korisniku.

Korišćenje Spring Security

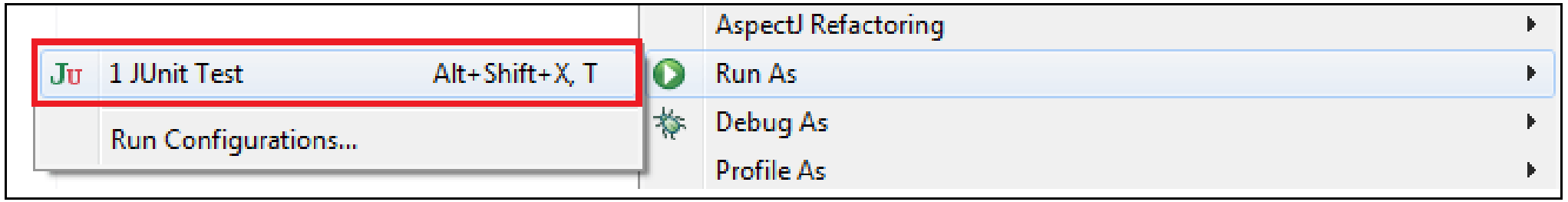


Spring Boot Starter Test

- Zavisnost **spring-boot-starter-test** je glavna za proces testiranja.
- Sadrži glavne elemente neophodne za naše testove.
- Postoji nekoliko različitih tipova testova, koje možemo pisati za testiranje aplikacije i automatizaciju zdravstvenog stanja aplikacije.
- Kada kreiramo Spring Boot aplikaciju, ona će u pom.xml fajlu imati zavisnosti za testirane, a testove pišemo u folderu: src/test/java

Primer jediničnog testa

```
package com.javatpoint.springboottestexample;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
class SpringBootTestExampleApplicationTests  {
    @Test
    void contextLoads() {
      //sadrzaj testa
    }
}
```



Integracioni test (za testiranje konekcije)

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootTomcatConnectionPoolIntegrationTest {

    @Autowired
    private DataSource dataSource;

    @Test
    public void givenConnectionPoolInstance_whenCheckedPoolClassName_thenCorrect() {
        assertThat(dataSource.getClass().getName()).isEqualTo("org.apache.tomcat.jdbc.pool.DataSource");
    }
}
```

Rezime

- Aplikativna klasa je ulazna tačka u Spring Boot aplikaciju i ona je označena sa @SpringBootApplication.
- Klasa koja je označena sa @RestController, označava da je to kontroler za veb servis.
- Osnovna URL putanja za sve krajnje tačke vezane za korisnike definiše se korišćenjem @RequestMapping(“/users”).
- Metoda koja mapira detalje o nekom korisniku, treba da sadrži obrazac /{userId} a ispred metode treba da se nađe @GetMapping(“/{userId}”). Promenljiva putanje userId path se ekstrahuje korišćenjem @PathVariable i prosleđuje kao parametar metode.
- Starteri su početna podešavanja projekta, a aktuatori su potprojekti koji nam pomažu da nadgledamo i upravljamo Spring Boot aplikacijom.
- Za proces testiranja Spring Boot aplikacije možemo koristiti jedinične testove.

Hvala na pažnji ☺

PITANJA?