Spring podrška za veb aplikacije

Autori:
Goran Savić
Milan Segedinac

1. Spring Boot

Pre nego što pređemo na implementaciju veb aplikacija korišćenjem Spring radnog okvira, prikazaćemo Spring Boot projekat, koji pojednostavljuje razvoj Spring aplikacija i u poslednje vreme se standardno koristi za razvoj veb aplikacija u Springu.

Spring Boot je Spring-bazirani radni okvir za pojednostavljeni razvoj Spring aplikacija. Spring Boot pojednostavljuje konfigurisanje i razvoj aplikacije kroz skup gotovih rešenja, tako da se jednostavnije dobija konfigurisana Spring aplikacija. Kada god je moguće, Spring Boot automatski konfiguriše Spring kontejner. Pored toga, ako je reč o veb aplikaciji, samo pokretanje aplikacije je jednostavnije jer Spring Boot sadrži ugrađen veb server. Takođe, jednostavnije je upravljanje zavisnostima aplikacije obzirom da Spring Boot agregira standardno korišćene zavisnosti. Generalno, glavna ideja Spring Boot projekta je da se programer inicijalno fokusira na razvoj aplikacije umesto na njen životni ciklus (konfiguraciju, postavljanje, upravljanje projektom, ...).

Kada je reč o upravljanju zavisnostima u Spring Boot aplikaciji, Spring Boot okuplja veći broj srodnih zavisnosti u jednu zavisnost. Agregirana zavisnost dalje zavisi od drugih zavisnosti koje će biti automatski dobavljene, jer postoji tranzitivnost zavisnosti. Na ovaj način spisak zavisnosti aplikacije je kraći i jednostavniji za održavanje. Npr. zavisnost *spring-boot-starter-web* sadrži sve zavisnosti potrebne za rad sa veb aplikacijom.

Pogledajmo u sledećem primeru kako izgleda Maven pom.xml fajl za veb orijentisanu Spring Boot aplikaciju.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>vp.spring</groupId>
  <artifactId>SpringBootCountry</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>SpringBootCountry</name>
  <url>http://maven.apache.org</url>

  <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

Vidimo da Spring boot aplikacija referencira zavisnosti iz *spring-boot-starter-parent* projekta postavljajući ga za roditeljski projekat. Pored toga, obzirom da su nam potrebne funkcionalnosti za veb aplikacije, u spisku zavisnosti je i *spring-boot-starter-web*. Definisanjem samo ove dve zavisnosti, projekat će tranzitivno učitati desetine biblioteka potrebnih za rad Spring veb aplikacije.

Kao što smo pomenuli, podrška za automatsku konfiguraciju smanjuje količinu konfiguracija u Spring kodu. Na osnovu klasa postavljenih u spisak zavisnosti, Spring Boot automatski pretpostavlja koji su potrebni Bean objekti i konfiguriše ih tako da budu raspoloživi za korišćenje bez potrebe ručnog konfigurisanja.

Čak i kada je reč o veb aplikaciji, Spring Boot aplikacija se pokreće pokretanjem obične Java klase koja ima main metodu. Ova klasa se anotira anotacijom @SpringBootApplication. Ova anotacija zamenjuje tri standardne Spring anotacije i to: @Configuration, @EnableAutoConfiguration i @ComponentScan. U samoj main metodi potrebno je startovati Spring Boot aplikaciju na način prikazan u sledećem primeru.

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

2. Apache Tomcat

U ranijim lekcijama smo se sreli sa pojmom veb servera. Veb server je aplikacija zadužena za prijem i obradu klijentskih zahteva u veb aplikaciji. Obzirom na važnost ove komponente veb aplikacije, kao i na kompleksnost implementacije, standardno se u Java veb aplikacijama veb serveri ne implementiraju nego se koriste postojeći veb serveri (slično kao što smo za rad sa bazama podataka koristili neki od postojećih sistema za upravljanje bazom podataka). Korišćenje postojećeg veb servera oslobađa programera pisanja koda koji vrši mrežnu komunikaciju sa klijentom, ali se podrazumeva da je neophodno napisati kod koji će izvršiti obradu zahteva. Dakle, veb server će obezbediti prijem zahteva i slanje odgovora, ali je sam sadržaj odgovora odgovornost programskog koda korisničke aplikacije koja je postavljena na veb server.

Ovde ćemo prikazati Apache Tomcat, kao vrlo često korišćen veb server u Java veb aplikacijama. Apache Tomcat je besplatan softver otvorenog koda koji omogućuje isporuku statičkih veb sadržaja, ali i dinamičkih veb sadržaja generisanih putem Java koda. Pod statičkim veb sadržajima smatraju se svi kreirani veb resursi (HTML, CSS, JavaScript fajlovi) koji se sa definisanim sadržajem isporučuju klijentu bez potrebe dinamičkog kreiranja, modifikacije ili procesiranja sadržaja za svaki pojedinačan zahtev. Sa druge strane, dinamički sadržaji se programski generišu u kodu. Na primer, ukoliko je na veb stranici potrebno prikazati listu osoba, ovaj sadržaj mora biti dinamički kreiran, jer lista osoba zavisi od trenutno uskladištenih podataka u aplikaciji i nije moguće napraviti statički HTML fajl sa ovim podacima. Treba pomenuti da veb stranica najčešće kombinuje statički i dinamički sadržaj. Za primer sa listom osoba, naslov na stranici i zaglavlje spiska je uvek isto i može se statički definisati, dok sam spisak osoba mora biti dinamički dodat na stranicu.

Apache Tomcat se instalira raspakivanjem u proizvoljan folder na disku. Pogledajmo strukturu Apache Tomcat foldera.

- bin – izvršni fajlovi za pokretanje i zaustavljanje servera, kao i pomoćni programi za rad sa serverom

- conf – konfiguracioni fajlovi za podešavanje servera
- lib – Java biblioteke (jar fajlovi) koje aplikacije postavljene na server koriste. Ovde se standardno stavljaju standardne biblioteke koje koristi većina aplikacija, dok se biblioteke specifične za određenu aplikaciju postavljaju u folder u kojem je sama aplikacija
- logs – log fajlovi u kojima se evidentiraju događaji na serveru (npr. tekst izuzetka kada se desi greška u izvršavanju aplikacije)
- temp – privremeni fajlovi
- webapps – u ovaj folder se postavlja kod veb aplikacija koje se izvršavaju na serveru
- work – folder u kojem Tomcat skladišti fajlove koje je dinamički izgenerisao u toku rada

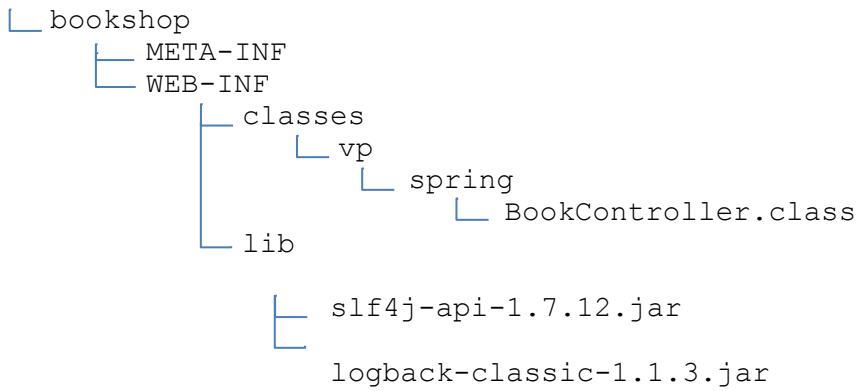
Analizirajmo detaljnije webapps folder u kojem se nalaze veb aplikacije. Svaka veb aplikacija se nalazi u posebnom folderu unutar webapps foldera. Ime foldera aplikacije predstavlja putanju preko koje će se veb zahtevi obraćati konkretnoj Tomcat aplikaciji. Delovi aplikacije moraju biti organizovani u podfoldere prema sledećem pravilu:

- statički sadržaji se smeštaju u proizvoljnu strukturu u okviru foldera aplikacije i biće dostupni za klijenta u skladu sa ovom strukturom. Na primer, za Tomcat server startovan na računaru koji ima IP adresu 192.168.0.2 i zahteve očekuje na portu 8080, zahtev http://192.168.0.2:8080/bookshop/pages/books.html će pronaći i vratiti fajl books.html ako se fajl nalazi na sledećoj lokaciji:

```
apache-tomcat-9.0
└─ webapps
   └─ bookshop
      └─ pages
         └─ books.html
```

- Java kod koji generiše dinamičke sadržaje se takođe nalazi u folderu aplikacije i to u podfolderu WEB-INF/classes. U ovom folderu se nalazi hijerarhija potfoldera u skladu sa hijerarhijom paketa u kojima su klase sa Java kodom. Paketi sadrže .class fajlove. Mapiranje URL-a iz klijentskog zahteva na Java kod koji će obraditi taj zahtev vrši se kroz konfiguraciju u aplikaciji. U narednom poglavlju prikazaćemo kako se ovakvo mapiranje definiše u Spring veb aplikaciji. Pored foldera classes, u WEB-INF folderu mogu da se nalaze i drugi fajlovi koje veb aplikacija koristi. Takođe, deo konfiguracije aplikacije se definiše u fajlovima koji se nalaze u META-INF podfolderu foldera aplikacije. Biblioteke koje aplikacija koristi se smeštaju u lib folderu u okviru WEB-INF foldera. Pogledajmo primer strukture ovog dela Tomcat foldera ako se generisanjem dinamičkog sadržaja bavi klasa BookController iz paketa vp.spring.

```
apache-tomcat-9.0
└─ webapps
```



Kao što smo videli, aplikacija mora biti smeštena u webapps folder i mora imati predefinisanu strukturu. Najčešće se aplikacija na server postavlja u webapps folder u zapakovanom obliku kao ZIP arhiva sa ekstenzijom .war. Tomcat server automatski vrši raspakivanje war fajlova koji se nalaze u webapps folderu.

Apache Tomcat se na Windows operativnom sistemu pokreće pokretanjem fajla startup.bat iz foldera bin. Nakon pokretanja, server očekuje klijentske zahteve na određenom portu. Podrazumevani port je 8080. Server se zaustavlja pokretanjem fajla shutdown.bat iz bin foldera.

3. Spring Boot veb aplikacija

Spring kontejner sadrži veb modul sa vrlo kompleksnom podrškom za veb aplikacije. Ovaj modul se oslanja na Spring MVC radni okvir, što je skraćenica od Model-View-Controller. MVC je često korišćen dizajn šablon u veb aplikacijama. Šablon razdvaja aplikaciju u tri komponente. *Model* komponenta je zadužena za reprezentaciju i upravljanje podacima. *View* komponenta se bavi vizuelnom reprezentacijom podataka, a *Controller* komponenta kontroliše tok akcija u aplikaciji. Na osnovu akcija u *View* komponenti, *Controller* inicira odgovarajuću reakciju koja može da izvrši izmenu nad podacima, nakon čega *View* komponenta prikazuje izmenjene podatke.

View komponenta u Spring MVC se zasniva na tehnologijama koje dinamički generišu odgovor klijentu na serverskoj strani. Ovo se naziva *server-side scripting* i o ovim tehnikama će biti reči u narednim lekcijama. Najčešće korišćena tehnologija ovog tipa je Java Server Pages (JSP). Ovakav pristup ne odgovara savremenim tehnologijama razvoja veb aplikacija koje sve više koriste tzv. *client-side scripting*, što podrazumeva dobijanje konačnog veb sadržaja izvršavanjem skript koda na klijentskoj strani u internet *browseru*. Iz tog razloga, ovde se nećemo baviti kompletnim Spring MVC radnim okvirom, već samo njegovom *Controller* komponentom koja omogućuje obradu klijentskog zahteva. Tako da ćemo se fokusirati na to kako možemo u Spring Boot aplikaciji postaviti statičke veb sadržaje, kao i napisati kod koji obrađuje klijentske zahteve i dinamički generiše veb sadržaj.

Da bi mogla da vrši mežnu komunikaciju sa klijentom, Spring Boot veb aplikaciju je potrebno postaviti u okviru veb servera. Spring Boot sadrži ugrađen Tomcat veb server. Pokretanjem main klase Spring Boot aplikacije vrši se postavljanje aplikacije na ugrađeni Tomcat server i pokretanje Tomcat servera. Moguće je projekat i konfigurisati tako da se aplikacija ne postavlja na ugrađeni veb server, već da se koristi proizvoljan eksterni veb server.

Java klase iz aplikacije će automatski biti postavljene na ugrađeni veb server. Statički veb sadržaji (HTML, CSS, JavaScript fajlovi) koje aplikacija koristi treba da budu postavljeni u predefinisane foldere u Spring Boot aplikaciji kako bi bili dostupni. Sadržaji će biti dostupni ako se postave u jedan od sledećih foldera u Spring Boot aplikaciji:

- `src/main/webapp/`
  - ako se aplikacija pakuje kao war, fajlovi iz ovog foldera biće spakovani u koren war fajla
- `/META-INF/resources/`
- `/resources/`
- `/static/`
- `/public/`

Što se tiče koda koji generiše dinamički veb odgovor, on se u Spring veb aplikaciji može napisati u klasi anotiranoj anotacijom @RestController. Ime anotacije je u vezi sa REST veb servisima o kojima će biti reči u narednim lekcijama. Generisanje odgovora se piše u metodama ove klase, pri čemu se za svaku metodu definiše za koje veb zahteve ona treba da bude pozvana da izgeneriše odgovor. Ovo se vrši mapiranjem podataka iz zahteva (URL-a i tipa HTTP metode) na ime metode. Mapiranje se sintaksno vrši anotacijom @RequestMapping. Pogledajmo ovo na sledećem primeru.

```
@RestController
public class CountryController {
	@RequestMapping(value="/test", method=RequestMethod.GET)
	public String testResponse() {
		return "Hello!";
	}
}
```

Ako je Tomcat server startovan na računaru sa IP adresom 192.168.0.2 i očekuje zahteve na portu 8080, tada će HTTP GET metoda sa zahtevom http://192.168.0.2:8080/test pozvati prikazanu metodu i vratiti klijentu odgovor “Hello”. Sam Spring MVC će izvršiti prijem i parsiranje zahteva i na osnovu sadržaja zahteva i definisanog mapiranja zaključiti koju metodu treba da pozove. Takođe, Spring kontejner će izvršiti i prosleđivanje rezultata metode klijentu putem mreže.

Pogledajmo sada nešto komplikovaniji primer u kojem se kao odgovor klijentu šalje spisak država evidentiranih u aplikaciji.

```
@RestController
public class CountryController {
	@Autowired
	CountryService countryService;

	@RequestMapping(value="api/countries", method = RequestMethod.GET)
	public String getAllCountries() {
		List<Country> countries = countryService.findAll();

		StringBuffer sb = new StringBuffer();
		for (Country c: countries) {
			sb.append(c).append("\n");
		}

		return sb.toString();
	}
	...
}
```

Prikazana metoda *getAllCountries* će biti pozvana kada klijent HTTP GET metodom uputi zahtev za URL api/countries. Za dobavljanje liste država koristi se objekat *countryService* koji je injektovan u prikazanu klasu *CountryController*. Listu objekata klase *Country* je potrebno pretvoriti u tekst koji će biti poslat klijentu. Ovo nazivamo **serijalizacija** objekata.

Analizirajmo kako bismo mogli napisati kod koji treba da klijentu obezbedi neke od države iz spiska, umesto vraćanje spiska svih država kao u prethodnom kodu. Klijent bi svakako morao u zahtevu da naznači kriterijum koje države želi da dobije. Ukoliko bi želeo da dobije tačno jednu državu, standardan način je da se specificira identifikator tražene države u URL-u zahteva. Na primer, ako bi klijent tražio državu sa identifikatorom 5, zahtev bi bio http://192.168.0.2:8080/api/countries/5. Vidimo da URL sada sadrži i vrednosti koje se dinamički menjaju u zavisnosti od toga koja država nam je potrebna. Ovakav tip zahteva moguće je obraditi na serveru korišćenjem promenljivih u putanji (eng. *path variable*). Potrebno je naznačiti u metodi da URL može da sadrži promenljivu i obezbediti da je ova promenljiva dostupna u metodi, obzirom da vrednost promenljive utiče na to koji će odgovor biti vraćen. Pogledajmo primer metode koja omogućuje pronalaženje države po identifikatoru.

```
@RequestMapping(value="api/countries/{id}", method = RequestMethod.GET)
public String getCountry(@PathVariable int id) {
	Country country = countryService.findOne(id);
	return country.toString();
}
```

Vidimo da se *path variable* u URL mapiranju definiše u okviru vitičastih zagrada. Takođe, *path variable* je dostupna u metodi kroz parametar, koji mora biti anotiran anotacijom @PathVariable. Spring će automatski vrednost iz URL-a zahteva upisati u ovaj parametar. Kada je poznat identifikator države, moguće je dobaviti tu državu i vratiti serijalizovan odgovor klijentu.

Drugi način prosleđivanja parametara od klijenta ka serveru je korišćenjem parametara HTML zahteva. Parametri se navode u okviru URL-a u formi parova {naziv, vrednost} u sledećoj sintaksi:

http://192.168.0.2:8080/api/countries?param1=value1&param2=value2&param3=value3

Na primer, ako bismo želeli da dobijemo sve države koje u nazivu sadrže reč Srb, klijent bi putem HTTP GET metode uputio zahtev http://192.168.0.2:8080/api/countries?name=Srb

Slično kao kod korišćenja *path variable*, i parametri HTTP zahteva mogu biti automatski preuzeti na serveru. Sledeći primer prikazuje metodu Spring kontrolera za preuzimanje spiska država koje u nazivu sadrže traženu reč.

```
@RequestMapping(value="api/countries/search", method = RequestMethod.GET)
public String getCountriesByName(@RequestParam String name) {
	List<Country> countries = countryService.findByName(name);

	StringBuffer sb = new StringBuffer();
	for (Country c: countries) {
		sb.append(c).append("\n");
	}
	return sb.toString();
}
```

Vidimo da se parametar zahteva automatski preuzima putem anotacije @RequestParam. U telu metode taj parametar može biti iskorišćen za konstruisanje odgovora. U prikazanom primeru, servis za upravljanje podacima će dobaviti države koje sadrže reč definisanu u promenljivoj *name*.