Računarski fakultet u Beogradu

Seminarski rad iz predmeta

Arhitektura softvera

Tema: Mikroservisna arhitektura korišćenjem Spring Boot tehnologije

Mentor:

Prof.dr.Branko Perišić

Student:

Nemanja Zirojević

Broj indeksa: M1

Beograd, 2017

Sadržaj

1. UVOD

Jedna od ključnih stvari pre samog procesa kodiranja softvera je definisanje arhitekture softvera, Nedostatak softverske arhitekture može otežati razvoj i onih manjih softvera. Po pravilu, takav softver je usko povezan među svojim komponentama,što u značajnoj meri otežava njegovo testiranje i buduće modifikacije. Veoma je bitno odabrati pravu arhitekturu s obzirom na vrstu softvera koji razvijamo. Poznavanje arhitekture softvera nam olakšava kodiranje, pogotovo novim razvojnim programerima koji se tek uključuju u projekt ili onima koji rade na njegovom održavanju i nadogradnji. Temelj svake arhitekture, tačnije rečeno svakog softvera bi trebali biti S.O.L.I.D. principi, tj. skup od pet principa (Single responsibility, Open/closed, Liskov substitution, Interface segregation, Dependency inversion) kojih, ukoliko pridržavamo, olakšavaju i razvoj i održavanje softvera.

U ovom radu dajemo kratak osvrt na monolitnu arhitekturu, arhitekturu koja je najčešće korištena među programerima zbog jednostavnosti implementacije i dobrog odabira onda kada ne znamo koju drugu arhitekturu odabrati. Glavni akcenat u radu, stavljamo na mikroservisnu arhitekturu, analizirajući njene prednosti i mane, najčešće slučajeve u kojima se koristi,kao i kratak pregled sličnosti i razlika između monolitne i servisne arhitekture sa jedne strane, i mikroservisne arhitekture arhitekture,sa druge .

U poslednjim poglavljima ovog rada,dajemo pregled nekih od alata za podršku izradi mikroservisa, sa posebnim osvrtom na Spring Boot tehnologiju.

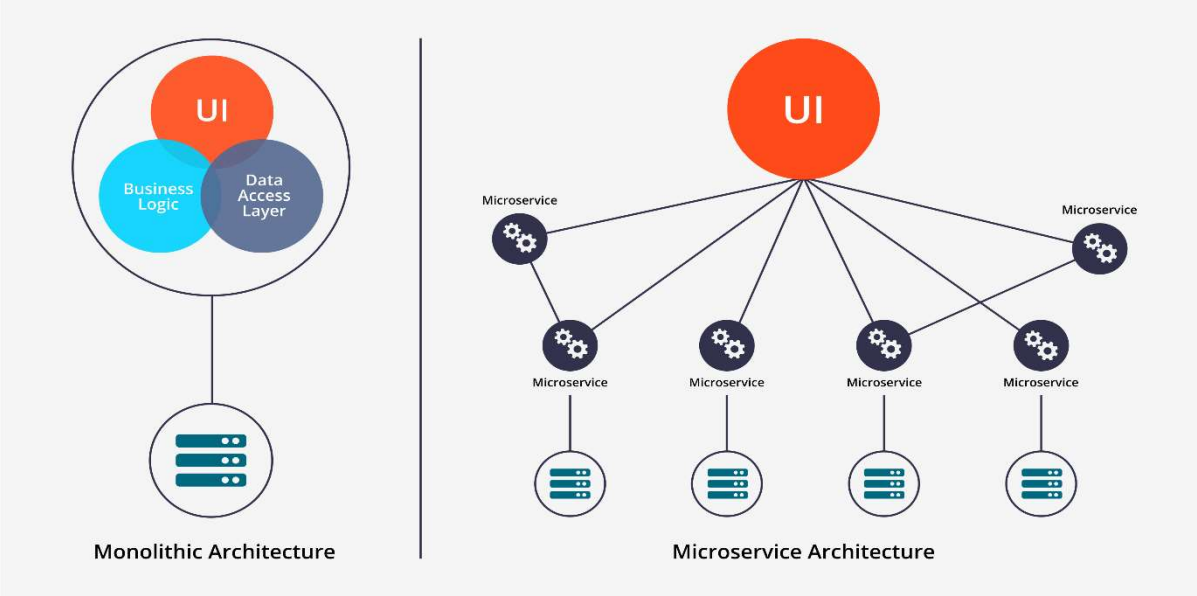
1.1 ŠTA JE MIKROSERVISNA ARHITEKTURA?

"Mikroservisna arhitektura je princip razvoja aplikacija u obliku malih, izdvojenih servisa, pri čemu svaki servis ima svoj proces i ostvaruje komunikaciju putem jednostavnih mehanizama kao što je HTTP API"

Martin Flower,ThoughtWorks

Mikroservisni pristup arhitekture softvera svaku funkcionalnost poslovne logike softvera razdvaja u različite procese, manje softvere, umesto tradicionalnog načina arhitekture gde je ceo softver jedan proces. Ova arhitektura pruža mogućnost vrlo fleksibilnog skaliranja softvera jer možemo različite delove, tj. mikroservise postaviti na različite računare (servere).

Ova vrste arhitekture je distribuirana arhitektura što znači da su svi mikroservisi potpuno razdvojeni jedan od drugih te im se pristupa preko nekog protokola, najčešće REST. Na svaki mikroservis možemo gledati kao na jednu komponentu softvera koju je moguće nezavisno zameniti ili nadograditi.

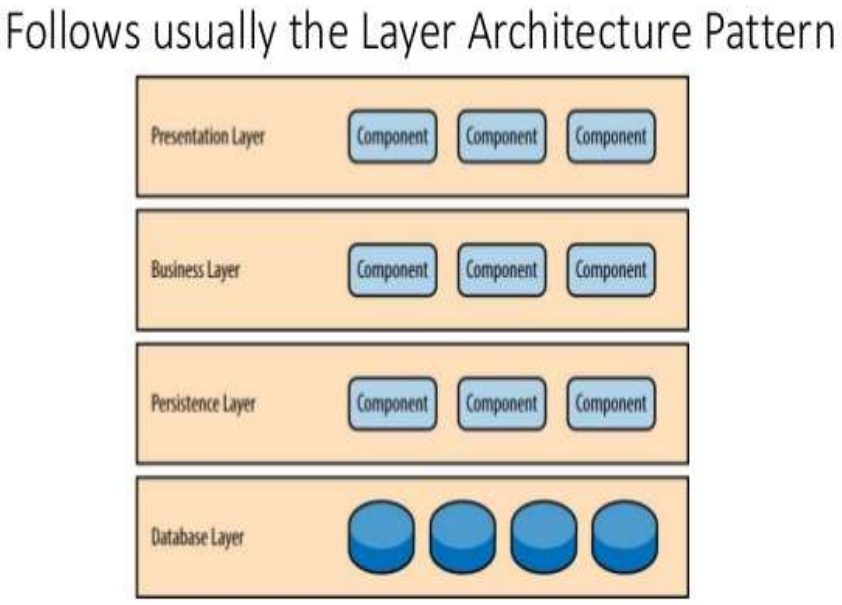


Kako je celi softver ogranizovan oko nezavisnih komponenti, a ne oko tehnologija (UI, server, baza podataka) na svakom mikroservisu mogu raditi različiti timovi koji se fokusiraju samo na određenu funkcionalnost za koju je zadužen mikroservis.

U nastavku ovog povglavlja analiziraćemo razlike između mikroservisne arhitekture i monolitne softverske arhitekture,kao i razlike između mikroservisne i servisno orijentisane arhitekture (SOA)

1.2 RAZLIKE IZMEĐU MONOLITNE I MIKROSERVISNE ARHITEKTURE

Najpoznatija arhitektura organiziranje koda je slojevita arhitektura, poznata još kao i n-slojevita arhitektura. Komponente softvera organizovane su u horizontalne slojeve, od kojih svaki obavlja neki specifični zadatak unutar softvera. Slojevita arhitektura nigde eksplicitno ne određuje koliko slojeva mora sadržavati. Međutim, većina softvera je organizovana u četiri sloja: prezentacijski sloj, sloj poslovne logike, perzistentni sloj i sloj baze podataka. Ponekad su sloj poslovne logike i perzistentni sloj objedinjeni u jedan sloj. Što je manji softver, moguće je imati samo tri sloja, dok veliki softveri mogu imati pet ili više slojeva. Svaki sloj ima neki zadatak koji obavlja. Na primer, prezentacijski sloj je zadužen za posluživanje korisničkog interfejsa korisnicima ili komunikaciju s web browser-om preko REST API ruta, dok je sloj poslovne logike zadužen za obavljanje specifičnih procesa manipulisanja s objektima ovisno o poslovnoj prirodi softvera. Svaki sloj je apstrakcija s obzirom na zadatak koji obavlja. Na primer, prezentacijski sloj ne mora znati kako i odakle se podaci dohvataju ili kako se manipuliše podacima. Isto tako, sloj poslovne logike ne mora znati kako se podaci šalju korisniku ili odakle dolaze podaci. On je samo zadužen na kontrolisanje kako se primjenjuje poslovna logika na podatke koje dobije ili koje treba proslediti prezentacijskom sloju.



Slika 1.2.1 Šematski prikaz slojevite arhitekture

Svaki sloj slojevite arhitekture je zatvoren, što znači da zahtev na putu mora proći redom svakim slojem od prezentacijskog sloja sve do sloja baze podataka. Iako bi direktan pristup bazi već u prezentacijskom sloju bio brži, ovakav način ima svojih prednosti. Ključ je u konceptu nazvanom izolacioni slojevi (*eng. layers of isolation*). Koncept izolacijskih slojeva znači da promene u jednom sloju ne bi trebale uticati na ostale slojeve. Promena je izolovana unutar sloja komponente. Dalje, koncept izolovanih slojeva takođe znači da je svaki sloj nezavistan od drugih slojeva, te da ne bi trebao znati ništa od logici i arhitekturi ostalih slojeva. Na slici iznad je dat primer kako zahtev putuje komponentama od prezentacijskog sloja do baze. Slojevita arhitektura je solidan koncept arhitekture opšte namene, što je čini pogodnom za početak kodiranja većine aplikacija, posebno kada nismo sigurni koji koncept je najbolji za našu aplikaciju. Prva stvar na koju trebamo obratiti pažnju su situacije kada zahtev samo prolazi slojevima bez primene neke logike na njima. Svaki softver koji je izgrađen na slojevitoj arhitekturi imaće ovakve situacije. Bitno je da je omjer između zahteva koje ne zahvata ovaj problem i onih koje zahvata bude 80:20. Ukoliko nam se dogodi da naš softver ima puno više zahteva koji samo prolaze slojevima možemo razmisliti da otvorimo neke delove softvera i na neki način razbijemo neke slojeve. Ovaj problem naziva se arhitecture sinkhole anti-pattern.
Dakle,imajući u vidu gore navedene karakteristike monolitne (slojevite) i mikroservisne arhitekture,dolazimo do sledeće tabele:

MONOLITNA ARHITEKTURA	**MIKROSERVISNA ARHITEKTURA**
- Centralizvano upravljanje podacima	- Decentralizovano upravljanje podacima
- Logika obrade zahteva se izvršava u jednom procesu,razdeljenom i organizovanom u klase,metode i pakete.	- Aplikacija je razdeljena na komponente,manje,nezavisne servisne aplikacije,koje se grade nad poslovnim celinama i imaju specifične zadatke prilikom obrade zahteva
Skaliranje se provodi skaliranjem cele aplikacije	Skaliranje se provodi skaliranjem delova koji zahtevaju više resursa
Promene zahtevaju dobru koordinaciju i planiranje	Promene se vrše nad komponentama, po timovima koji su zaduženi za njihov razvoj
Povećani troškovi izmena	Smanjeni troškovi izmena

Tabela 1.2.2 Poređenje mikroservisne i monolitne arhitekture

1.3 RAZLIKE IZMEĐU SERVISNE (SOA) I MIKROSERVISNE ARHITEKTURE

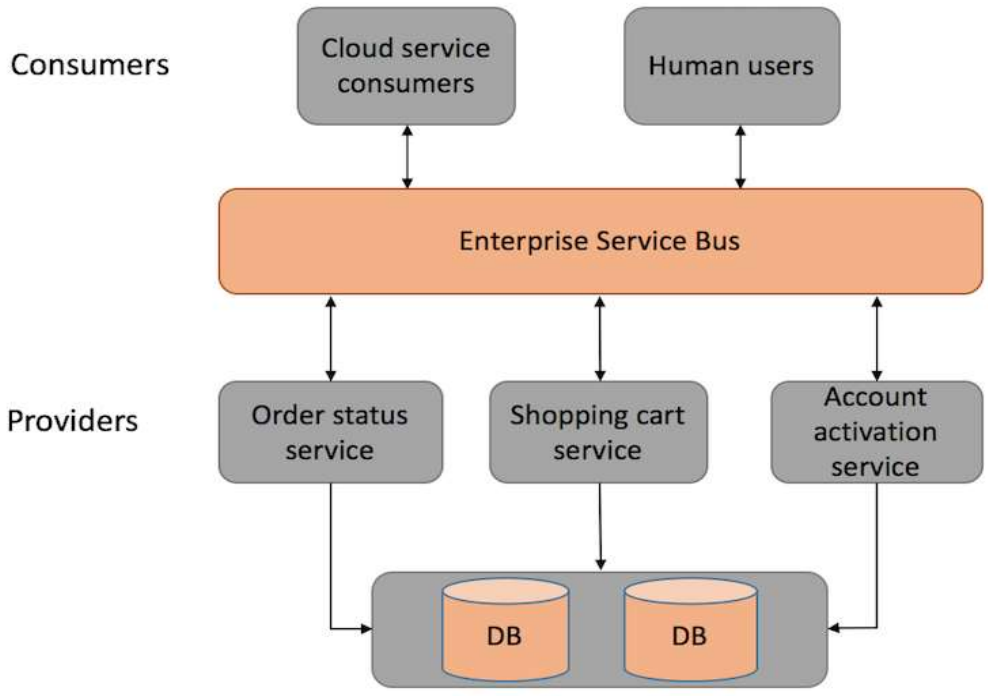
SOA je obrazac arhitekture softvera u kome komponente aplikacije pružaju usluge drugim komponentama preko komunikacionih protokola kroz mrežu. Komunikacija može uključiti jednostavno slanje podataka, ili može uključivati dva ili više servisa koji koordinišu međusobno povezivanje servisa.

Servisi (kao što je RESTful) obavljaju neke male funkcije (kao što je potvrđivanje narudžbe, aktiviranje računa i sl.)

Postoje dve glavne uloge u SOA :

- Provajder servisa
- Potrošač servisa

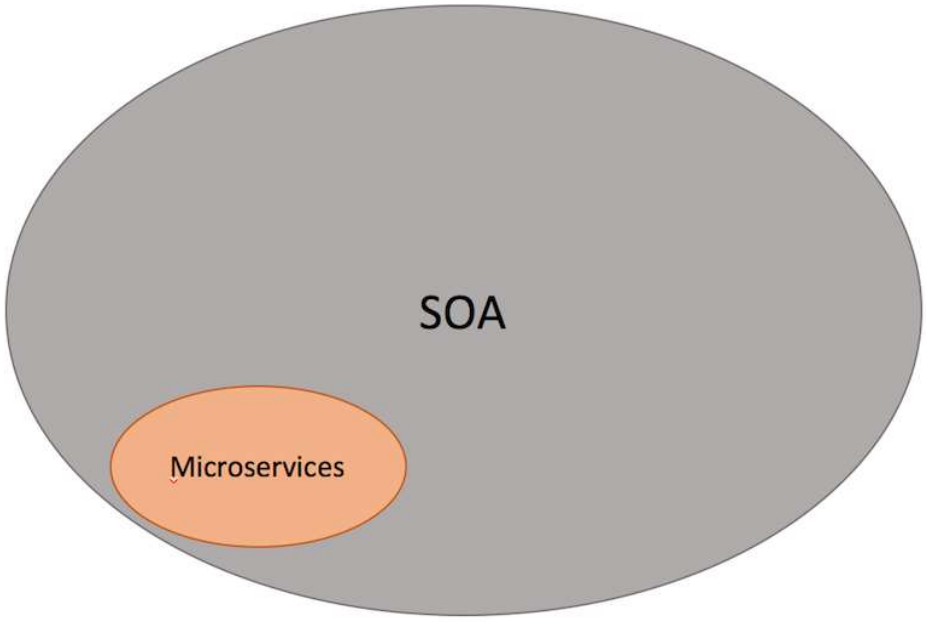
Sloj provajdera sastoji se od svih servisa definisanih u SOA, dok je sloj potrošača tačka u kojoj su potošači (ljudski korisnici, druge usluge ili treća lica) u interakciji sa SOA.



Slika 1.3.1 primer izgleda SOA arhitekture

U nastavku dajemo kratak pregled razlika između SOA arhitekture i mikroservise arhitekture.

- U obe arhitekure, svaki servis ima određenu odgovornost ;
- Razvoj servisa može biti organizovan u više timova, ali svaki tim mora poznavati mehanizam komunikacije u SOA ;
- U mikroservisima, servis može raditi i biti razvijen nezavisno od ostalih servisa, za razliku od SOA ;
- Lakše je često postavljanje nove verzije servisa,te skaliranje servisa neovisno od drugih
- U SOA, Enterprise Service Bus (ESB) može postati mesto kvara koje će uticati na celu aplikaciju;
- Budući da svaki servis komunicira kroz ESB, ako se jedan servis uspori, može se desiti da se ESB zaguši zahtevima za taj servis;
- S druge strane,mikroservisi su mnogo bolji u tolerisanju greške;
- U SOA,servisi dele skladišta podataka,dok u mikroservisima svaki servis može imati nezavisno skladište podataka;
- Glavna razlika između SOA i mikroservisa leži u obimu



Slika 1.3.2 Odnos mikrosevisa i SOA

- Mikroservis mora biti znatno manji od SOA;
- S druge strane,SOA može biti monolit ili može biti sastavljena od više mikroservisa;
- Važno je da je SOA dizajnirana i implementirana sa različitim stilovima, obično zbog fokusa na ESB koji se koristi za integraciju monolitnih aplikacija;

2. PREDNOSTI I MANE MIKROSERVISA

U ovom poglavlju rada sumiraćemo do sad navedene karakteristike mikrosevisa, dajući tabelarni prikaz prednosti i mana njihovog korišćenja.

PREDNOSTI MIKROSERVISA
Komponente se mogu nezavisno isporučivati i automatski postavljati na servere
Mikrosevisna arhitektura dobro funkcioniše uz Continuous integration i Continuous delivery
Moguće je često izdvajanje i ažuriranje verzija komponenti uz održavanje stabilnosti ostatka sistema
Razvojni timovi mogu razvijati i isporučivati komponente mikroservisa relativno nezavisno jedni o drugima
Komponente mogu biti implementirane sa različitim tehnologijama i pisane u različitim programskim jezicima
Različiti timovi mogu razvijati različite komponente
Moguće je korišćenje različitih baza podataka
Centralizovano upravljanje je minimalno

Tabela 2.1 Prednosti mikroservisa

MANE MIKROSERVISA
Složenost distribuiranih sistema
Složenost međuservisne komunikacije
Nepostojanje distribuiranih transakcija
Složenost operativih procesa
Otežano testiranje zbog interakcija komponenti
Potreba za robusnim upravljanjem grešaka i automatskim oporavkom
Potreba za sofisticiranim nadgledanjem rada

Tabela 2.2. Mane mikroservisa

3. IZAZOVI KORIŠĆENJA MIKROSERVISA

Porast popularnosti upotrebe mikroservisa, kao sredstva brže isporuke i razvoja softvera,poslednjih godina je sve više u porastu.Mirkoservisna arhitektura, dosta pomaže, ali ipak dolazi sa svojim izazovima.Jinesh Parekh (Idyllic software CEO) izdvaja četiri najveća izazova sa kojima se kompanije suočavaju,kada odluče da zakorače u svet mikroservisa :

1. Nedovoljno uputstava

Kada dodje do potrebe za implementacijom novih tehnologija i prihvatanja novih strategija razvoja,to u početku može dosta da podseća na "Divlji zapad".
Postoji vrlo malo striktnih pravila i smernica ili dokumentacije koje može da pomogne u izradi i deploy-u softvera korišćenjem mikroservisa. Srećom, sajtovi kao što su DevOps.com i ContainerJournal pružaju koliko-toliko smernica organizacijama prilikom prelaska na mikroservisnu arhitekturu.

2. Greške

Postoji niz "pokretnih delova" uključenih u uspešnu implementaciju mikroservisa Svaka kompanija,koja planira da implementira mikroservisnu arhitekturu na svome softveru,trebalo bi da usvoji usvojiti kulturu DevOps i da ima odgovarajuću infrastrukturu kako bi se izbjegla pogrešna implementacija. Arhitektura mikroservisa obično je izgrađena na temelju cloud-a i virtuelizacije i oslanja se na automatizaciju kako bi osigurala uspeh i izbjegla neuspjehe.

3. Nema povratka nazad

Trebali bi biti svjesni da je izuzetno teško vratiti se kada se započne sa prelaskom na mikroservise. Promjene koje se moraju dogoditi u smislu kulture, alata i procesa za prihvatanje mikro servisa se ne mogu lako poništiti. Dobra vest je da je vrlo malo – (ako uopše I postoje) razloga,da se kompanija koja je prešla na DevOps i kontejnere kao razvojni okvir,odluči da se vrati na stari, monolitni način obavljanja stvari.

4. Isporučivanje vredosti

Parekhov fokus na isporučivanju vrednosti je zasnovan na iskustvima Idyllic Software kompanije. Njegova kompanija se bavi isporučivanjem aplikacija za klijente preko mikroservisa, a Parekh sugeriše da kompanije trebaju da prenose razvoj na iskusnog lidera kako bi osigurali da mikroservisi ispunjavaju očekivanja. Međutim, ne morate obavezno da outsource-ujete Idyllic Software, ili da uopšte ne koristite outsource.Samo treba da budete sigurni da je vaš pristup mikroserviserima pogodan za vašu kompaniju

4. ALATI ZA PODRŠKU IZRADI MIKROSERVISA

U ovom poglavlju bavićemo se Spring Boot i Dropwizard razvojnim okvirima za ubrzanu izgradnju mikroservisnik aplikacija, kroz analizu njihovih karakteristika, sličnosti i razlika.

4.1 Spring Boot

Zajedno sa izlaskom verzije 4.0 Spring frameworka i novi projekat je izašao na videlo-Spring Boot. Glavna svrha Spring Boot-a je da obezbedi JAVA web aplikacijama brzo i jednostavno pokretanje,kroz ugrađenu verziju Tom Cat-a (podržava i Jetty iUndertow), tako da u startu eliminiše potrebu za JAVA EE kontejnerima.

Uz pomoć Spring Boot-a,možemo izložiti komponente (mikroservise) kao RESTful servise (kroz standardnu podršku iz Spring frameworka) ,nezavisno, onako kako je predviđeno u mikroservisnoj arhitekturi, tako da prilikom njihove nadogradnje ne moramo raditi re-deploy cele aplikacije,nego samo te komponente. Spring Boot razvojni okvir ima podršku nadgledanje,logovanje i proveravanje aplikacija, kao i podršku za korišćenje standardnih biblioteka.

Pored toga, Spring Boot sadrži poseban modul za testiranje koji uključuje Junit.

4.2 Dropwizard

Glavna razlika između Spring Boot-a i Dropwizard kontejnera je podrška za dependency injection.

Spring ima ugrađenu podršku, dok Dropwizard ostavlja mogućnost integracije sa okvirom po izboru. Pored toga,Dropwizard podržava Jetty,i implementira REST preko Jersy-a dok za logovanje koristi Logback biblioteku.

U nastavku, dajemo tabelarni prikaz sličnosti i razlika između Spring Boota i Dropwizard-a.

Slika 4.1.1 Razlike između Spring Boot-a I Dropwizarda

5. PRIMER RAZVOJA MIKROSERVISNE APLIKACIJE KORIŠĆENJEM TEHNOLOGIJE SPRING BOOT

U cilju ilustracije koncepta mikroservisa, kreiraćemo tri Maven projekta, pri čemu će svaki od njih predstavljati back-end funkcionalnost,(API), dok će jedan od njih pozivati ostala dva.

U Eclipse okruženju,kreirajmo tri jednostavna Maven projekta, bez definisanog archetype-a i nazovimo ih Product-backend, Customer-backend i Order-backend. U .pom fajlovima ova tri projekta dodaćemo neophodne dependency-e za kreiranje REST servisa i pokretanje Spring Boot-a.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.2.0.RELEASE</version>
</parent>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jersey</artifactId>
</dependency>
</dependencies>
```

Sa unetim dependency-ma,može početi da pišemo kod.

Prva klasa koju ćemo kreirati (ukoliko koristimo Spring tools suite razvojno okruženje prilikom kreiranje Spring Boot projekta ova klasa se automatski generiše), a koja će biti ista u sva tri projekta je klasa Application, čija je svrha da označi projekat kao Spring Boot projekat,tj.da služi kao inicijator za Spring Boot pa je zato i označena sa @SpringBootApplication anotacijom.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
public static void main(String[] args) {
SpringApplication.run(Aplication.class, args);
}
}
```

Sledeća klasa koju ćemo kreirati je ApplicationConfig klasa. U ovoj klasi, koju označavamo sa *@Configuration* springovom anotacijom, da bi je Spring framework tretirao kao klasu odgovornu za konfiguraciju resursa projekta, podešavamo Jersy koji je naš ResourceManager odgovoran za izlaganje REST servisa klijentima.

U aplikacijama koje se koriste u produkciji, ova klasa bi takođe kreirala resurse potrebne za pristup bazi,ali da bi zadržali jednostavnost i pažnju usmerili na Spring Boot, ovde ćemo sami kreirati instance objekata koje ćemo potom stavljati u liste,iz kojih ćemo ih kasnije izvlačiti na klijentskoj strani.

```
import javax.inject.Named;
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ApplicationConfig {
@Named
static class JerseyConfig extends ResourceConfig {
public JerseyConfig() {
this.packages("com.nemanja.rest");
}
}
}
```

Gore napisana klasa će se biti identična u projektima Customer-backend i Product-backend, s tim da ćemo u projektu Order-backend ,pošto će nam ta klasa pozivati predhodna dva servisa,dodati jos objekat tipa RestTemplate (slika niže). Klasa RestTemplate je jedna novina u Spring framework verziji 4.0 koja nam pruža standardizovan i veoma jednostavan interfejs za korišćenje REST servisa.

```
import javax.inject.Named;
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
@Configuration
public class ApplicationConfig {
@Named
static class JerseyConfig extends ResourceConfig {
public JerseyConfig() {
this.packages("rs.nemanja.rest");
}
}
@Bean
public RestTemplate restTemplate() {
RestTemplate restTemplate = new RestTemplate();
return restTemplate;
}
}
```

Najzad, počnimo sa implementacijom samih REST servisa. U projektu odgovornom za rukovanje zahtevima vezanim za ažuriranjem kupaca (registracija novih korisnika, pretraga korisnika, nalaženje korisnika na osnovu id-a) napravićemo jedno POJO klasu Customer, a zatim i REST servis.

```
public class Customer {
private long id;
private String name;
private String email;
public long getId() {
return id;
}
public void setId(long id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
}
```

REST servis će, radi jednostavnosti, imati tri funkcionalnosti :

1. Nalaženje korisnika na osnovu njegovog id-a
2. Vraćanje liste svih korisnika
3. Registraciju novih korisnika

```
4. import java.util.ArrayList;
5. import java.util.List;
6. import javax.inject.Named;
7. import javax.ws.rs.GET;
8. import javax.ws.rs.Path;
9. import javax.ws.rs.Produces;
   import javax.ws.rs.QueryParam;
10.  import javax.ws.rs.core.MediaType;
11.  @Named
12.  @Path("/")
13.  public class CustomerRest {
14.  private static id=0;
15.  private static List<Customer> Customers = new ArrayList<Customer>();
16.  @GET
17.  @Produces(MediaType.APPLICATION_JSON)
18.  public List<Customer> getCustomers() {
19.  return customers;
20.  }
21.  @GET
22.  @Path("customer")
23.  @Produces(MediaType.APPLICATION_JSON)
24.  public Customer getCustomer(@QueryParam("id") long id) {
25.  Customer cli = null;
26.  for (Customer c : customers) {
27.  if (c.getId() == id)
28.  cli = c;
29.  }
30.  return cli;
31.  }
32.  }
33.
34.  @Post
35.  @Path("add/{name}/{email}")
36.  @Produces(MediaType.APPLICATION_JSON)
37.  public Customer addCustomer(@PathVariable("name") String customerName,
38.  @PathVariable("email")String customerEmail)
39.   {
40.  id++;
41.  Customer newCustomer=new Customer();
42.  newCustomer.setName(customerName);
43.  newCustomer.setEmail(customerEmail);
44.  newCustomer.setId(id);
45.  Customers.add(newCustomer);
46.
47.  return newCustomer;
48.  }
```

Ovim je naš REST servis za korisnike završen. Analogno, za proizvode, kao i za korisnike, definisaćemo metode kojima se vrši pretraga svih proizvoda, vraća proizvod na osnovu njegovog id-a i dodaje novi proizvod. Na kraju, kreiraćemo servis za narudžbe, u kome ćemo definisati metod submitOrder, koji kao parameter uzimati id kupca i id proizvoda, na osnovu čega će se formirati narudžba. Za početak, kreirajmo dve POJO klase : Product.java, Order.java (u projektu vezanom za manipulaciju proizvodima, i u projektu vezanom za manipulaciju narudžbama, po jednu, respektivno).

```
public class Product {
private long id;
private String name;
private String price;
private String description;

public long getId() {
return id;
}
public void setId(long id) {
this.id = id;
}
public String getName() {
return name;
}
public String getPrice() {
return price;
}

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

public String getDescription() {
return description;
}
public void setDescription(String description) {
this.description = description;
}
}
```

```
import java.util.Date;
public class Order {
private long id;
private long amount;
private Date dateOrder;
private Customer customer;
private Product product;
public long getId() {
return id;
}
public void setId(long id) {
this.id = id;
}
public long getAmount() {
return amount;
}
public void setAmount(long amount) {
this.amount = amount;}
public Date getDateOrder() {
return dateOrder;
}
public void setDateOrder(Date dateOrder) {
```

```
this.dateOrder = dateOrder;
}
public Customer getCustomer() {
return customer;
}
public void setCustomer(Customer customer) {
this.customer = customer;
}
public Product getProduct() {
return product;
}
public void setProduct(Product product) {
this.product = product;
}
}
```

Nakon toga, u skladu sa gore navedenim funkcionalnostima, napravimo po jedan REST servis u svakom od dva, gore navedena projekta.

ProductRest.java (unutar projekta za manipulaciju proizvodima, zajedno sa Product.java POJO klasom) :

```
import java.util.ArrayList;
import java.util.List;
import javax.inject.Named;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
@Named
@Path("/")
public class ProductRest {
private static id=0;
private static List<Product> products = new ArrayList<Product>();
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Product> getProducts() {
return products;
}
@GET
@Path("product")
@Produces(MediaType.APPLICATION_JSON)
public Product getProduct(@QueryParam("id") long id) {
Product prod = null;
for (Product p : products) {
if (p.getId() == id)
prod = p;
}
return prod;
}

@POST
@Path("add/{name}/{description}/{price}")
@Produces (MediaType.APPLICATION_JSON)
```

```
public Product addNewProduct(@PathVariable (“name”) String name,@PathVariable
(“description”) String description,@PathVariable(“price”)String price )
{
id++;
product product=new Product();
product.setName(name);
product.setId(id);
product.setDescription(description);
product.setPrice(price);
products.add(product);
return product;
}
```

OrderRest.java (unutar projekta za manipulaciju narudžbama, zajedno sa Product.java POJO klasom)

```
import java.util.Date;
import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import org.springframework.web.client.RestTemplate;
@Named
@Path(“/”)
public class OrderRest {
private long id = 1;
@Inject
private RestTemplate restTemplate;
@GET
@Path(“order”)
@Produces(MediaType.APPLICATION_JSON)
public Order submitOrder(@QueryParam(“idCustomer”) long idCustomer,
@QueryParam(“idProduct”) long idProduct,
@QueryParam(“amount”) long amount) {
Order order = new Order();
Customer customer = restTemplate.getForObject(
“http://localhost:8081/customer?id={id}”, Customer.class,
idCustomer);
Product product = restTemplate.getForObject(
“http://localhost:8082/product?id={id}”, Product.class,
idProduct);
order.setCustomer(customer);
order.setProduct(product);
order.setId(id);
order.setAmount(amount);
order.setDataOrder(new Date());
id++;
return order;
}
}
```

Primetimo, da se Product.java i Customer.java klase koriste u okviru order servisa. Te klase, međutim, nisu direktne refernce na klase implementirane u ranije kreiranim projektima, već klase ''klonirane" iz originala,u okviru order projekta,

Da bi testrali kreirane REST servisa, za početak je potrebno da u Eclipse okruženju konfigurišemo portove na kojima će biti pokrenuti servisi. U tu svrhu, kreiraćemo konfiguracione fajlove (u okviru svakog od tri projekta po jedan), gde navodimo port koji će spring boot koristiti za pokretanje servisa. Ilustracije radi, neka to budu portovi 8081,8082,8083 za customer servis,product servis i order servis, respektivno.

Naredba za specifikaciju porta u Eclipse okruženju :

```
-Dserver.port=8081
```

Kada startujemo sva tri servisa, primera radi, pozovimo product servis za unošenje novog proizvoda, unošenjem sledećeg URL-a u browser :

http://localhost:8082/product/SamsungS4/32gb/450e

Kao odgovor dobijamo sledeći JSON string:

```
{"id":1,"name":SamsungS4,"description":32GB}
```

6. ZAKLJUČAK

Uzimajući u obzir jednostavnu implementaciju, kao i konfigurabilnost, uz snažne mogućnosti, Spring Boot se nameće kao dobra opcija za implementaciju mikroservisne arhitekture.