Prvo predavanje

Uvod

Poslovne se aplikacije uobičajeno razvijaju troslojnom arhitekturom:

1. **prezentacijski sloj** (tu se dešava interakcija s korisnikom, recimo kroz web stranicu ili mobilnu aplikaciju)
2. **servisni sloj** ili **sloj poslovne logike** (tu je sva "pamet" aplikacije)
3. **podatkovni sloj** (tu se obavlja spremanje i čitanje podataka, npr. iz baze podataka)

U ovom predavanju cilj nam je postaviti osnovnu arhitekturu aplikacije. Po završetku, biti ćemo u stanju pristupiti svojoj aplikaciji kroz browser i dobiti (čak i JSON!) odgovor.

Zašto bi radili troslojno?

Osim generalnih prednosti modularnog pristupa razvoju softvera, ovakva separacija nam omogućava da softver koji smo pisali u 2010. godini preradimo u 2014. tako da cijeli prezentacijski sloj prepišemo u Angular.js tehnologiji koja tad nije postojala, bez da ostali slojevi znaju da se nešto mijenja.
Isto bi tako mogli i PostgreSQL bazu podataka, koju koristi podatkovni sloj, zamijeniti sa MySQL, bez da moramo mijenjati ostatak aplikacije. Naravno, nismo ludi napraviti takvu glupost, Postgre je puno bolji.

Zašto koristimo Spring?

Zato jer želimo raditi ozbiljne web aplikacije na Java platformi koje mogu odgovoriti na najozbiljnije zahtjeve korisnika.

Kako ćemo to postići?

Tako da ne izmišljamo kotač nanovo.

Koristiti ćemo razne Spring module koji će nam olakšati rješavanja uobičajenih izazova koji se pojavljuju u razvoju takvog tipa aplikacija, na primjer:

- Aspect-oriented programming - za elegantno riješiti logiranje ili transakcije
- Authentication and authorization - svaka ozbiljna aplikacija zahtjeva implementaciju prava pristupa, Spring ima svoj modul koji nam pomaže to implementirati
- Data access - sigurno ćemo koristiti relacijske ili NoSQL baze, trebati će nam standardni mehanizam pristupa podacima, objektno-relacijski maper i ostale zgodne stvari
- Testing - ozbiljne aplikacije zahtijevaju i ozbiljno testiranje, a Spring nam daje odličnu podršku za *unit* i integracijsko testiranje
- ⋯ *i mnoge druge super stvari*

Ukratko, Spring koristimo da sebi olakšamo život i ne radimo indijanski. Tjera nas da koristimo neke dobre prakse koje su pametni ljudi definirali nakon što su godinama upadali u iste probleme, pa ima smisla ići utabanim putem.

Razvojno okruženje i alati

Koristiti ćemo IntelliJ IDEA razvojni alat (Community Edition), Javu 7, Gradle Build Automation. Web aplikaciju vrtjeti ćemo na Tomcat application serveru, ali o tome ne moramo brinuti jer će Spring Boot to podići i konfigurirati za nas.

A što je Spring Boot

Spring Boot je napravljen kako bi nama programerima olakšao izradu web aplikacija baziranih na Springu.

Kako nam olakšava? Tako da dolazi "predkonfiguriran" na način da od starta *radi*, što inače nije slučaj kod Springa gdje je sve potrebno ručno konfigurirati (a to zna biti naporno i dugotrajno). Na primjer, dolazi s ugrađenim aplikacijskim serverom tako da ne moramo mi instalirati i konfigurirati zasebni, zna sam skinuti sve potrebne biblioteke (*dependencies*), prati status aplikacije kroz metrike i druge lijepe stvari. Convention over configuration pristup.

OK, shvaćam, a što je Gradle i za što ga koristimo?

Gradle je alat kojeg ćemo koristiti za kompajliranje (build) aplikacije i za "dependency management" (pokušao sam prevesti ali ne ide).

Evo kako izgleda naš prvi `build.gradle`:

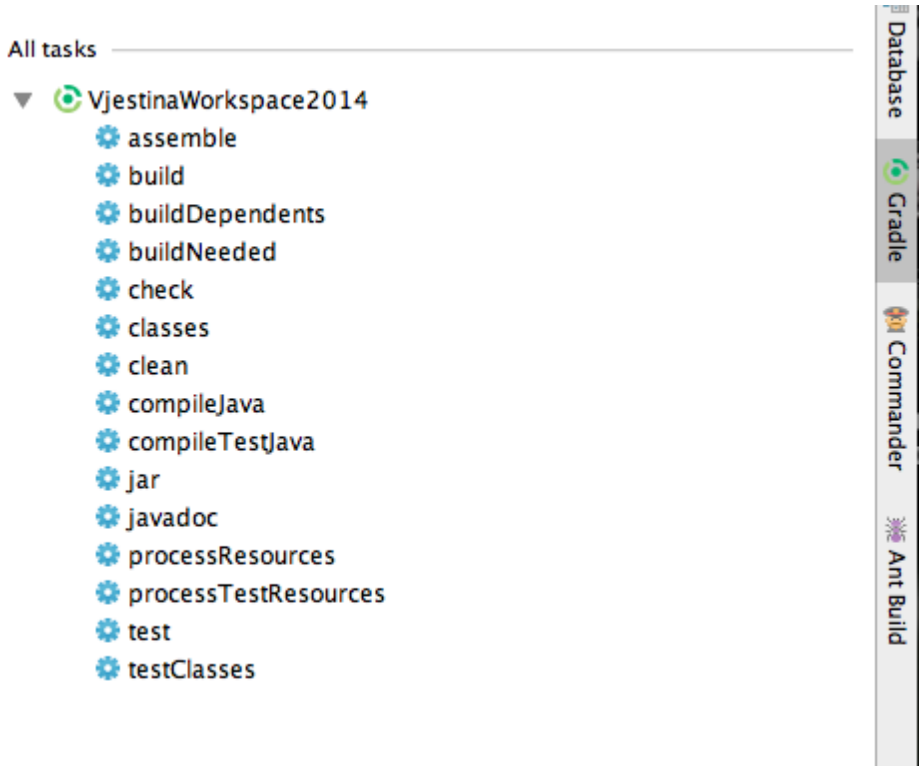
```
apply plugin: 'java'
apply plugin: 'idea'

sourceCompatibility = 1.7
version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.1.7.RELEASE")
}
```

Prvo ćemo reći Gradleu da je ovdje riječ o Java projektu i kako koristimo IntelIJ IDEA alat, kroz “apply plugin” direktive. Na taj način dobiti ćemo mogućnost kompajliranja Java aplikacija, testiranja, pakiranja u JAR/WAR, kao i datoteke potrebne za IDEA alat, koje definiraju projektnu strukturu. Svaki “plugin” dodaje određen broj tzv. zadataka (tasks), možemo vidjeti koji su to kroz Gradle meni u IDEA okruženju:

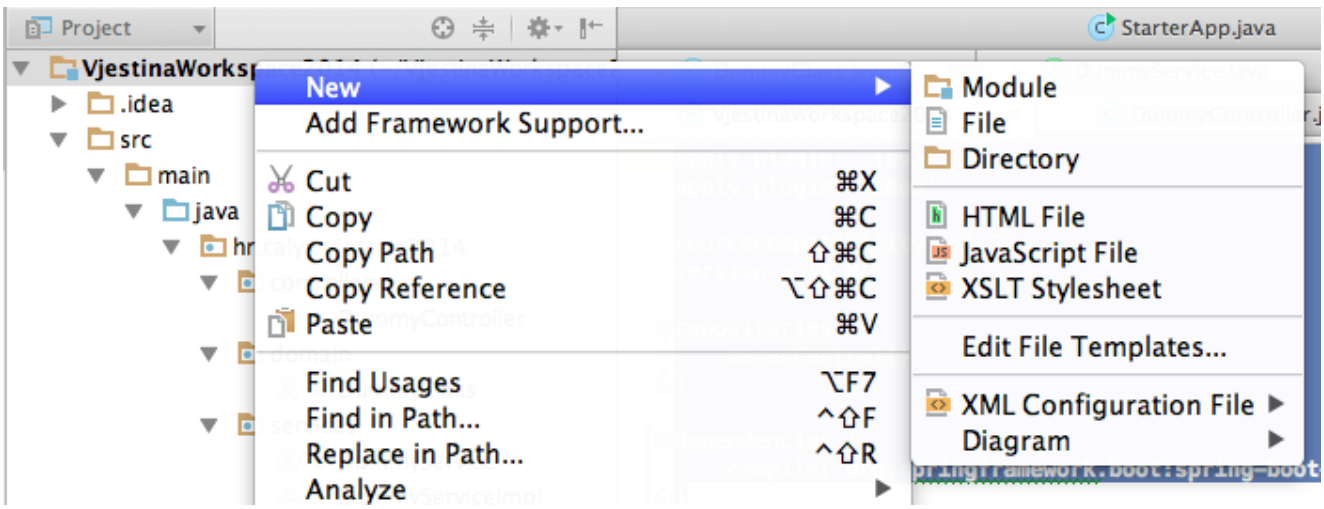


Nakon toga, definiramo verzije Gradle alata i Java platforme koje ćemo koristiti.

U dijelu "repositories" definiramo repozitorij(e) koje će Gradle koristiti za *dependency management*. Dakle, u našem slučaju, kada mu zatreba neka biblioteka, otići će na Mavenov centralni repozitorij i pokušati je pronaći, skinuti i smjestiti u projektnu strukturu.

OK, već sam umoran od bla-bla, možemo li nešto konačno pokrenuti?

Možemo! Zadnji preduvjet je da napravimo foldersku strukturu za naš programski kod. Dakle, desni klik na projekt u IDEA i odabrati akciju "New - Directory" sa slike u nastavku:



U prozoru koji se otvori napisati "src/main/java". Ponoviti postupak i upisati "src/main/resources". U "Java" direktoriju napraviti paket (New - Package), npr.

“hr.calyx.vjestina2014”. Unutar tog paketa napraviti još 3 paketa, “controllers”, “domain” i “services”.

E sad smo spremni napisati nešto egzekutabilno!

U baznom paketu (“hr.calyx.vjestina2014”) napraviti razred `StarterApp.java` koji će biti sljedećeg sadržaja:

```
package hr.calyx.vjestina2014;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;

@EnableAutoConfiguration
@ComponentScan
public class StarterApp {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(StarterApp.class, args);
    }
}
```

Ovo je dovoljno za pokrenuti Spring *i* našu aplikaciju!

Idemo objasniti što smo napravili:

1. definirali smo ulaznu točku naše aplikacije, naš *static void main*
2. razred u kojem je ulazna točka definirana anotirali smo s dvije važne anotacije koje ćemo objasniti u nastavku
3. u *main* metodi pokrenuli smo Spring

EnableAutoConfiguration anotacija dio je Spring Boot modula, a možemo ju zamisliti kao naredbu Springu da pokuša sam shvatiti što želimo i konfigurirati sve potrebno. Npr. to će značiti da podigne ugrađeni Tomcat server na adresi localhost:8080, registrirati sve komponente (zasad ih nemamo!) koje se nalaze unutar paketa i slično.

ComponentScan anotacija zadaje Springu zadatak da pretraži sve razrede unutar našeg paketa i prepozna/registrira Spring komponente. Zasad tu nemamo ništa, ali već u idućem paragrafu nam stiže prvi @Controller.

SpringApplication je razred kojeg nam daje Spring Boot modul, a služi za jednostavnu inicijalizaciju Springa.

Kako se pokreće naš StarterApp? Desni klik na datoteku, akcija **Run ‘StarterApp.main()’**.

Kad već radimo web aplikaciju, idemo nešto ispisati u browser!

Za to, morat ćemo napraviti prvi Controller. A što je to Controller?

U MVC (model-view-controller) arhitekturi, koju Spring prati, kontroler je zadužen za procesiranje korisničkog zahtjeva (u našem slučaju HTTP zahtjeva kojeg ćemo zadati kroz *browser*). Procesiranje se sastoji od samog zaprimanja zahtjeva, izgradnje *modela* podataka i proslijeđivanja istog do *View* komponente koja će podatak prikazati korisniku kao odgovor. U našem slučaju, to će biti web stranica koja prikazuje rezultat operacije.

Pa, idemo napraviti svoj prvi kontroler koji će vratiti neki statički string.

U paketu "controllers" napravite `DummyController.java` sa sljedećim sadržajem (nisam uključio importe u listing):

```
@Controller
public class DummyController {

    @RequestMapping(value="/dummy", method = RequestMethod.GET)
    @ResponseBody
    public String dummy() {
        return "Ovo je dummy String";
    }
}
```

Anotacijom @Controller dali smo Springu do znanja da je ovo Controller komponenta.

Nakon toga, definirali smo našu metodu (*dummy*) te ju anotirali sa sljedećim anotacijama:

1. RequestMapping - mapiranje na URL endpoint (/dummy, odnosno, u konačnici će to biti http://localhost:8080/dummy) i HTTP metodu GET (klasični zahtjev web browsera)
2. ResponseBody - anotacija koja naređuje Springu da string koji vraćamo upakira u tijelo HTTP poruke koje će vratiti korisniku (a koja će se, u konačnici, prikazati u korisničkom web browseru)

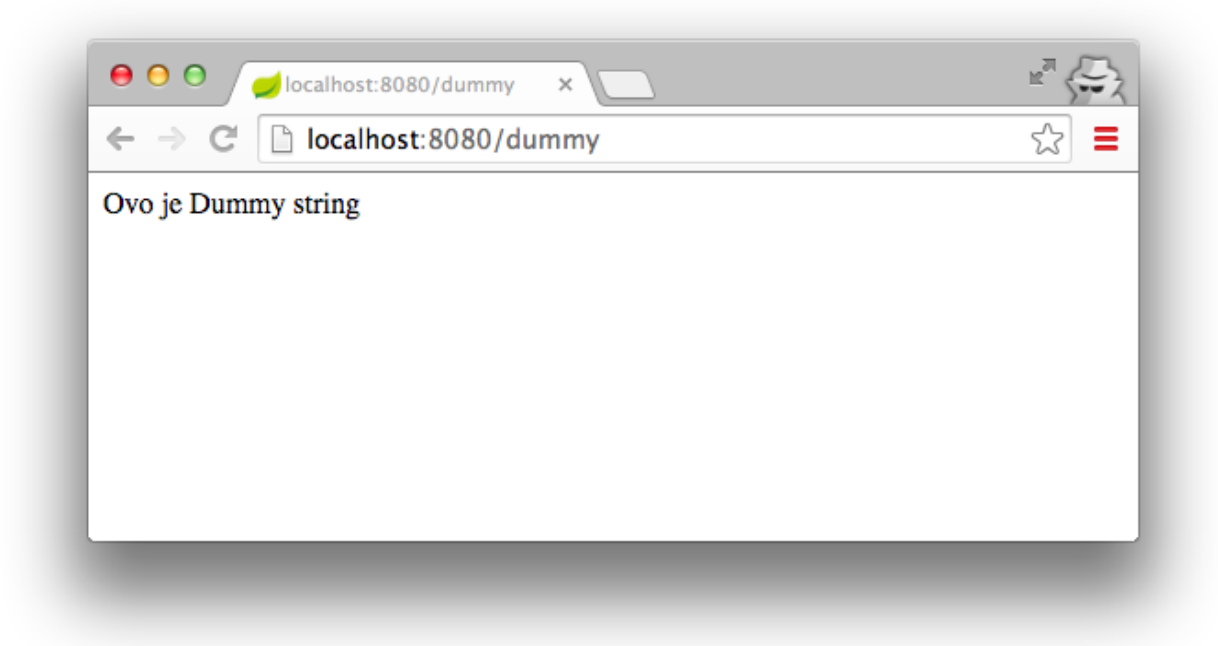
Super. Idemo sada opet pokrenuti našu aplikaciju.

U konzoli koja prikazuje poruke inicijalizacije Spring konteksta uočiti ćemo jednu novu liniju:

```
2014-10-07 09:08:18.623  INFO 16156 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/dummy],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}
" onto public java.lang.String
hr.calyx.vjestina2014.controllers.DummyController.dummy()
```

Dakle, Spring je shvatio da smo dodali kontroler i mapirali ga na endpoint "/dummy"!

Ako je to istinito, trebali bi moći kroz web browser pristupiti tom endpointu i dobiti odgovor:



Super! A kako vratiti nešto što nije statični String?

Jednostavno - tako da kontaktiramo naš prvi servis, koji ćemo upravo napraviti, i pitati ga da nam vrati nešto pametno. Kao što znamo, kontroleri nisu mjesta za pametne stvari.

Dakle, u paketu "services" idemo napraviti sučelje koje će definirati metode koje naš servis mora implementirati. Zasad nećemo ulaziti u dubino zašto radimo sučelje pa onda implementaciju, umjesto da jednostavno napravimo razred koji implementira servis, morat ćete mi vjerovati na riječ da je to dobro za vas.

Pa, napravite `DummyService.java` sa sljedećim sadržajem:

```
public interface DummyService {
    public String getDummyString();
}
```

I odmah nakon toga, u istom paketu, idemo napraviti (jednu) implementaciju tog servisa, u datoteci `DummyServiceImpl.java`:

```
@Service
public class DummyServiceImpl implements DummyService {
   @Override
   public String getDummyString() {
      return "Servis govori, ovo je Dummy String";
   }
```

```
}
```

Važno za primjetiti - @Service anotacija koju smo objesili na servis, kako bi Spring znao prepoznati i učitati komponentu.
Servis sada možemo pozvati iz našeg DummyControllera:

```
@Controller
public class DummyController {

    @Autowired
    DummyService dummyService;

    @RequestMapping(value="/dummy", method = RequestMethod.GET)
    @ResponseBody
    public String dummy() {
        return dummyService.getDummyString();
    }
}
```

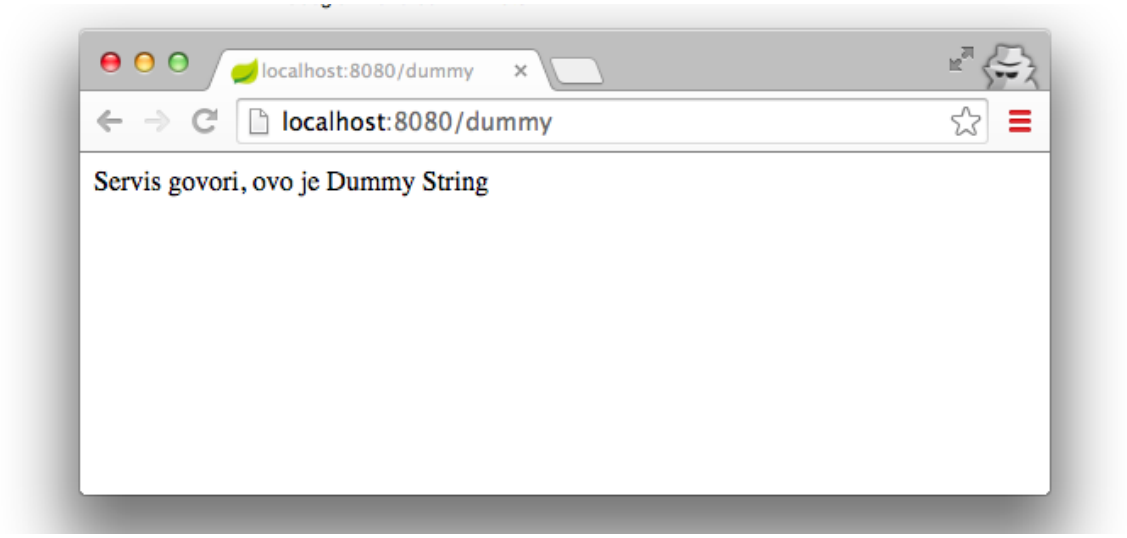
Cool stvar koju smo ovdje iskoristili zove se ***Dependency Injection.***

Koristeći Reflection API, Spring će znati prepoznati da postoji razred DummyServiceImpl koji implementira sučelje DummyService, instancirati ga i dodijeliti varijabli `dummyService`.

Možda ne izgleda na prvu, ali to je ustvari jako cool.

Taj će nam mehanizam “sutra” omogućiti da dinamički guramo različite implementacije servisa, ovisno o potrebama. Npr. kada ćemo htjeti testirati, koristiti ćemo *mock* podatke kroz posebnu implementaciju servisa koju ćemo napraviti samo za tu potrebu. Vjerujte mi, jako korisno.

OK, idemo sada pokrenuti aplikaciju i pristupiti našem Dummy kontroleru:



Sveradi!

A što ako želim vratiti cijeli svoj objekt umjesto običnog Stringa?

Joj koliko pitanja. Ovo je zadnje za danas!

U paketu "domain" idemo napraviti svoj prvi domenski razred, `DummyClass.java`:

```
public class DummyClass {
    private Long id;
    private String name;
    private int age;

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public int getAge() {
```

```
        return age;
    }

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

U sučelju koji definira servis, definirajmo novu metodu koja vraća taj objekt:

```
public interface DummyService {
    public String getDummyString();
    public DummyClass getDummyObject();
}
```

A u implementaciji servisa, implementirajmo ju:

```
@Service
public class DummyServiceImpl implements DummyService {
    @Override
    public String getDummyString() {
        return "Servis govori, ovo je Dummy String";
    }

    @Override
    public DummyClass getDummyObject() {
        DummyClass retval = new DummyClass();
        retval.setAge(12);
        retval.setId(1L);
        retval.setName("Luka Modric");
        return retval;
    }
}
```

Ostalo je još samo nadograditi kontroler sa metodom koja će vraćati naš objekt:

```
@Controller
public class DummyController {

   @Autowired
   DummyService dummyService;
```

```
    @RequestMapping(value="/dummy", method = RequestMethod.GET)
    @ResponseBody
    public String dummy() {
        return dummyService.getDummyString();
    }

    @RequestMapping(value="/dummyobject", method=RequestMethod.GET)
    public ResponseEntity<DummyClass> getDummyObject() {
        return new ResponseEntity<DummyClass>(dummyService.getDummyObject(),
HttpStatus.ACCEPTED);
    }
}
```

Prije nego pokrenemo, kratko ćemo objasniti **ResponseEntity** razred. Riječ je o pomoćnom razredu koji nam olakšava vraćanje objekata kroz HTTP odgovor. Za nas će serijalizirati objekt u JSON notaciju i konfigurirati HTTP odgovor koji se vraća korisničkom web browseru. U našem slučaju, postavili smo kod na 202 (HttpStatus.ACCEPTED).

Idemo to pogledati u browseru: