



IT355 - WEB SISTEMI 2

Spring MVC

Lekcija 05

PRIRUČNIK ZA STUDENTE

# IT355 - WEB SISTEMI 2

## Lekcija 05

### *SPRING MVC*

- ✓ Spring MVC
- ✓ Poglavlje 1: Dispatcher Servlet
- ✓ Poglavlje 2: Mapiranje zahteva sa @RequestMapping
- ✓ Poglavlje 3: Presretanje zahteva
- ✓ Poglavlje 4: Podrška za internacionalizaciju
- ✓ Poglavlje 5: Rešavanje pogleda pomoću naziva
- ✓ Poglavlje 6: Mapiranje izuzetaka u pogledu
- ✓ Poglavlje 7: Pridruživanje vrednosti u kontroleru
- ✓ Poglavlje 8: Upravljanje formama pomoću kontrolera
- ✓ Poglavlje 9: Spring MVC - pokazna vežba
- ✓ Poglavlje 10: Spring MVC - Individualna vežba 5
- ✓ Poglavlje 11: Domaći zadatak 5
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

# ▼ Uvod

## UVOD

*Lekcija se bavi razvojem web aplikacija upotrebom Spring MVC okvira.*

U ovoj lekciji akcenat će biti na savladavanju tehnika razvoja web baziranih aplikacija primenom *Spring MVC* okvira. *Spring MVC predstavlja jedan od najznačajnijih modula Spring okvira. Izgrađen je na moćnom Spring IoC kontejneru i omogućava široku upotrebu kontejnerskih elemenata.*

*Model View Controller (MVC)* je opšti šablon dizajna u UI dizajnu. On razdvaja poslovnu logiku od UI, razdvajanjem uloga modela, pogleda i kontrolera u aplikaciji. Modeli su odgovorni za enkapsulaciju podataka aplikacije koje pogledi prikazuju. Pogledi prikazuju samo ove podatke, bez uključivanja poslovne logike. Kontroleri su odgovorni za prijem zahteva od korisnika i pozivanje *back - end* servisa za poslovno procesiranje. Nakon završenog procesiranja, *back - end* servisi vraćaju izvesne podatke pogledima na prikazivanje. Osnovna ideja *MVC* šablona jeste razdvajanje poslovne logike od UI da bi mogli da se menjaju i razvijaju nezavisno, bez međusobnih uticaja.

U *Spring MVC* aplikaciji, model se obično sastoji od objekata domena koji su procesirani servisnim nivoom, a čuvani perzistentnim nivoom. Pogledi su obično *JSP* šabloni napisani standardnom JAVA bibliotekom tagova (*Java Standard Tag Library - JSTL*). Takođe, poglede *je moguće definisati i kao PDF i Excel datoteke, RESTful web servise ili čak Flex interfejse.*

Po savladavanju ove lekcije, studenti će biti sposobni da razvijaju JAVA web aplikacije primenom *Spring MVC* okvira.

## ▼ Poglavlje 1

# Dispatcher Servlet

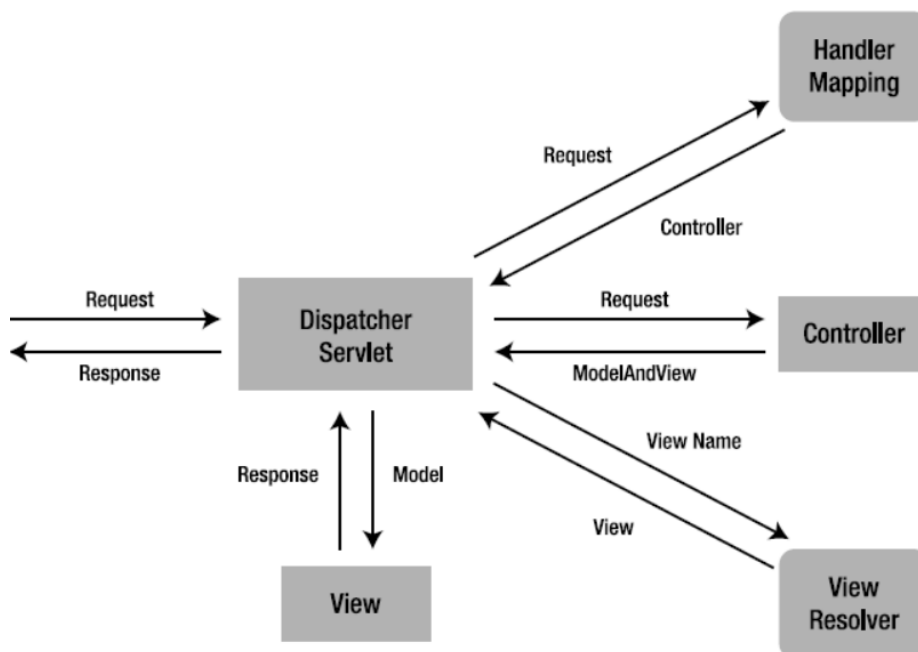
## PREDSTAVLJANJE MVC KOMPONENATA

*Centralna komponenta Spring MVC je kontroler.*

U prvom koraku neophodno je predstaviti razvoj jednostavne web aplikacije primenom *Spring MVC* radnog podokvira sa ciljem upoznavanja i savladavanja osnovnih koncepata ovog okvira.

Rešenje i njegova demonstracija, a sve u cilju lakšeg savladavanja lekcije, biće dato kroz odgovarajući primer. **Centralna komponenta Spring MVC je kontroler.** U najjednostavnijim JAVA aplikacijama, kontroler je, prosto, jedini servlet kojeg je neophodno podesiti u JAVA deskriptoru web razvoja, u datoteci **web.xml**. *Spring MVC kontroler*, koji se često naziva **Dispatcher Servlet**, implementira jedan od centralnih JAVA EE dizajn šablona poznatijih pod nazivom **front controller**. On upravo i vrši ulogu prednjeg ili glavnog kontrolera *Spring MVC okvira* i svaki web zahtev mora da prođe kroz njega da bi mogao da upravlja celokupnim procesom rukovanja zahtevima.

Kada je web zahtev prosleđen *Spring MVC* aplikaciji, kontroler prvi prima zahtev. Tada organizuje različite komponente podešene u kontekstu Spring web aplikacije ili anotacijama i sve u svrhu upravljanja zahtevima. Sledećom slikom prikazan je primarni tok upravljanja zahtevima u *Spring MVC*.



Slika 1.1 Primarni tok upravljanja zahtevima u Spring MVC [izvor: spring.io]

## DEFINISANJE KLASA KONTROLERA

*Odgovarajuća klasa mora da bude obeležena anotacijom **@Controller**.*

Da bi bila kreiranja klasa kontrolera, odgovarajuća klasa mora da bude obeležena anotacijom **@Controller**. Klasa kontrolera ne mora da implementira neki specifičan interfejs ili da proširi neku osnovnu klasu diktiranu okvirom, kao što je to bilo u ranijim verzijama Spring okvira.

Kada klasa koja je obeležena anotacijom **@Controller** kada primi zahtev, ona traži odgovarajuću metodu za rukovanje zahtevom. To traži da klasa kontrolera povezuje svaki zahtev sa metodom za rukovanje zahtevima, pomoću jednog ili više mapiranja. Sa namerom da to učini, metode klasa kontrolera su obeležene sa **@RequestMapping** anotacijom, postajući tako metode rukovaoci (**handler methods**). Moguće je specificirati proizvoljan naziv metode i definisati različite argumente metode. Isto tako, metoda rukovalac može da vrati bilo koji od niza vrednosti (na primer **String** ili **void**) u zavisnosti od logike aplikacije koju ispunjava. Postoje brojni i različiti argumenti koji mogu biti korišćeni u metodama koje su obeležene anotacijom **@RequestMapping**:

- **HttpServletRequest** ili **HttpServletResponse**;
- **@RequestParam** - zahtev parametara proizvoljnog tipa;
- **@ModelAttribute** - atributi modela proizvoljnog tipa;
- **@CookieValue** - vrednost kolačića uključenih u dolazni zahtev;
- **Map** ili **ModelMap** - za rukovalac metodu da doda atribut u model;
- **Errors** ili **BindingResult** - pristup rukovaoca povezivanju i proveru rezultata za komandni objekat;
- **SessionStatus** - za rukovalac metodu da obavesti da je završila procesiranje u sesiji.

Kada kontroler klasa pokupi odgovarajuću rukovalac metodu, ona poziva izvršavanje logike te metode zahtevom. Uglavnom, radi se o pozivanju **back - end** servisa za rukovanje zahtevima. U nastavku, logika metode obično dodaje i briše informacije od brojnih ulaznih argumenata (na primer **HttpServletRequest**, **Map**, **Errors** ili **SessionStatus**) koji će činiti deo tekućeg *Spring MVC* toka.

Nakon što metoda završi obradu zahteva, vraća kontrolu pogledu, koja je označena kao povratna vrednost metode rukovaoca. Za obezbeđivanje fleksibilnog pristupa, ova povratna vrednost ne reprezentuje implementaciju pogleda (na primer, *user.jsp* ili *report.pdf*) već logički pogled (na primer, *user* ili *report*) - primećuje se izostavljanje ekstenzije.

Povratna vrednost, **handler** metode, može biti **String** - predstavlja naziv logičkog pogleda, ili **void** - u tom slučaju je određen podrazumevani naziv logičkog pogleda u zavisnosti od naziva metode rukovaoca ili kontrolera.

Sa namerom prosleđivanja informacija od kontrolera ka pogledu, irelevantno je kako se vraća naziv logičkog pogleda - **String** ili **void**, pošto će ulazni argumenti metode rukovaoca biti dostupni pogledu. Na primer, ako metoda primi kao ulazne argumente objekte tipa **Map** i **SessionStatus**, modifikacijom njihovog sadržaja u okviru logike metode, ovi objekti će biti dostupni pogledu, vraćeni rukovalac metodom.

## DEFINISANJE KLASA KONTROLERA - DOPUNSKA RAZMATRANJA

*Rešavač je zрно podešeno u kontekstu web aplikacije, koje implementira ViewResolver interfejs.*

Kada klasa kontrolera primi pogled, ona rešava naziv logičkog pogleda u implementaciji pogleda (na primer, *user.jsp* ili *report.pdf*) u smislu rešavača pogleda (*view resolver*). Rešavač je zрно podešeno u kontekstu web aplikacije i koje implementira ViewResolver interfejs. Njegova dužnost je da vrati specifičnu implementaciju pogleda (*HTML, JSP, PDF* ili drugi) za naziv logičkog pogleda. Nakon ovoga, dolazi do formiranja objekata (na primer *HttpServletRequest, Map, Errors* ili *SessionStatus*) prosleđenih kontrolerovom metodom rukovaocem. Dužnost pogleda je prikazivanje objekata dodatih u logiku metode rukovaoca korisniku.

## SPRING MVC APLIKACIJA - DOMENSKE KLASKE

*Razvija se aplikacija za demonstraciju navedenog izlaganja.*

Pretpostavka je da se razvija aplikacija za rezervaciju terena u okviru sportskog centra. UI ove aplikacije je web baziran pa korisnici mogu da obavljaju rezervacije putem Interneta. Aplikacija se razvija pomoću *Spring MVC* okvira. Prvi zadatak je definisanje domenskih klasa u okviru potpaketa *domain* koji će da sadrži domenske klase projekta.

Prva domenska klasa koja će biti kreirana predstavlja osnov za kreiranje zrna rezervacije:

```
package com.metropolitan.domain;

import java.util.Date;

/**
 *
 * @author Vlada
 */
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    public Reservation(String courtName, Date date, int hour, Player player,
SportType sportType) {
        this.courtName = courtName;
        this.date = date;
        this.hour = hour;
    }
}
```

```
        this.player = player;
        this.sportType = sportType;
    }

    public String getCourtName() {
        return courtName;
    }

    public void setCourtName(String courtName) {
        this.courtName = courtName;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public int getHour() {
        return hour;
    }

    public void setHour(int hour) {
        this.hour = hour;
    }

    public Player getPlayer() {
        return player;
    }

    public void setPlayer(Player player) {
        this.player = player;
    }

    public SportType getSportType() {
        return sportType;
    }

    public void setSportType(SportType sportType) {
        this.sportType = sportType;
    }
}
```

Domenski nivo aplikacije izgrađuje se dalje kreiranjem klase za definisanje različitih tipova sporta:

```
package com.metropolitan.domain;

/**
```

```
*
* @author Vlada
*/
public class SportType {

    private int id;
    private String name;

    public SportType(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

}
```

Konačno, razvoj domena završava se dodavanjem klase za kreiranje objekata igrača.

```
package com.metropolitan.domain;

/**
 *
 * @author Vlada
 */
public class Player {

    private String name;
    private String phone;

    public Player(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    public String getName() {
        return name;
    }

}
```



```

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

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

## RAZVOJ SERVISNOG NIVOVA SPRING MVC APLIKACIJE

*Servisna implementaciona klasa je obeležena je anotacijom @Service*

U prethodnim lekcijama je detaljno opisana uloga servisnog sloja aplikacije. Da bi *Spring IoC* kontejner dobio informaciju da se u aplikaciji nalazi klasa zrna iz servisnog nivoa, odgovarajuća klasa mora da bude obeležena anotacijom *@Service*. Ova klasa predstavlja implementaciju nekog servisnog interfejsa.

Da bi navedeno bilo implementirano u strukturu projekta, posmatrane veb aplikacije, biće kreiran sledeći servisni interfejs, u potpaku *service*, za obezbeđivanje servisa rezervacije u prezentacionom nivou.

```

package com.metropolitan.service;

import com.metropolitan.domain.Reservation;
import java.util.List;

/**
 *
 * @author Vlada
 */
public interface ReservationService {

    public List<Reservation> query(String courtName);
}

```

Dalje, ovaj interfejs, u realnim uslovima, bi trebalo da bude implementiran sa perzistencijom iz baze podataka. Međutim, zbog pojednostavljenja primera i testiranja, nekoliko rezervacionih slogova biće ugrađeno u programski kod. Sledi listing implementacione klase kreiranog servisnog interfejsa.

```

package com.metropolitan.service;

```

```
import com.metropolitan.domain.Player;
import com.metropolitan.domain.Reservation;
import com.metropolitan.domain.SportType;
import org.springframework.stereotype.Service;
import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

/**
 *
 * @author Vlada
 */
@Service
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");
    private final List<Reservation> reservations = new ArrayList<>();

    public ReservationServiceImpl() {
        reservations.add(new Reservation("Tennis #1", new GregorianCalendar(2018,
9, 14).getTime(), 16,
        new Player("Roger", "N/A"), TENNIS));
        reservations.add(new Reservation("Tennis #2", new GregorianCalendar(2018,
10, 14).getTime(), 20,
        new Player("Novak", "N/A"), TENNIS));
    }

    @Override
    public List<Reservation> query(String courtName) {
        return this.reservations.stream().filter(reservation ->
Objects.equals(reservation.getCourtName(), courtName))
        .collect(Collectors.toList());
    }
}
```

Za potrebe demonstracije, kreirana su dva objekta igrača: "Roger" i "Novak" koji će rezervisati dva različita teniska terena, a ovi podaci će naknadno biti prikazani u prezentacionom nivou aplikacije.

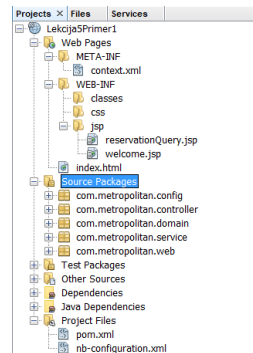
## PODEŠAVANJE SPRING MVC APLIKACIJE

*Spring MVC aplikacija koristi web.xml ili implementaciju ServletContainerInitializer interfejsa.*

Sledeće što je neophodno uraditi jeste kreiranje rasporeda (layout) Spring MVC aplikacije. U osnovi, Spring MVC aplikacija se podešava skoro naisti način kao klasična JAVA web aplikacija,

sa razlikom što je neophodno dodati nekoliko konfiguracionih datoteka i specifičnih biblioteka karakterističnih za Spring MVC.

Konkretna *JAVA EE* specifikacija definiše strukturu direktorijuma JAVA web aplikacije sačinjenu od web arhiva ili *WAR* datoteka. Na primer, neophodno je obezbediti deskriptor web razvoja, na primer *web.xml*, u korenu *WEB-INF* ili, *kako jeto aktuelnije, jednu ili više klasa za implementaciju ServletContainerInitializer* interfejsa. Datoteke klase i JAR datoteke za web aplikaciju, mogu da budu smeštene u *WEB-INF/classes* i *WEB-INF/lib* direktorijume, respektivno. Za sistem rezervacije terena, predložena je sledeća struktura aplikacije.



Slika 1.2 Struktura projekta Spring MVC aplikacije [izvor:autor]

Datoteke izvan *WEB-INF* direktorijuma su direktno dostupne korisnicima preko URL, zato *CSS* datoteke i *image* datoteke moraju da budu smeštene ovde. Kada se koristi *Spring MVC*, *JSP* datoteke se ponašaju kao šabloni. Njih čita okvir za generisanje dinamičkog sadržaja, pa je *JSP* datoteke neophodno smestiti u *WEB-INF* direktorijum zbog prevencije direktnog pristupa njima. Međutim, neki aplikacioni serveri ne dozvoljavaju da datoteke iz *WEB-INF* direktorijuma budu čitane interno web aplikacijom. U tom slučaju, moguće je samo njih izolovati izvan *WEB-INF* direktorijuma.

Korišćenjem *Maven*-a, neophodno je ugraditi i odgovarajuće zavisnosti u *pom.xml* da bi aplikacija mogla da bude uspešno kreirana i prevedena.

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.0.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
```

```

        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <type>jar</type>
    </dependency>
</dependencies>

```

## KREIRANJE KONFIGURACIONIH JAVA KLASA ZA SPRING MVC APLIKACIJU

*DispatcherServlet je osnovna servlet klasa koja prima i prosleđuje veb zahteve.*

Deskriptor veb razvoja **web.xml** je osnovni konfiguracioni fajl JAVA veb aplikacija. U ovom fajlu definišu se servleti aplikacije i načini mapiranja web zahteva sa njima. Za *Spring MVC* aplikaciju, neophodno je definisati jednu *DispatcherServlet* instancu koja se ponaša kao prednji kontroler za *Spring MVC*. Međutim u savremenim Java veb aplikacijama **web.xml** ima opcionu ulogu i u velikom broju slučajeva čak se i gubi potreba za njegovim korišćenjem. On može biti zamenjen implementacijom *ServletContainerInitializer* interfejsa.

U velikim, korporacijskim aplikacijama, moguće je izabrati scenario kreiranja većeg broja *DispatcherServlet* instanci vezanih za specifične URL. Na ovaj način se dodatno razdvaja programska logika, a samo kreiranje i održavanje koda je dodatno pojednostavljeno.

```

package com.metropolitan.web;

/**
 *
 * @author Vlada
 */
public class CourtServletContainerInitializer implements
ServletContainerInitializer {

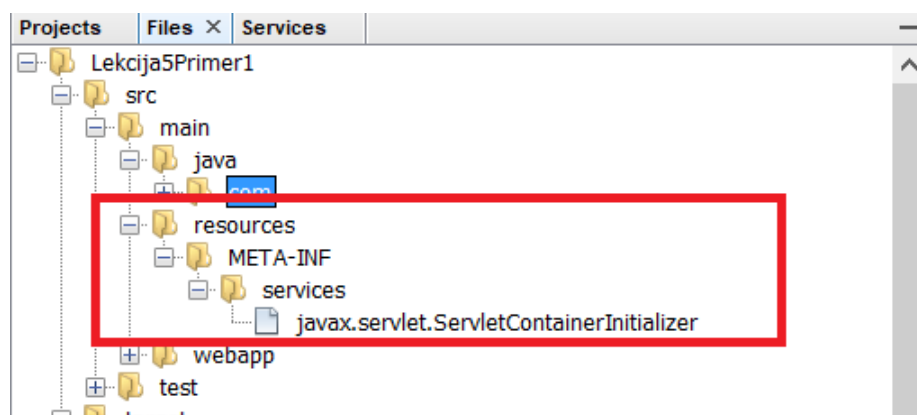
    @Override
    public void onStartup(Set<Class<?>> c, ServletContext ctx) throws

```

```
ServletException {
    AnnotationConfigWebApplicationContext applicationContext
        = new AnnotationConfigWebApplicationContext();
    applicationContext.register(CourtConfiguration.class);
    DispatcherServlet dispatcherServlet = new
DispatcherServlet(applicationContext);
    ServletRegistration.Dynamic courtRegistration
        = ctx.addServlet("court", dispatcherServlet);
    courtRegistration.setLoadOnStartup(1);
    courtRegistration.addMapping("/");
}
}
```

U ovoj klasi, definisan je servlet tipa **DispatcherServlet**. Ovo je osnovna servlet klasa u **Spring MVC** koja prima zahteve i prosleđuje ih odgovarajućim rukovaocima. Naziv ovog servleta biće podešen na **court**, a sva URL mapiranja biće urađena primenom znaka **/**.

Da bi oba klasa mogla da bude otkrivena, od strane **Spring IoC kontejnera**, i upotrebljena na pravi način, neophodno je kreirati datoteku pod nazivom **javax.servlet.ServletContainerInitializer** u folderu **META-INF/services**. Sadržaj ove datoteke bi trebalo da bude pun naziv klase **CourtServletContainerInitializer**.



Slika 1.3 Lokacija javax.servlet.ServletContainerInitializer datoteke [izvor:autor]

Sledećim listingom je prikazan sadržaj ove datoteke:

```
com.metropolitan.web.CourtServletContainerInitializer
```

## KREIRANJE KONFIGURACIONIH KLASA - DOPUNSKA RAZMATRANJA.

*Konačno je moguće kreirati glavnu konfiguracionu klasu aplikacije.*

Konačno je moguće kreirati glavnu konfiguracionu klasu aplikacije. Biće joj dodeljen naziv *CourtConfiguration* i biće smeštena u paket označen sa *config*. Sledećim listingom je priložen kod ove klase.

```
package com.metropolitan.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

/**
 *
 * @author Vlada
 */
@Configuration
@EnableWebMvc
@ComponentScan("com.metropolitan")
public class CourtConfiguration {

    @Bean
    public InternalResourceViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver viewResolver = new
InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

Da se radi o konfiguracionoj klasi jasno se vidi pošto je obeležena anotacijom *@Configuration*. Dalje, primena anotacije *@ComponentScan* dozvoljava pretragu komponentata u okviru paketa *com.metropolitan*, kao i svih njegovih potpaketa, i registrovanje zrna koja su tom prilikom otkrivena. Učitavanjem ove klase u *Spring IoC kontejner* odmah će biti registrovano zrno klase *ReservationServiceImpl* jer je obeleženo anotacijom *@Service*.

Na kraju, biće registrovano i zrno tipa *InternalResourceViewResolver*. Zadatak ovog zrna jeste povezivanje logičkih pogleda (povratna String vrednost u odgovarajućoj metodi kontrolera) sa lokacijom odgovarajuće JSP datoteke. Na primer, logički pogled pod nazivom *reservationQuery* je dodeljen implementaciji pogleda smeštenoj na lokaciji */WEB-INF/jsp/reservationQuery.jsp*. Kažemo da ovakva zrna "rešavaju poglede" i zato ih možemo nazivati *rešavačima pogleda*.

Dodavanjem anotacije *@EnableWebMVC*, konfiguraciona klasa dobija mogućnost učitavanja Spring MVC konfiguracija.

# KREIRANJE SPRING MVC KONTROLERA - OSNOVNE ANOTACIJE

*Kontroler klasa može biti proizvoljna klasa koja ne implementira interfejs ili nasleđuje neku klasu.*

Kontroler klasa, koja koristi anotacije, može biti proizvoljna klasa koja ne implementira određeni interfejs ili nasleđuje neku osnovnu klasu. Obeležava se anotacijom **@Controller**. Ova klasa može da poseduje jednu ili više metoda rukovalaca za upravljanje jednom ili više akcija. Potpis ovih metoda je dovoljno fleksibilan za prihvatanje različitih argumenata.

Anotacija **@RequestMapping** može biti primenjena na nivo klase ili nivo metode. Prva strategija se odnosi na mapiranje izvesnog URL šablona sa klasom kontrolera, a potom i izvesne HTTP metode sa svakom metodom rukovaocem:

```
package com.metropolitan.controller;

import java.util.Date;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 *
 * @author Vlada
 */
@Controller
@RequestMapping("/welcome")
public class WelcomeController {

    @RequestMapping(method = RequestMethod.GET)
    public String welcome(Model model) {
        Date today = new Date();
        model.addAttribute("today", today);
        return "welcome";
    }
}
```

Ovaj kontroler kreira `java.util.Date` objekat za prihvatanje tekućeg datuma, kojeg posle dodaje u ulazni `Model` objekat kao atribut kojeg će prikazati ciljni pogled. Budući da je već aktivirano skeniranje anotacija za paket `com.metropolitan` (i sve potpake) anotacije klase kontrolera su detektovane odmah nakon aktiviranja konfiguracione klase.

Anotacija **@Controller** definiše klasu kao *Spring MVC kontroler*. Anotacija **@RequestMapping** je interesantnija jer sadrži osobine i može biti deklarirana na nivou klase ili metode rukovaoca. Prva korišćena vrednost `"/welcome"` je korišćena da specificira URL za koji će kontroler biti aktivan, a to znači da bilo koji zahtev primljen `/welcome` URL-om je pod pažnjom klase `WelcomeController`.

Kada je zahtev prisutan za klasu kontrolera, obavlja se pozivanje podrazumevane HTTP GET metode rukovaoca deklarisanе u kontroleru.

Anotacija `@RequestMapping(method = RequestMethod.GET)` je korišćena za označavanje `welcome` metode kao podrazumevane HTTP GET metode rukovaoca. Važno je napomenuti da ukoliko podrazumevana HTTP GET metode nije deklarisanа, biće izbačen izuzetak `ServletException`.

Izvršavanjem navedene metode vraća se String vrednost "`welcome`" koja se naziva logičkim pogledom. Vraćanjem ove vrednosti dolazi do pronalaženja i učitavanja pogleda (veb stranice) `welcome.jsp`.

## KREIRANJE SPRING MVC KONTROLERA - DODATNE ANOTACIJE

*Druga varijacija ovog pristupa je deklarisanje obeju vrednosti - URL i metoda.*

Druga varijacija ovog pristupa je deklarisanje obeju vrednosti - URL putanje i podrazumevane HTTP GET handler metode - u anotaciji `@RequestMapping` primenjenoj na nivou metode:

```
@Controller
public class WelcomeController {
    @RequestMapping(value = "/welcome", method=RequestMethod.GET)
    public String welcome(Model model) {
        ...
    }
}
```

Ova deklaracija je ekvivalentna prethodnoj. Atribut `value` ukazuje na URL koji se vezuje za rukovalac metodu, a atribut `method` definiše metodu kao kontrolerovu podrazumevanu HTTP GET metodu.

Postoje, takođe, još neke pogodne anotacije kojima je moguće dodatno pojednostaviti podešavanja unutra kontrolera. Na primer, primenom anotacija `@GetMapping` i `@PostMapping` jasno se pokazuje koji je tip HTTP metode u pitanju. Primenom navedenih anotacija prethodni listing može da dobije sledeći ekvivalentan oblik.

```
@Controller
public class WelcomeController {
    @GetMapping("/welcome")
    public String welcome(Model model) {
        ...
    }
}
```

## KREIRANJE SPRING MVC KONTROLERA - UPOTREBA SERVISA

*Spring MVC kontroler često koristi pogodne back-end servise.*



Prethodni kontroler je poslužio za demonstraciju osnovnih principa *Spring MVC*. Međutim, klasičan kontroler veoma često može da poziva pozadinske servise za procesiranje izvesne poslovne logike. Na primer, moguće je kreirati sledeći kontroler koji obrađuje upite rezervisanja sportskog terena.

```
/**
 *
 * @author Vlada
 */
@Controller
@RequestMapping("/reservationQuery")
public class ReservationQueryController {

    private final ReservationService reservationService;

    public ReservationQueryController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @GetMapping
    public void setupForm() {

    }

    @PostMapping
    public String submitForm(@RequestParam("courtName") String courtName, Model
model) {
        List<Reservation> reservations = java.util.Collections.emptyList();
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
        model.addAttribute("reservations", reservations);
        return "reservationQuery";
    }
}
```

Kao što je već diskutovano, kontroler će potražiti podrazumevanu metodu za rukovanje HTTP GET zahtevom. Otuda je javna i *void* metoda *setupForm()* pridružena neophodnoj anotaciji *@RequestMapping* u ovom slučaju. Ova metoda se poziva sledeća.

Ovde je moguće primetiti da je ona prazna i da ne poseduje ulazne parametre, a pri tome je i tipa *void*. To može da znači dve stvari:

- pogled prikazuje samo podatke koji su "čvrsto kodirani" u implementacionom šablonu (JSP) budući da nema podataka prosleđenih kontroleru;
- *void* povratna vrednost znači da će podrazumevani naziv pogleda (baziran na URL - u zahteva) biti korišćen. U konkretnom slučaju, */reservationQuery* je podrazumevani povratni naziv pogleda (logički pogled).

## KREIRANJE SPRING MVC KONTROLERA - POST METODA

*Zahtevi tipa HTTP POST se uglavnom koriste prilikom korisničkog popunjavanja HTML forme.*

Preostaje još analiza sledećeg izolovanog koda kontrolera.

```
@PostMapping
public String submitForm(@RequestParam("courtName") String courtName, Model
model) {
    List<Reservation> reservations = java.util.Collections.emptyList();
    if (courtName != null) {
        reservations = reservationService.query(courtName);
    }
    model.addAttribute("reservations", reservations);
    return "reservationQuery";
}
```

Poslednja rukovalac metoda je obeležena anotacijom `@RequestMapping(method = RequestMethod.POST)`. Metoda se poziva kada je **HTTP POST** zahtev kreiran za `/reservationQuery` URL. **HTTP POST** metoda uključuje dva ulazna parametra. Prva deklaracija `@RequestParam("courtName") String courtName`, primenjena je da izvuče zahtevani parametar `courtName`. U ovom slučaju, **HTTP POST** zahtev dolazi u obliku `/reservationQuery?courtName = <value>`. Druga **Model** deklaracija, definiše objekat u kojem se prosleđuje podatak povratnom pogledu.

Logika izvršena rukovalac metodom, sastoji se od primene kontrolerovog `reservationService` za izvođenje upita primenom varijable `courtName`. Rezultat dobijen ovim upitom pridružen je **Model** objektu da bi kasnije postao dostupan za prikazivanje u povratnom pogledu.

Konačno, metoda je vratila pogled pod nazivom `reservationQuery`.

Metoda je mogla da vrati tip `void`, baš kao podrazumevani **HTTP GET** i da bude u vezi sa istim `reservationQuery` podrazumevanim pogledom, a na račun zahtevanog URL - a. Oba pristupa su identična.

Nakon završenog konstituisanja **Spring MVC** kontrolera, vreme je za istraživanje pogleda kojima kontrolerove handler metode prosleđuju rezultate za prikazivanje.

Većina zahteva u veb aplikacijama je tipa **HTTP GET**. Zahtevi tipa **HTTP POST** se uglavnom koriste prilikom korisničkog popunjavanja **HTML** forme.

## KREIRANJE JSP POGLEDA.

*U Spring MVC aplikacijama, pogledi su najčešće JSP šabloni.*

U *Spring MVC* aplikacijama, pogledi su najčešće *JSP* šabloni kreirani primenom *JSTL*. Kada **DispatcherServlet** primi naziv pogleda, vraćen od rukovaoca, on prosleđuje naziv logičkog pogleda u implementaciju pogleda koji se generiše. Pokazano je kako se podešava zrna *InternalResourceViewResolver* konteksta web aplikacije za povezivanje naziva pogleda sa JSP datotekom u direktorijumu */WEB-INF/jsp/*.

Primenom poslednje konfiguracije, logički pogled sa nazivom *reservationQuery* je određen za implementaciju pogleda lociranog na */WEB-INF/jsp/reservationQuery.jsp*.

Znajući to, moguće je kreirati sledeći šablon za kontroler dobrodošlice, dajući mu naziv *welcome.jsp* i čuvajući ga na lokaciji */WEB-INF/jsp/*.

```
<%--
    Document    : welcome
    Created on   : 18.09.2018., 12.19.53
    Author      : Vlada
--%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
    <head>
        <title>Welcome</title>
    </head>
    <body>
        <h2>Welcome to Court Reservation System</h2>
        Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />.
    </body>
</html>
```

U ovom JSP šablonu, omogućena je upotreba biblioteke tagova *fmt* iz *JSTL*, za formatiranje *today* atributa modela u šablonu *yyyy-MM-dd*.

Sada je moguće kreirati drugi JSP šablon za kontroler upita rezervacije i nazvati ga *reservationQuery.jsp* za podudaranje sa nazivom pogleda:

```
<%--
    Document    : reservationQuery
    Created on   : 18.09.2018., 12.19.31
    Author      : Vlada
--%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
    <head>
        <title>Reservation Query</title>
    </head>
    <body>
        <form method="post">
            Court Name
            <input type="text" name="courtName" value="${courtName}" />
            <input type="submit" value="Query" />
        </form>
```

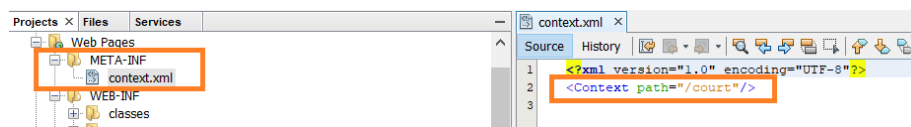
```
<table border="1">
  <tr>
    <th>Court Name</th>
    <th>Date</th>
    <th>Hour</th>
    <th>Player</th>
  </tr>
  <c:forEach items="${reservations}" var="reservation">
    <tr>
      <td>${reservation.courtName}</td>
      <td><fmt:formatDate value="${reservation.date}"
pattern="yyyy-MM-dd" /></td>
      <td>${reservation.hour}</td>
      <td>${reservation.player.name}</td>
    </tr>
  </c:forEach>
</table>
</body>
</html>
```

U ovom JSP šablonu, uključena je forma za unos podataka od strane korisnika - naziv terena za koji se vrši upit. Potom je korišćen tag petlja `<c:forEach>` za prolaz kroz attribute modela *reservations* i generisanje tabele rezultata.

## ANGAŽOVANJE WEB APLIKACIJE

*Neophodno je izabrati odgovarajući web kontejner za testiranje i debugovanje.*

U strukturi kreirane veb aplikacije postoji još jedna značajna datoteka *context.xml*. U njoj je definisan osnovni URL za pozivanje kreiranih pogleda. Sledećom slikom je pokazana lokacija u projektu i sadržaj ove datoteke.

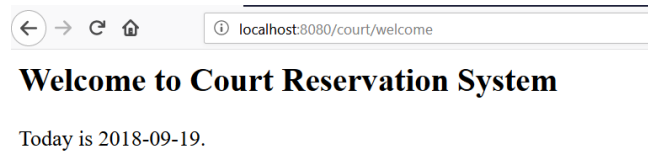


Slika 1.4 Lokacija u projektu i sadržaj datoteke context.xml. [izvor:autor]

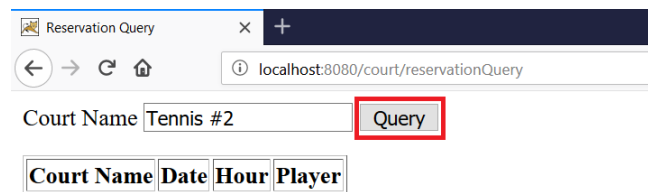
Tokom proces razvoja i angažovanja veb aplikacije strogo je preporučljivo imati instaliran *JAVA EE* server aplikacija koji dolazi sa veb kontejnerom za svrhu testiranja i otklanjanja grešaka. Zbog olakšane konfiguracije i razvoja, moguće je izabrati i *Apache Tomcat* kao veb kontejner.

Razvojni direktorijum za ovaj veb kontejner je lociran pod direktorijumom *webapps*. Po osnovnim podešavanjima *Apache Tomcat* osluškuje na **portu 8080** i upošljava aplikacije na kontekstu po istom nazivu kao što je pokazano prethodnom slikom. Otuda, pozdravnom kontroleru i kontroleru upita rezervacije moguće je pristupiti na sledeći način:

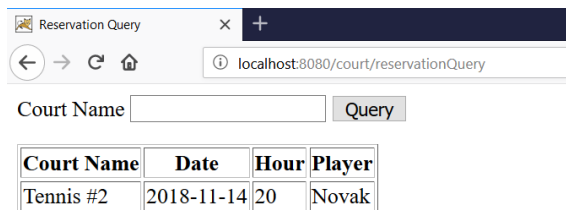
```
http://localhost:8080/court/welcome
http://localhost:8080/court/reservationQuery
```



Slika 1.5 Pristup pozdravnom kontroleru [izvor:autor]



Slika 1.6 Pristup kontroleru upita - GET [izvor:autor]



Slika 1.7 Pristup kontroleru upita - POST [izvor:autor]

## SPRING MVC KONFIGURACIJE I DISPATCHERSERVLET - VIDEO

*Izalaganje ovog dela lekcije je moguće zaokružiti sledećim video materijalima.*

Spring MVC Java konfiguracije

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

DispatcherServlet

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

# Mapiranje zahteva sa @RequestMapping

## DEFINISANJE PROBLEMA

*Neophodno je definisati strategiju mapiranja zahteva primenom anotacija @RequestMapping.*

PROBLEM: Kada `DispatcherServlet` primi web zahtev, pokušava da prosledi zahtev različitim klasama kontrolera, koje su deklarisanе anotacijom `@Controller`. Proces prosleđivanja zavisi od anotacija `@RequestMapping`, kao i njihovog sadržaja, deklarisanih u klasi kontrolera i odgovarajućih metoda rokovalaca. Neophodno je definisati strategiju mapiranja zahteva primenom anotacija `@RequestMapping`.

Kako to funkcioniše u `Spring MVC` okviru? U `Spring MVC` aplikaciji, web zahtevi su mapirani sa rukovaocima pomoću jedne ili više anotacija `@RequestMapping`, deklarisanim u klasama kontrolera.

Mapiranje rukovaoca vrši podudaranje sa URL-om na osnovu njihovih putanja u odnosu na putanju konteksta i putanju servleta (na primer, putanja mapirana sa `DispatcherServlet`). Na primer, u URL-u `http://localhost:8080/court/welcome`, putanja za mapiranje je `/welcome`, dok je putanja konteksta `/court` i nema putanje servleta - deklarisan je kao `/` u implementaciji za `ServletContainerInitializer` (ili datoteci `web.xml`).

Neophodno je razlikovati dva načina primene mapiranja zahteva primenom anotacije `@RequestMapping`:

- mapiranje na nivou klase;
- mapiranje na nivou metode.

U sledećem izlaganju sledi analiza i demonstracija oba navedena načina primene posmatrane anotacije.

## MAPIRANJE ZAHTEVA METODOM

*Najjednostavnija primena @RequestMapping anotacija je direktno obeležavanje metode rukovaoca.*

Najjednostavnija strategija za primenu `@RequestMapping` anotacija je direktno obeležavanje metode rukovaoca. Da bi ovo funkcionisalo, neophodno je deklarirati svaku od ovih metoda primenom anotacije `@RequestMapping` koja sadrži odgovarajući URL šablon.

Ako se anotacija rukovaoca podudara sa URL zahteva, **DispatcherServlet** vrši prosleđivanje zahteva rukovaocu za njihovo obrađivanje.

```
/**
 *
 * @author Vlada
 */
@Controller
public class MemberController {

    private MemberService memberService;

    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("/member/add")
    public String addMember(Model model) {

        model.addAttribute("member", new Member());

        model.addAttribute("guests", memberService.list());

        return "memberList";
    }

    @RequestMapping(value = {"/member/remove", "/member/delete"}, method =
RequestMethod.GET)
    public String removeMember(@RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect:";
    }
}
```

Poslednji listing ilustruje kako se svaka rukovalac metoda mapira u odgovarajući URL primenom **@RequestMapping** anotacije. Druga metoda ilustruje angažovanje više URL-a, tako da oba `/member/remove` i `/member/delete` iniciraju izvršenje metode rukovaoca. Po osnovnim podešavanjima, pretpostavlja se da su svi dolazni zahtevi tipa `HTTP GET`.

## MAPIRANJE ZAHTEVA KLASOM

*Anotacija **@RequestMapping**. takođe, može biti iskorišćena i za obeležavanje klase.*

Anotacija **@RequestMapping**. takođe, može biti iskorišćena i za obeležavanje klase. Sledećim listingom prikazana je primena URL džokera u anotaciji **@RequestMapping**, baš kao i fino granulirano URL podudaranje bazirano na **@RequestMapping** anotacijama za metode rukovaoca.

```
@Controller
@RequestMapping("/member/*")
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("add")
    public String addMember(Model model) {
        model.addAttribute("member", new Member());
        model.addAttribute("guests", memberService.list());
        return "memberList";
    }

    @RequestMapping(value = {"remove", "delete"},
        method = RequestMethod.GET)
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect:";
    }

    @RequestMapping("display/{user}")
    public String removeMember(
        @RequestParam("memberName") String memberName,
        @PathVariable("user") String user) {
        ....
    }

    @RequestMapping
    public void memberList() {
        ....
    }

    public void memberLogic(String memberName) {
        ....
    }
}
```

Moguće je primetiti da anotacija **@RequestMapping** na nivou klase koristi URL džoker `/member/*`. Ovo, za uzvrat, šalje sve zahteve pod `/member/` URL-om kontrolerovim metodama rukovaocima. Prve dve metode koriste **@RequestMapping** anotaciju. Metoda `removeMember()` se poziva kada je zahtev **HTTP GET** napravljen za bilo koji URL `/memeber/remove` ili `/memeber/delete`.

Treća metoda koristi posebnu vrstu anotacije `{path_variable}` za specificiranje vrednosti vlastite **@RequestMapping** anotacije. Na taj način, vrednost prezentovana URL-om može biti prosleđena kao ulaz za handler metodu. Neophodno je pogledati i sledeću deklaraciju ove



metode: `@PathVariable("user") String user`. Na ovaj način, ako je zahtev primljen u formi `member/display/vlada`, metoda ima pristup `user` varijabli čija je vrednost `"vlada"`.

Četvrta metoda takođe koristi `@RequestMapping` anotaciju, ali u ovom slučaju nedostaje URL vrednost. Budući da je na nivou klase primenjen džoker `/member/*`, ova rukovalac metoda se izvršava po scenariju *uhvati sve*. Znači, bilo koji URL zahtev pokreće ovu metodu. Takođe, neophodno je primetiti povratnu vrednost tipa `void`, a to zauzvrat čini metodu rukovaoca podrazumevanom za pogled sa istim nazivom (`memberList`).

Poslednja metoda, `memberLogic`, ne poseduje `@RequestMapping` anotaciju, to znači da je metoda alat klase koji nema uticaja na Spring MVC.

## MAPIRANJE ZAHTEVA POMOĆU TIPRA HTTP ZAHTEVA

*`@RequestMapping` anotacija pretpostavlja da su svi dolazni zahtevi tipa `HTTP GET`.*

Po osnovnim podešavanjima, `@RequestMapping` anotacija pretpostavlja da su svi dolazni zahtevi tipa `HTTP GET`. Međutim, ako je dolazni zahtev nekog drugog tipa, to je neophodno eksplicitno specificirati u `@RequestMapping` anotaciji, baš kao što je urađeno u sledećem listingu:

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@ModelAttribute("member") Member member,
    BindingResult result, Model model) {
    ....
}
```

Ova poslednja metoda ima tretman podrazumevane rukovalac metode za bilo koji `HTTP POST` zahtev učinjen na klasi kontrolera. Takođe, moguće je koristiti atribut `value` za specificiranje eksplicitnog URL -a za metodu rukovaoca. To je ilustrovano sledećim:

```
@RequestMapping(value= "processUser" method = RequestMethod.POST)
public String submitForm(@ModelAttribute("member") Member member,
    BindingResult result, Model model) {
    ....
}
```

Postoje različiti tipovi **HTTP** zahteva: **HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS** i **CONNECT**. Podrška za upravljanje svim ovim zahtevima izlazi iz oblasti **Spring MVC**, budući da web server, kao stana koja traži zahteve, mora da podržava ove tipove zahteva. Dakle, ovde je bio akcenat na **GET** i **POST** tipu zahteva kao dominantnim tipovima za koje ovde postoji podrška. Za ostale tipove, retko će biti tražena podrška u praksi.

Za najčešće korišćene metode, Spring MVC je obezbedio odgovarajuće anotacije i one su prikazane sledećom tabelom.

Metoda zahteva	Anotacija
POST	@PostMapping
GET	@GetMapping
DELETE	@DeleteMapping
PUT	@PutMapping

Slika 2.1 Anotacije metoda zahteva [izvor:autor]

## DODAVANJE KORISNIKA - MODEL I SERVIS

### *Kontroler zahteva modelsku klasu.*

Kao prethodnice prikazanom kontroleru, bilo je neophodno kreirati modelsku klasu *Member.java*. Sledi listing ove klase.

```
package com.metropolitan.domain;

/**
 *
 * @author Vlada
 */
public class Member {

    private String name;
    private String phone;
    private String email;

    public Member() {
    }

    public Member(String name, String phone, String email) {
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getName() {
```

```
        return name;
    }

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

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

Dalje, po već ustanovljenoj praksi kreira se interfejs za korisnika.

```
package com.metropolitan.service;

import com.metropolitan.domain.Member;
import java.util.List;

/**
 *
 * @author Vlada
 */
public interface MemberService {

    void add(Member member);

    void remove(String memberName);

    List<Member> list();

}
```

Konačno, tu je i implementaciona klasa kreiranog servisa.

```
/**
 *
 * @author Vlada
 */
@Service
class MemberServiceImpl implements MemberService {

    private Map<String, Member> members = new TreeMap<>();

    @Override
    public void add(Member member) {
        members.put(member.getName(), member);
    }
}
```

```

    }

    @Override
    public void remove(String memberName) {
        members.remove(memberName);
    }

    @Override
    public List<Member> list() {

        return new ArrayList<>(members.values());
    }
}

```

## JSP STRANICA I DEMO

*Neophodno je kreirati i JSP stranice koje odgovaraju logičkim pogledima kontrolera.*

Sledi listing JSP datoteke *memberList.jsp* za prikazivanje liste kreiranih članova.

```

<%- -
    Document    : memberList
    Created on  : 19.09.2018., 13.14.52
    Author      : Vlada
- -%>

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>
    <head>
        <title>Member List</title>
        <style>
            .error {
                color: #ff0000;
                font-weight: bold;
            }
        </style>
    </head>

    <body>

        <form:form method="post" modelAttribute="member" >
            <form:errors path="*" cssClass="error" />
            <table>
                <tr>
                    <td>Name <form:input path="name"/></td>

```

```

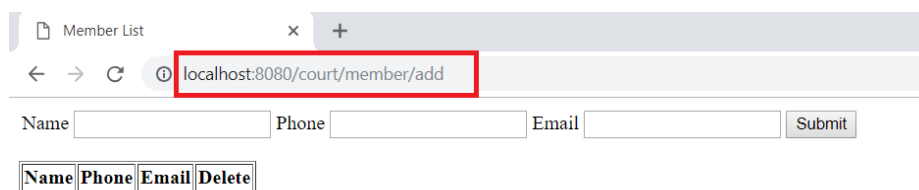
        <td>Phone <form:input path="phone"/></td>
        <td>Email <form:input path="email"/></td>
        <td><input type="submit" /></td>
    </tr>
</table>
</form:form>
<table border="1">
    <tr>
        <th>Name</th>
        <th>Phone</th>
        <th>Email</th>
        <th>Delete</th>
    </tr>

    <c:forEach items="${guests}" var="mem">
        <tr>
            <td>${mem.name}</td>
            <td>${mem.phone}</td>
            <td>${mem.email}</td>
            <td><a href="remove?memberName=${mem.name}">Remove</a></td>
        </tr>
    </c:forEach>
</table>
</body>
</html>

```

Stranica poseduje formu u kojoj se unosi nov član. Unosom novog člana, kontrola se prenosi na kontroler preko URL - a [/member/add](#). Kontroler obrađuje zahtev, kreira listu korisnika koju upisuje u model i vraća nazad stranici [memberList.jsp](#) za prikazivanje. Kod stranice sadrži petlju for-each čiji je zadatak da članove iz liste, jednog po jednog, prikazuje u tabeli JSP stranice.

Konačno, moguće je prikazati rezultate urađenog posla. Aplikacija se ponovo prevodi i angažuje na serveru. Izborom linka <http://localhost:8080/Court/member/add> prikazuje se sledeća stranica za dodavanje novog člana, baš kao na sledećoj slici.



Slika 2.2 Dodavanje novog člana - demo [izvor:autor]

## ▼ Poglavlje 3

# Presretanje zahteva

## KONCEPT HANDLER INTERCEPTOR

*Spring MVC podržava upravljanje zahtevima primenom presretača rukovalaca.*

**PROBLEM:** Servlet filteri, definisani, Servlet API-jem, mogu rukovati svakim web zahtevom pre i posle njegovog rukovanja servletom. Ovo su tako zvani **pre-handle** i **post-handle** scenariji rukovanja zahtevima. Neophodno je konfigurisati nešto slično filterima u Spring kontekstu web aplikacije za iskorišćavanje prednosti kontejnerskih alata.

Šta više, ponekad je neophodno rukovati veb zahtevima kojima upravlja *Spring MVC*, po **pre-handle** i **post-handle** scenarijima i manipulirati atributima modela vraćenim od rukovalaca pre nego što su prosleđeni pogledima.

**REŠENJE:** *Spring MVC* podržava ovakvo upravljanje zahtevima primenom *presretača rukovalaca* (**handler interceptors**). Presretači su konfigurisani u kontekstu Spring web aplikacije i mogu da koriste bilo koji kontejnerski alat i da ukažu na bilo koje zрно deklarirano u kontejneru. Presretač može biti registrovan za izvesno URL mapiranje, ta da presreće zahteve mapirane za konkretan URL.

## KREIRANJE PRESRETAČA

*Svaki presretač mora da implementira interfejs **HandlerInterceptor**.*

Svaki presretač mora da implementira interfejs **HandlerInterceptor**, koji obezbeđuje specifične povratne metode **preHandle()**, **postHandle()** i **afterCompletion()**. Prva i druga metoda pozivaju se pre i posle rukovanja zahteva rukovaocem. Druga metoda, takođe, dozvoljava dobijanje pristupa vraćenom objektu **ModelAndView** i omogućava manipulisanje atributima modela. Poslednja metoda se poziva po okončanju obrade svih zahteva, na primer nakon učitavanja pogleda koji prikazuje željeni sadržaj i rezultate obrade MVC aplikacije.

Pretpostavka je da se želi izmeriti vreme rukovanja svakim web zahtevom, svakog rukovaoca i dati dozvola pogledima da prikažu to vreme korisnicima. U tu svrhu se koristi sledeći presretač:

```
/**
 *
 * @author Vlada
 */
```

```
public class MeasurementInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        long startTime = (Long) request.getAttribute("startTime");
        request.removeAttribute("startTime");
        long endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime);
        modelAndView.addObject("handlingTime", endTime - startTime);
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
    }
}
```

## METODE PRESRETAČA

### *Obarada metoda kreiranog presretača.*

U `preHandle()` metodi presretača zabeleženo je početno vreme koje je sačuvano u atributu zahteva (slika 1 - crveno). Ova metoda bi trebalo da vrati vrednost **true**, dozvoljavajući da **DispatcherServlet** nastavi sa rukovanjem zahtevima. U suprotnom, **DispatcherServlet** će podrazumevati da je metoda već rukovala zahtevom i vratiće odgovor direktno korisniku.

```
@Override
public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
    long startTime = System.currentTimeMillis();
    request.setAttribute("startTime", startTime);
    return true;
}
```

Slika 3.1 Početno vreme koje je sačuvano u atributu zahteva [izvor:autor]

Tada, u metodi `postHandle()`, iz atributa zahteva se učitava početno vreme i poredi se sa tekućim. Izračunava se ukupno vreme trajanja rukovanja zahtevom što se prosleđuje u model za prosleđivanje pogledu (slika 2 - plavo). Konačno, ako ne postoji ništa što bi metoda `afterCompletion()` radila, moguće je ostaviti telo ove metode prazno.

```
@Override
public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler,
    ModelAndView modelAndView) throws Exception {
    long startTime = (Long) request.getAttribute("startTime");
    request.removeAttribute("startTime");
    long endTime = System.currentTimeMillis();
    System.out.println(endTime - startTime);
    modelAndView.addObject("handlingTime", endTime - startTime);
}
```

Slika 3.2 Ukupno vreme trajanja rukovanja zahtevom i dodavanje u model [izvor:autor]

Konačno, ako ne postoji ništa što bi metoda **afterCompletion()** radila, moguće je ostaviti telo ove metode prazno.

```
@Override
public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
}
```

Slika 3.3 Prazna metoda afterCompletion() [izvor:autor]

## KREIRANJE PRESRETAČA - PRIMENA ADAPTER KLASA

*Klasa presretača može na pogodniji način da nasledi presretač Adapter klasu.*

Sa implementacijom interfejsa, neophodno je implementirati sve metode čak i u slučaju kad nema potrebe za nekima od njih. Bolji način je nasleđivanje presretač **Adapter** klase.

Po osnovnim podešavanjima, ova klasa implementira sve metode interfejsa **HandlerInterceptor**. U ovom slučaju, moguće je redefinisati samo one metode koje su neophodne za konkretan zadatak.

Sledećim listingom je ilustrovano prethodno izlaganje.

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class MeasurementInterceptor extends HandlerInterceptorAdapter {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        ...
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
```



```
    ...  
}  
}
```

## KONFIGURACIJA PRESRETAČA I KREIRANJE POGLEDA

*Neophodno je izvršiti redefinisavanje konfiguracija dodavanjem konfiguracije za presretače.*

Da bi presretač bio registrovan, neophodno je izvršiti redefinisavanje postojećeg sistema konfiguracionih datoteka (Java klasa). Ovaj zadatak se svodi na implementiranje interfejsa *WebMvcConfigurer* i redefinisavanje njegove metode *addInterceptors()* u konfiguracionoj klasi presretača. Ova metoda daje pristup objektu klase *InterceptorRegistry* koji može biti upotrebljen za dodavanje novih presretača u registar. Sledećim listingom je data konfiguraciona klasa presretača sa jednim registrovanim zrnom prethodno registrovane klase *MeasurementInterceptor.java*.

```
package com.metropolitan.config;  
  
import com.metropolitan.interceptor.MeasurementInterceptor;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
/**  
 *  
 * @author Vlada  
 */  
@Configuration  
public class InterceptorConfiguration implements WebMvcConfigurer {  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry) {  
        registry.addInterceptor(measurementInterceptor());  
    }  
  
    @Bean  
    public MeasurementInterceptor measurementInterceptor() {  
        return new MeasurementInterceptor();  
    }  
}
```

Sada je moguće malo i modifikovati poznatu stranicu *welcome.jsp* za prikazivanje rezultata angažovanja presretača.

```
<%--  
Document    : welcome  
Created on  : 18.09.2018., 12.19.53
```

```
Author      : Vlada
--%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
  <head>
    <title>Welcome</title>
  </head>

  <body>
    <h2>Welcome to Court Reservation System</h2>
    Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd"/>.

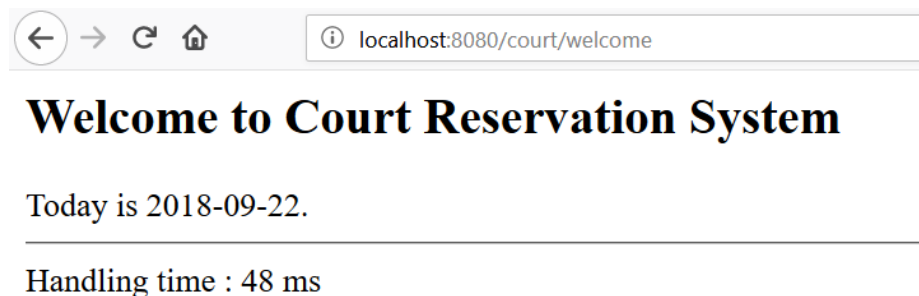
    <hr/>
    Handling time : ${handlingTime} ms

  </body>
</html>
```

Tada je moguće prikazati ovo vreme u *welcome.jsp* za proveru funkcionalnosti presretača. Pozivom:

`http://localhost:8080/court/welcome`

dobija se sledeći rezultat angažovanja presretača:



Slika 3.4 Vreme obrade zahteva - angažovanje presretača [izvor:autor]

## HANDLER INTERCEPTOR - VIDEO MATERIJAL

### *Presretanje zahteva u Spring MVC aplikaciji - video materijal*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

# Podrška za internacionalizaciju

## LOKALIZACIJA

*Neophodno je identifikovati preferiranu lokaciju za svakog korisnika.*

Lokalizacija i internacionalizacija je novina u Java programima koja je uvedena sa Javom 8. Da bi Spring MVC aplikacija imala sposobnost da podrži internacionalizaciju, neophodno je identifikovati preferiranu lokaciju za svakog korisnika i u tom kontekstu prikazivati sadržaj.

Kako okvir Spring MVC nudi rešenje ovog problema? U Spring MVC aplikaciji, lokacija korisnika je određena konceptom rešavača - locale resolver koji mora da implementira interfejs **LocaleResolver**. U Spring MVC postoji nekoliko različitih implementacija, po različitim kriterijumima. Takođe, moguće je kreirati i vlastitog rešavača lokalizacije i izvršiti implementaciju navedenog interfejsa.

Moguće je definisati rešavača registracijom zrna tipa LocaleResolver u kontekstu veb aplikacije. Naziv zrna je neophodno podesiti da bude localeResolver da bi automatski mogao da bude otkriven pomoću **DispatcherServlet**-a. **Moguće je definisati samo jedan rešavač ovog tipa za DispatcherServlet.**

## REŠAVANJE LOKALIZACIJE ATRIBUTOM SESIJE

*Opcija za rešavanje lokalizacije je primena rešavača **SessionLocaleResolver**.*

Pre ulaska u konkretnu problematiku, neophodno je napomenuti da se problem lokalizacije može rešiti i u **zaglavlju HTTP zahteva**. Podrazumevani rešavač, kojeg Spring koristi, je **AcceptHeaderLocaleResolver**. On rešava lokalizaciju proverom accept-language zaglavlja HTTP zahteva.

Druga opcija za rešavanje lokalizacije je primena rešavača **SessionLocaleResolver**. On rešava problem lokalizacije proverom predefinisiranog atributa u sesiji korisnika. Ako atribut sesije ne postoji, rešavač određuje podrazumevanu lokaciju iz **accept-language** HTTP zaglavlja.

```
@Bean
public LocaleResolver localeResolver () {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
}
```

```
localeResolver.setDefaultLocale(new Locale("en"));  
return localeResolver;  
}
```

Moguće je podesiti atribut `defaultLocale` za slučaj da atribut sesije ne postoji.

## REŠAVANJE LOKALIZACIJE KOLAČIĆIMA

*Moguće je koristiti rešavača `CookieLocaleResolver` za rešavanje problema lokalizacije.*

Takođe, moguće je koristiti rešavača `CookieLocaleResolver` za rešavanje problema lokalizacije, proverom kolačića u korisnikovom veb pregledaču. Ukoliko ne postoji odgovarajući kolačić, rešavač određuje podrazumevanu lokaciju iz `accept-language` HTTP zaglavlja.

```
@Bean  
public LocaleResolver localeResolver() {  
    return new CookieLocaleResolver();  
}
```

Kolačić iskorišćen ovi rešavačem, može biti prilagođen podešavanjem vrednosti `cookieName` i `cookieMaxAge`. Poslednja osobina ukazuje na vreme, u sekundama, koliko se kolačić čuva. Vrednost -1 govori da kolačić neće biti validan nakon zatvaranja web čitača.

```
@Bean  
public LocaleResolver localeResolver() {  
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();  
    cookieLocaleResolver.setCookieName("language");  
    cookieLocaleResolver.setCookieMaxAge(3600);  
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));  
    return cookieLocaleResolver;  
}
```

Moguće je podesiti atribut `defaultLocale` za slučaj da kolačić ne postoji u web čitaču.

## PROMENA LOKALNIH PODEŠAVANJA KORISNIKA

*Promena lokalnih podešavanja korisnika može biti obavljena na alternativni način.*

Promena lokalnih podešavanja korisnika može biti obavljena eksplicitno pozivom `LocaleResolver.setLocale()`, ali i primenom `LocaleChangeInterceptor` presretača na mapiranje rukovalaca. Ovaj presretač detektuje da li je specijalni parametar prisutan u HTTP zahtevu. Parametar može biti prilagođen osobinom `paramName` ovog presretača. Ukoliko je ovakav parametar prisutan u zahtevu, presretač će promeniti lokalna podešavanja korisnika u zavisnosti od vrednosti parametra.

```
/**
 *
 * @author Vlada
 */
@Configuration
public class I18NConfiguration implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("language");
        return localeChangeInterceptor;
    }

    @Bean
    public CookieLocaleResolver localeResolver() {
        CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
        cookieLocaleResolver.setCookieName("language");
        cookieLocaleResolver.setCookieMaxAge(3600);
        cookieLocaleResolver.setDefaultLocale(new Locale("en"));
        return cookieLocaleResolver;
    }
}
</beans>
```

Sada lokalna podešavanja korisnika mogu biti promenjena primenjujući bilo koji URL sa parametrom **language**.

```
http://localhost:8080/court/welcome?language=en_US
http://localhost:8080/court/welcome?language=sr
```

Na kraju, moguće je prikazati HTTP odgovor lokalnog objekta u *welcome.jsp* za proveru lokalne konfiguracije presretača:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
  <head>
    <title>Welcome</title>
  </head>

  <body>
    <h2>Welcome to Court Reservation System</h2>
    Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd"/>.

    <hr/>
    Handling time : ${handlingTime} ms
```

```
<br/>
  Locale : ${pageContext.response.locale}

</body>
</html>
```

## Welcome to Court Reservation System

Today is 2018-09-23.

Handling time : 89 ms  
Locale : en\_US

Slika 4.1 Prihvatanje lokalnih podešavanja [izvor:autor]

## RAD SA LOKALNO OSETLJIVIM PORUKAMA - REGISTRACIJA ZRNA

### *Obrada rada sa lokalnim porukama u Spring MVC radnom okviru.*

Kada se kreira internacionalna veb aplikacija, posebno je poželjno prikazivanje stranica na jeziku kojeg koriste korisnici aplikacije. Ovde je posebno važno izbegavanje kreiranje većeg broja veb stranica koje prikazuju isti sadržaj ali na različitim jezicima.

*Spring MVC* nudi rešenje ovog problema upotrebom izvora poruka (eng. *message source*) koji se zasniva na implementaciji interfejsa *MessageSource*. Tada JSP stranice dobijaju mogućnost korišćenja specifičnih `<spring:message>` tagova za obradu koda koji se odnosi na navedene poruke.

Prvi korak koji je neophodno učiniti jeste registrovanje zrna tipa *MessageSource* u kontekstu kreirane veb aplikacije. Da bi ovo zrno moglo da bude automatski otkriveno od strane centralne komponente *DispatcherServlet*, njemu mora biti dodeljen naziv *messageSource*. Implementacija *ResourceBundleMessageSource* će omogućiti primenu poruka iz različitih izvora za različite lokacije.

Dodaćemo sledeći kod u prethodno kreiranu datoteku *118NConfiguration.java* sa ciljem učitavanja resursa sa osnovnim nazivom "*messages*".

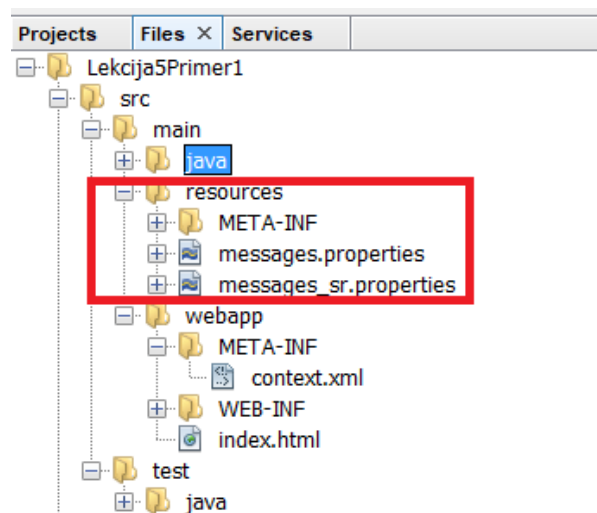
```
@Bean
public MessageSource messageSource() {
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
    messageSource.setBasename("messages");
    return messageSource;
}
```

## KREIRANJE EKSTERNIH RESURSA ZA PORUKE

*JSP stranice čitaju lokalne poruke iz datoteka koje se čuvaju kao resursi.*

JSP stranice čitaju lokalne poruke iz datoteka koje se čuvaju kao resursi. U daljim izlaganjima će posebno biti posvećena pažnja ovim tzv. datotekama osobina. Njihova uloga je velika u Spring MVC aplikacijama, a ovde ćemo fokus biti na radu sa lokalnim porukama.

U folderu `main/resources` biće kreirane dve tekstualne datoteke: `messages.properties` i `messages_sr.properties`. Prava datoteka će omogućiti prikazivanje sadržaja, u stranici `welcome.jsp`, na Engleskom jeziku, a druga na Srpskom. Sledećom slikom je prikazana lokacija ovih datoteka u kreiranom projektu.



Slika 4.2 Lokacija "property" datoteka [izvor:autor]

Sada je moguće kreirati datoteke popuniti odgovarajućim sadržajem. U datoteku `messages.properties` biće upisano sledeće:

```
welcome.title=Welcome  
welcome.message=Welcome to Court Reservation System
```

U datoteku `messages_sr.properties` biće upisano sledeće:

```
welcome.title=Dobrodošli  
welcome.message=Dobrodošli na sistem za rezervaciju terena
```

## MODIFIKACIJA JSP

*Neophodno je modifikovati JSP stranicu da koristi Spring lokalne tagove.*

U poslednjem koraku je neophodno izvršiti modifikaciju JSP datoteke koja će da prikazuje lokalizovani sadržaj. Dobro poznata stranica *welcome.jsp* sada dobija nov oblik prikazan sledećim listingom.

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>
  <head>
    <title><spring:message code="welcome.title" text="Welcome" /></title>
  </head>

  <body>
    <h2><spring:message code="welcome.message"
      text="Welcome to Court Reservation System"/></h2>

    Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd"/>.

    <hr/>
    Handling time : ${handlingTime} ms

    <br/>
    Locale : ${pageContext.response.locale}

  </body>
</html>
```

Da bi ovakva JSP stranica mogla da koristi tagove *<spring:message>* neophodno je dodati na sam početak stranice (slično kao importovanje u Javi) odgovarajući prostor naziva (eng. *name space*) :

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

U određenim slučajevima JSP stranica zahteva i sledeći kod:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

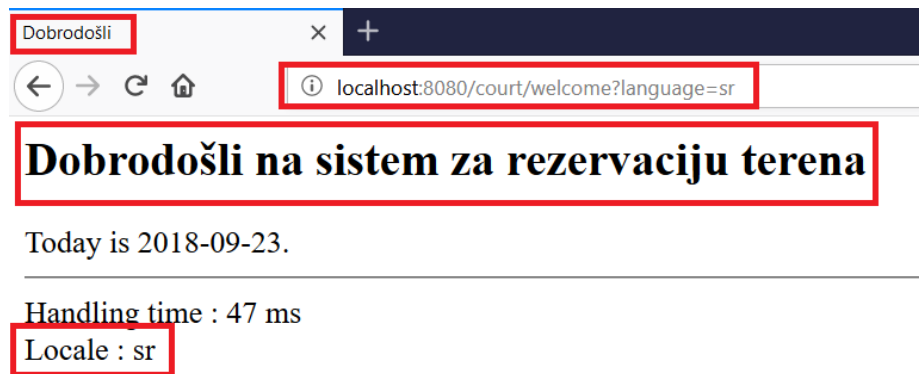
da bi uspešno obradila tagove sa prefiksom spring.

Sve je uspešno odrađeno, aplikacija je ponovo prevedena i pokrenuta. U veb pregledaču se navodi link:

```
http://localhost:8080/court/welcome?language=sr
```

Učitava se stranica *welcome.jsp* i prikazuje sledeći sadržaj:





Slika 4.3 JSP sa lokalizovanim sadržajem [izvor:autor]

## ZADATAK ZA SAMOSTALNI RAD 1

### *Pokušajte sami*

1. U prilogu ovog objekta učenja se nalazi potpuno urađen zadatak;
2. Pre nego ga preuzmete i testirate, vodeći se smernicama iz ovog objekta učenja, pokušajte samostalno da kompletirate primer.

## ▼ Poglavlje 5

# Rešavanje pogleda pomoću naziva

## DEFINISANJE PROBLEMA

*Spring MVC ima nekoliko ViewResolver implementacija za rešavanje pogleda za različite strategije.*

PROBLEM: Nakon što je rukovalac završio rukovanje zahtevom, on vraća naziv logičkog pogleda. U tom slučaju **DispatcherServlet** šalje kontrolu ka šablonu pogleda i informacija je ugrađena u pogled. Ova procedura je označena kao rešavanje pogleda. Cilj je kreiranje strategije za **DispatcherServlet** za rešavanje pogleda po njihovim logičkim nazivima.

REŠENJE: U *Spring MVC* aplikaciji, pogledi se rešavaju jednim ili pomoću više zrna rešavača (**resolvers**) deklariranih u kontekstu web aplikacije. Ova zrna moraju da implementiraju interfejs **ViewResolver** da bi ih **DispatcherServlet** automatski prepoznao. *Spring MVC* podržava nekoliko **ViewResolver** implementacija za rešavanje pogleda kroz različite strategije.

## REŠAVANJE POGLEDA NA OSNOVU NAZIVA ŠABLONA I LOKACIJE

*Osnovna strategija za rešavanje pogleda je njihovo direktno mapiranje u naziv šablona i lokaciju*

Osnovna strategija za rešavanje pogleda je njihovo direktno mapiranje u naziv šablona i lokaciju. Rešavač **InternalResourceViewResolver** mapira naziv svakog pogleda u direktorijum aplikacije putem prefiks i sufiks deklaracije. Za registrovanje **InternalResourceViewResolver**, neophodno je deklarirati zrna ovog tipa u kontekstu aplikacije.

```
@Bean
public InternalResourceViewResolver viewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

Na primer, **InternalResourceViewResolver** rešava nazive pogleda *welcome* i *reservationQuery* na sledeći način:

```
welcome ---> ' /WEB-INF/jsp/welcome.jsp
reservationQuery ---> ' /WEB-INF/jsp/reservationQuery.jsp
```

## REŠAVANJE POGLEDA IZ XML KONFIGURACIONE DATOTEKE

*Sledeća strategija rešavanja pogleda jeste njihovo deklarisanje kao Spring zrna.*

Sledeća strategija rešavanja pogleda jeste njihovo deklarisanje kao Spring zrna i njihovo rešavanje pomoću naziva tih zrna. Zrna pogleda mogu da se deklariraju u istom konfiguracionom fajlu kao i kontekst aplikacije, ali je bolje razdvojiti u zasebne konfiguracione datoteke. Po osnovnim podešavanjima, **XmlViewResolver** učitava zrna pogleda sa lokacije [/WEB-INF/views.xml](#), ali lokacija može da bude redefinisana osobinom **location**.

```
@Configuration
public class ViewResolverConfiguration implements WebMvcConfigurer,
ResourceLoaderAware {
    private ResourceLoader resourceLoader;
    @Bean
    public ViewResolver viewResolver() {
        XmlViewResolver viewResolver = new XmlViewResolver();
        viewResolver.setLocation(resourceLoader.getResource("/WEB-INF/
court-views.xml"));
        return viewResolver;
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader=resourceLoader;
    }
}
```

Moguće je primetiti implementaciju interfejsa **ResourceLoaderAware** pri čemu je neophodno učitavanje resursa koji odgovara XML dokumentu. Kada je reč o konverziji stringa (XML dokumenta) u resurs, Spring će obaviti ceo posao umesto nas. Međutim, neophodno je još malo posla.

U konfiguracionom fajlu [court-views.xml](#) moguće je deklarirati svaki pogled kao normalno Spring zrno podešavanjem naziva klase i osobina.

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <bean id="welcome"
        class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/welcome.jsp" />
    </bean>
```

```
<bean id="reservationQuery"
      class="org.springframework.web.servlet.view.JstlView">
  <property name="url" value="/WEB-INF/jsp/reservationQuery.jsp" />
</bean>

<bean id="welcomeRedirect"
      class="org.springframework.web.servlet.view.RedirectView">
  <property name="url" value="welcome" />
</bean>
</beans>
```

## REŠAVANJE POGLEDA IZ RESOURCE BUNDLE

*Moguće je deklarirati zrna pogleda iz kutije resursa.*

Kao dodatak XML konfiguracionom fajlu, moguće je deklarirati zrna pogleda iz skladišta resursa - **resource bundle**. **ResourceBundleViewResolver** učitava zrna pogleda iz skladišta resursa iz korena putanje klasa.

```
@Bean
public ResourceBundleViewResolver viewResolver() {
    ResourceBundleViewResolver viewResolver = new ResourceBundleViewResolver();
    viewResolver.setBasename("court-views");
    return viewResolver;
}
```

Ako se specificira **views** kao osnovni naziv za **ResourceBundleViewResolver**, podrazumevano skladište resursa je **views.properties**. U ovom skladištu resursa, moguće je definisati zrna pogleda u formatu osobina. Ovaj tip deklaracije je ekvivalentan XML deklaraciji zrna.

```
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp
reservationQuery.(class)=org.springframework.web.servlet.view.JstlView
reservationQuery.url=/WEB-INF/jsp/reservationQuery.jsp
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome
```

## REŠAVANJE POGLEDA SA VIŠE REŠAVAČA

*U praksi, često je neophodno kombinovati različite strategije rešavanja.*

Ako postoji puno pogleda u web aplikaciji, često je nedovoljno izabrati samo jednu strategiju rešavanja. Tipično, **InternalResourceViewResolver** može rešiti većinu internih JSP pogleda, ali obično postoje i ostali tipovi pogleda koje je neophodno rešiti sa

**ResourceBundleViewResolver**. O ovom slučaju, neophodno je kombinovati ove strategije rešavanja pogleda.

```
@Bean
public ResourceBundleViewResolver viewResolver() {
    ResourceBundleViewResolver viewResolver = new ResourceBundleViewResolver();
    viewResolver.setOrder(0);
    viewResolver.setBasename("court-views");
    return viewResolver;
}

@Bean
public InternalResourceViewResolver internalResourceViewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setOrder(1);
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

Kada se bira više strategija istovremeno, veoma je bitno definisati prioritet rešavanja. U ovu svrhu, moguće je podesiti osobinu **order** zrna rešavača pogleda. Manja vrednost osobine znači viši prioritet. Najniži prioritet se dodeljuje za **InternalResourceViewResolver** iz razloga što on uvek rešava pogled, bilo da on postoji ili ne. To znači da ostali rešavači neće dobiti šansu da budu angažovani ukoliko imaju niži prioritet nego **InternalResourceViewResolver**. Navedeno je prikazano u priloženom listingu u linijama koda 4 i 12.

Sada bi skladište resursa *views.properties* trebalo da sadrži samo poglede koji ne mogu biti rešeni sa **InternalResourceViewResolver** (na primer, preusmereni pogledi).

```
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome
```

## PREFIKS PREUSMERAVANJA

*Pokazana je uloga prefiksa **redirect** za preusmeravanje pogleda.*

Ako postoji **InternalResourceViewResolver** podešen u kontekstu web aplikacije, on može da reši preusmerene poglede primenom prefiksa **redirect** u nazivu pogleda. Tada je ostatak naziva pogleda tretiran kao preusmereni URL. Na primer, naziv pogleda **redirect:welcome** pokreće preusmeravanje ka relativnom URL - u **welcome**. Takođe, moguće je specificirati apsolutni URL u nazivu pogleda.

## ZADATAK ZA SAMOSTALNI RAD 2

*Pokušajte sami*

1. Ponovo iskoristite primer koji je priložen u prethodnom izlaganju kao Shared Resources aktivnost;
2. Kreirajte *views.property* datoteku i kreirajte neophodne izmene u datoteci *court-servlet.xml*;
3. Proverite da li je neophodno proširiti *pom.xml* novim zavisnostima;
4. Vodeći se smernicama iz ovog objekta učenja, pokušajte samostalno da kompletirate primer.

## ▼ Poglavlje 6

# Mapiranje izuzetaka u pogledu

## DEFINISANJE PROBLEMA

*Kada se javi nepoznati izuzetak, server aplikacije obično prikaže korisniku stack trace izuzetak.*

Kada se javi nepoznati izuzetak, server aplikacije obično prikaže korisniku **stack trace** izuzetak. Korisnici nemaju ništa sa izuzetkom **stack trace** i počinju da se žale da aplikacija nije **user-friendly**. U datoteci **web.xml** moguće je podesiti prikazivanje "prijateljskih" **JSP** stranica u slučaju **HTTP** greške ili klase izuzetka. Međutim, savremeni Spring pristup nudi šira rešenja

U **Spring MVC** aplikaciji, moguće je registrovati jedno ili više zrna rešavača izuzetaka u kontekstu web aplikacije, za rešavanje *neuhvaćenih* izuzetaka. Ova zrna moraju da implementiraju interfejs **HandlerExceptionResolver** da bi ih **DispatcherServlet** automatski otkrio. **Spring MVC** dolazi sa jednostavnim rešavačem izuzetaka, za mapiranje svake kategorije izuzetaka u pogled.

Uvodi se konkretan primer. Pretpostavka je da servis za rezervaciju terena izbacuje sledeći izuzetak ako rezervacija nije dostupna:

```
package com.metropolitan.service;
...
public class ReservationNotAvailableException extends RuntimeException {
    private String courtName;
    private Date date;
    private int hour;

    // konstruktori, setteri i getteri
    ...
}
```

Za rešavanje izuzetka moguće je kreirati vlastiti rešavač koji implementira **HandlerExceptionResolver** interfejs. Obično, omogućava se prikazivanje različitih kategorija izuzetaka na različitim **error** stranicama. **Spring MVC** dolazi sa **SimpleMappingExceptionResolver** rešavačem za podešavanje mapiranja izuzetaka u kontekstu web aplikacije. Na primer, moguće je registrovati sledeći rešavač izuzetaka u kontekstu aplikacije:

```
@Override
public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
```

```
exceptionResolvers) {
    exceptionResolvers.add(handlerExceptionResolver());
}

@Bean
public HandlerExceptionResolver handlerExceptionResolver() {
    Properties exceptionMapping = new Properties();
    exceptionMapping.setProperty(
        ReservationNotAvailableException.class.getName(), "reservationNotAvailable");
    SimpleMappingExceptionResolver exceptionResolver = new
SimpleMappingExceptionResolver();
    exceptionResolver.setExceptionMappings(exceptionMapping);
    exceptionResolver.setDefaultErrorView("error");
    return exceptionResolver;
}
```

Ovde je definisan naziv logičkog pogleda *reservationNotAvailable* za *ReservationNotAvailableException*. Moguće je dodati bilo koji broj klasa izuzetaka koristeći iskaz *exceptionMapping.setProperty()*. Na kraju, osobina *defaultErrorView* je upotrebljena za definisanje podrazumevanog pogleda pod nazivom *error*.

## JSP STRANICA IZUZETKA

*Informacija o nedostupnosti rezervacije prikazuje se odgovarajućom JSP stranicom.*

Ako je *InternalResourceViewResolver* podešen u kontekstu aplikacije, sledeća *reservationNotAvailable.jsp* stranica će se prikazati ukoliko rezervacija nije dostupna:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
<title>Reservation Not Available</title>
</head>
<body>
Your reservation for ${exception.courtName} is not available on
<fmt:formatDate value="${exception.date}" pattern="yyyy-MM-dd" /> at
${exception.hour}:00.
</body>
</html>
```

Na stranici greške, instanci izuzetka moguće je pristupiti pomoću varijable *\${exception}*, pa je korisniku moguće prikazati više detalja u vezi sa izuzetkom. Dobra je praksa definisanje podrazumevane stranice sa greškom za bilo koji nepoznati izuzetak. U tu svrhu je moguće iskoristiti *defaultErrorView* za definisanje podrazumevanog pogleda za greške. Tada je moguće kreirati pogled *error.jsp* :

```
<html>
<head>
```



```
<title>Error</title>
</head>
<body>
An error has occurred. Please contact our administrator for details.
</body>
</html>
```

## ZADATAK ZA SAMOSTALNI RAD 4

### *Pokušajte sami*

1. Ponovo iskoristite primer koji je priložen u prethodnom izlaganju kao Shared Resources aktivnost;
2. Omogućite rad sa stranicama za upravljanje izuzecima vodeći se izlaganjem priloženim u ovom objektu učenja;
3. Proverite da li je neophodno proširiti [pom.xml](#) novim zavisnostima;
4. Vodeći se smernicama iz ovog objekta učenja, pokušajte samostalno da kompletirate primer.

## ▼ Poglavlje 7

# Pridruživanje vrednosti u kontroleru

## ANOTACIJA @VALUE

*Anotacija @Value je moguće koristiti sa SpEL jezikom u kontekstu aplikacije.*

PROBLEM: Sa kreiranjem kontrolera, nastoji se zaobići **hard - code** vrednosti polja. Umesto toga, moguće je dodeliti vrednost prisutnu u zrnju ili datoteci sa osobinama (na primer, *message.properties*).

REŠENJE: Anotacija **@Value** dozvoljava kontrolerovnim poljima da budu dodeljena primenom jezika *Spring ExpressionLanguage (SpEL)*. Ovu anotaciju je moguće koristiti zajedno sa SpEL zrnima upita, prisutnim u kontekstu aplikacije, i tako dobiti vrednosti koje će pomoći kod inicijalizacije polja kontrolera.

Na primer, postoji jednostavan kontroler čiji je zadatak formiranje *About* stranice, po uzoru na sledeći JSP:

```
<html>
<head>
<title>About</title>
</head>
<body>
<h2>Court Reservation System</h2>
<table>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
</table>
</body>
</html>
```

Dodavanje administratorovog e-maila na ovoj stranici, kao kontakta, jeste ustaljena praksa. Ukoliko dođe do promene adrese administratora, ona bi morala da bude promenjena na ovoj, ali i na svim ostalim stranicama na kojima se nalazi. Zato je moguće centralizovati ovu informaciju i izmenu vršiti na jednom mestu, na primer u datoteci *message.properties*. Upravo će sledećom instrukcijom biti ugrađen administratorov e-mail u *message.properties* datoteku:

```
admin.email=reservation@domain.com
```

Sada je moguće modifikovati `about.jsp` pogled da prikazuje email atribut prosleđen kontrolerom kao atribut modela.

```
<html>
<head>
<title>About</title>
</head>
<body>
<h2>Court Reservation System</h2>
<table>
...
<tr>
<td>Email:</td>
<td><a href="mailto:${email}">${email}</a></td>
</tr>
</table>
</body>
</html>
```

## ANOTACIJA @VALUE - KREIRANJE ODGOVARAJUĆEG KONTROLERA

*Neophodno je kreirati odgovarajući kontroler koji će email atribut proslediti pogledu.*

Nakon kreiranja stranice `about.jsp`, na lokaciji `/WEB-INF/jsp/`, neophodno je kreirati odgovarajući kontroler koji će email atribut proslediti pogledu. Sledeći `AboutController` pridružuje `email` polje iz datoteke `message.properties` koristeći anotaciju `@Value`.

```
package com.metropolitan.controller;
...
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Value;

@Controller
public class AboutController

    @Value("#{ messageSource.getMessage('admin.email',null,'en')}")
    private String email;

    @RequestMapping("/about")
    public String courtReservation(Model model) {
        model.addAttribute("email", email);
        return "about";
    }
}
```

Vrednost dodeljena anotaciji `@Value` predstavlja SpEL izraz. U ovom slučaju `messageSource` predstavlja vrednost zrna `org.springframework.context.support.ResourceBundleMessageSource` koje je deklarirano u kontekstu aplikacije za pristup fajlu `message.properties`. U referencu zrna je dodata metoda `getMessage('admin.email',null,'en')` koja, kada se pozove sa ovim parametrima, vraća vrednost osobine `admin.email`. Kroz `@Value` anotaciju ova vrednost je automatski dodeljena polju `email`.

Sledeće, moguće je pronaći jedinu kontrolerovu rukovalac metodu koja definiše *Model* objekat kao svoj ulazni parametar. Unutar metode, polje `email` je pridruženo atributu modela sa imenom `email`, tako da kasnije može biti referencirano u odgovarajućem pogledu. Povratna vrednost `about` označava naziv logičkog pogleda, koje je, u ovom slučaju, u vezi sa `about.jsp`. Konačno, zahvaljujući anotaciji `@RequestMapping("/about")` metode rukovaoca, moguće je pristupiti kontroleru na sledeći način:

```
http://localhost:8084/court/about
```

## ZADATAK ZA SAMOSTALNI RAD 5

### *Pokušajte sami*

1. Ponovo iskoristite primer koji je priložen u prethodnom izlaganju kao Shared Resources aktivnost;
2. Implementirajte `@Value` anotacije na način prikazan u izlaganju ovog objekta učenja;
3. Proverite da li je neophodno proširiti `pom.xml` novim zavisnostima;
4. Vodeći se smernicama iz ovog objekta učenja, pokušajte samostalno da kompletirate primer.

## ▼ Poglavlje 8

# Upravljanje formama pomoću kontrolera

## DEFINISANJE PROBLEMA

*Upravljanje formom može biti kompleksan zadatak.*

PROBLEM: U web aplikacijama se veoma često radi sa formama. Kontroler forme mora da prikaže formu korisniku ali takođe i da rukuje potvrđivanjem forme. Upravljanje formom može biti kompleksan zadatak.

REŠENJE: Kada korisnik komunicira sa formom, on zahteva podršku za dve operacije kontrolera: upit kontroleru da prikaže formu na osnovu [HTTP GET](#) zahteva, i kada je forma potvrđena, [HTTP POST](#) zahtev je napravljen da rukuje proverama ili poslovnom procesiranju podataka prikazanih u formi. Ako je formom uspešno rukovano, ona će formirati odgovarajući pogled korisniku. U suprotnom, ponovo će biti kreiran pogled forme sa informacijama o greškama.

Nastavlja se sa primerom. Pretpostavka je da se želi obaviti rezervacija terena popunjavanjem odgovarajuće forme. Za bolje shvatanje upravljanja podacima pomoću kontrolera, biće predatavljen prvo pogled kontrolera (na primer, konkretna forma).

## KREIRANJE POGLEDA FORME

*Forma se oslanja na Spring biblioteku tagova.*

Kreira se pogled forme **reservationForm.jsp**. Forma se oslanja na Spring biblioteku tagova jer se pojednostavljuje povezivanje podataka, prikazivanje poruka sa greškama ili ponovno prikazivanje originalnih vrednosti, koje je uneo korisnik, u slučaju pojave grešaka.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<html>
<head>
    <title>Reservation Form</title>
    <style>
        .error {
            color: #ff0000;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <form:form name="reservationForm" method="POST">
        <form:input type="text" name="location" value="New York" />
        <form:input type="text" name="date" value="2010-01-01" />
        <form:input type="button" value="Book" />
    </form:form>
</body>
</html>
```

```

    </style>
</head>
<body>

<form:form method="post" modelAttribute="reservation">
    <form:errors path="*" cssClass="error"/>

    <td>Court Name</td>
    <td><form:input path="courtName"/></td>
    <td><form:errors path="courtName" cssClass="error"/></td>

    <tr>
        <td>Date</td>
        <td><form:input path="date"/></td>
        <td><form:errors path="date" cssClass="error"/></td>
    </tr>
    <tr>
        <td>Hour</td>
        <td><form:input path="hour"/></td>
        <td><form:errors path="hour" cssClass="error"/></td>
    </tr>

    <tr>
        <td colspan="3"><input type="submit"/></td>
    </tr>

</form:form>
</body>
</html>

```

Spring element `<form:form>` deklarira dva atributa: `method="post"` ukazuje na formu koja izvodi *HTTP POST* zahtev nakon potvrde i `modelAttribute="reservation"` ukazuje da su podaci forme obuhvaćeni modelom pod nazivom **reservation**. Dalje, `<form:errors>` tag definiše lokaciju gde se čuvaju greške u slučaju da se forma ne pridržava određenih pravila diktiranih kontrolerom. Tagovi `<form:input>` su u vezi sa osobinama koje odgovaraju **modelAttribute** primenom atributa putanje **path**. Ove osobine prikazuju korisniku originalne vrednosti polja. Moraju da se koriste unutar `<form:form>` taga koji definiše formu koja se povezuje sa **modelAttribute** na osnovu naziva.

Konačno, tu je i standardni HTML tag `<input type="submit" />` koji generiše *Submit* dugme i inicira slanje podataka serveru.

U slučaju da su forma i podaci procesirani uspešno, neophodno je obezbediti pogled koji će obavestiti korisnika o uspešno obavljenoj rezervaciji (na primer, *reservationSuccess.jsp*).

```

<% --
    Document    : reservationSuccess
    Created on  : 25.09.2018., 09.27.06
    Author      : Vlada
-- %>

```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <title>Reservation Success</title>
  </head>
  <body>
    Your reservation has been made successfully.
  </body>
</html>
```

## OBRADA GREŠAKA

*Takođe, moguće je pojavljivanje i grešaka, ako su nekorektne vrednosti unete u formu.*

Takođe, moguće je pojavljivanje i grešaka, ako su nekorektne vrednosti unete u formu. Na primer, datum nije unet u traženom formatu, a kontroler je dizajniran da odbije ovakvu vrednost polja. Kontroler će tada generisati listu selektivnih kodova grešaka, za svaku grešku koja će biti vraćena iz pogleda forme, vrednosti koje će biti smeštene u tagu `<form:errors>`.

Na primer, za nekorektan unos polja **date**, kontroler je generisao sledeće kodove grešaka:

```
typeMismatch.command.date
typeMismatch.date
typeMismatch.java.util.Date
typeMismatch
```

Ukoliko postoji definisan, `ResourceBundleMessageSource`, moguće je uključiti ove greške u datoteku resursa za odgovarajuću lokalizaciju:

```
typeMismatch.date=Invalid date format
typeMismatch.hour=Invalid hour format
```

## OBRADA SERVISA FORME

*Razjašnjeni elementi koji interaguju sa kontrolerom - pogledi forme i klasa servisa rezervacije.*

Ovde se ne radi o kontroleru već o servisu kojeg kontroler koristi za obradu podataka forme za rezervaciju. U nastavku će u postojeći interfejs `ReservationService` biti dodata nova metoda `make()`.

```
package com.metropolitan.service;

import com.metropolitan.domain.Reservation;
import java.util.List;
```

```
/**
 *
 * @author Vlada
 */
public interface ReservationService {

    public List<Reservation> query(String courtName);

    void make(Reservation reservation) throws ReservationNotAvailableException;
}
```

Zatim će ova metoda biti implementirana dodavanjem objekta *Reservation* kao stavke liste koja čuva rezervacije. U slučaju dupliranja rezervacija biće izbačen izuzetak: ***ReservationNotAvailableException***.

```
package com.metropolitan.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    @Override
    public void make(Reservation reservation) throws
ReservationNotAvailableException {
        long cnt = reservations.stream()
            .filter(made -> Objects.equals(made.getCourtName(),
reservation.getCourtName()))
            .filter(made -> Objects.equals(made.getDate(),
reservation.getDate()))
            .filter(made -> made.getHour() == reservation.getHour())
            .count();
        if (cnt > 0) {
            throw new ReservationNotAvailableException(reservation
                .getCourtName(), reservation.getDate(), reservation
                .getHour());
        } else {
            reservations.add(reservation);
        }
    }
}
reservations.add(reservation);
}
```

Sada su razjašnjena oba elementa koji interaguju sa kontrolerom - pogledi forme i klasa servisa rezervacije.

## KREIRANJE KONTROLERA FORME

*Sada je neophodno uvesti u analizu odgovarajući kontroler.*



Kontroler koji se koristi za upravljanje formama iz praktičnih razloga koristi iste anotacije koje su već korišćene u prethodnom izlaganju.

```
/**
 *
 * @author Vlada
 */
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {

    private final ReservationService reservationService;

    @Autowired
    public ReservationFormController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(Model model) {
        Reservation reservation = new Reservation();
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") Reservation reservation,
        BindingResult result, SessionStatus status) {
        reservationService.make(reservation);
        return "reservationSuccess";
    }
}
```

Kontroler počinje primenom standardne **@Controller** anotacije i koristi, takođe i **@RequestMapping** za dozvolu pristupa kontroleru pomoću:

```
http://localhost:8080/court/reservationForm
```

Unošenjem ovog URL -a u web čitač, upućuje se **HTTP GET** zahtev aplikaciji. Pokreće se metoda **setupForm** koja definiše **Model** objekat kao ulazni parametar čiji je zadatak slanje podataka pogledu.

Unutra metode je kreiran prazan objekat **Reservation** koji je dodat kao atribut kontrolerovom objektu tipa **Model**. Tada kontroler vraća tok izvršavanja pogledu **reservationForm** koji će, u ovom slučaju, biti rešen formom **reservationForm.jsp**. Pored dodavanja objekta **Reservation**, još jedan važan aspekt predstavlja primena anotacije **@SessionAttributes("reservation")**. Budući da mogu da se pojave greške, može doći i do gubljenja podataka koje je do sad obezbedio korisnik. Da bi se to sprečilo, **@SessionAttributes** je korišćen za čuvanje reservation polja tokom korisnikove sesije.

Nakon popunjavanja svih potrebnih polja forme i potvrđivanjem, pokreće se zahtev **HTTP POST** koji poziva metodu **submitForm** koja ima tri deklarirana polja za unos: **@ModelAttribute("reservation") Reservation reservation** - ukazuje na **reservation** objekat, objekat **BindingResult** - koji ukazuje na nove podatke koje je korisnik uneo i **SessionStatus** objekat za slučaj da je neophodno pristupiti sesiji korisnika. Jedina operacija koju metoda izvodi je **reservationService.make(reservation)**, kojom se poziva servis rezervacije primenom tekućeg stanja objekta **reservation**.

Konačno, metoda vraća pogled pod nazivom **reservationSuccess**.

## INICIJALIZACIJA OBJEKTA ATRIBUTA MODELA

*Još uvek nedostaju dva polja za kreiranje kompletnog objekta rezervacije.*

Forma je kreirana da omogući korisnicima obavljanje rezervacija. Ako se analizira klasa domena **Reservation**, moguće je primetiti da još uvek nedostaju dva polja za kreiranje kompletnog objekta rezervacije. Jedno od tih polja je **player**, koje odgovara objektu **Player**. Po definiciji klase **Player**, odgovarajući objekti sadrže polja **name** i **phone**. Postavlja se pitanje da li **player** može da se ugradi u formu ili kontroler? Prvo će biti analiziran pogled:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<html>
  <head>
    <title>Reservation Form</title>
    <style>
      .error {
        color: #ff0000;
        font-weight: bold;
      }
    </style>
  </head>

  <form:form method="post" modelAttribute="reservation">
    <form:errors path="*" cssClass="error"/>

    ****
    <tr>
      <td>Player Name</td>
      <td><form:input path="player.name"/></td>
      <td><form:errors path="player.name" cssClass="error"/></td>
    </tr>
    <tr>
      <td>Player Phone</td>
      <td><form:input path="player.phone"/></td>
      <td><form:errors path="player.phone" cssClass="error"/></td>
    </tr>
```

```
<tr>
    <td>Sport Type</td>
    <td>
        <form:select path="sportType" items="${sportTypes}"
                    itemValue="id" itemLabel="name"/>
    </td>
    <td><form:errors path="sportType" cssClass="error"/></td>
</tr>
<tr>
    <td colspan="3"><input type="submit"/></td>
</tr>

</form:form>

</html>
```

Kao što je moguće primetiti angažovana su dva nova `<form:input>` taga koja su obavila postavljeni zadatak. Sada je neophodno, u ovom svetlu, modifikovati kontroler. Kroz kontroler `HTTP GET` metoda vraća prazan objekat `Reservation` poslednjem pogledu. Vrednost za `player` je `null`, a to će izazvati izuzetak prilikom kreiranja forme. Za rešenje ovog problema, neophodno je inicijalizovati prazan objekat `Player` i pridružiti ga objektu `Reservation` koji je vraćen pogledu.

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(
    @RequestParam(required = false, value = "username") String username,
    Model model) {
    Reservation reservation = new Reservation();
    reservation.setPlayer(new Player(username, null));
    model.addAttribute("reservation", reservation);
    return "reservationForm";
}
```

U ovom slučaju, nakon kreiranja praznog objekta `Reservation`, metoda `setPlayer()` je iskorišćena da mu pridruži prazan `Player` objekat. Dalje, kreiranje objekta `Person` oslanja se na vrednost `username`. Izvesna vrednost je primljena kao ulazna vrednost anotacije `@RequestParam` koja je takođe dodata metodi rukovaocu. Nakon ovoga, `Player` objekat može da bude kreiran sa specifičnom vrednošću za `username` prosleđenom kao parametar zahteva i ugrađenom u formu. Na primer, prosleđen je sledeći zahtev formi:

```
http://localhost:8080/court/reservationForm?username=Vlada
```

Ovo dozvoljava rukovalac metodi da uzme parametar `username` za kreiranje objekta `Player`, puneci `username` polje forme vrednošću `Vlada`. Primećuje se još da `@RequestParam` anotacija za `username` parametar koristi osobinu `required=false`, a to omogućava da zahtev forme bude obrađen čak i ako takav parametar zahteva ne postoji.

## OBEZBEĐIVANJE PODATAKA REFERENCI FORME

*Od kontrolera se može tražiti da obezbedi izvesne tipove podataka referenci za formu.*

Kada se od kontrolera forme zatraži da kreira pogled forme, od njega se može tražiti da obezbedi izvesne tipove podataka referenci za formu (na primer stavke za prikazivanje u HTML selekciji). Pretpostavka je da se želi igraču dozvoliti izbor tipa sporta prilikom rezervacije terena - a to može biti poslednje traženo polje za klasu **Reservation**.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<html>
    <head>
        <title>Reservation Form</title>
        <style>
            .error {
                color: #ff0000;
                font-weight: bold;
            }
        </style>
    </head>

    <form:form method="post" modelAttribute="reservation">
        <form:errors path="*" cssClass="error"/>

        <tr>

            <td>Court Name</td>
            <td><form:input path="courtName"/></td>
            <td><form:errors path="courtName" cssClass="error"/></td>
        </tr>

        ***
        <tr>
            <td>Sport Type</td>
            <td>
                <form:select path="sportType" items="${sportTypes}"
                    itemValue="id" itemLabel="name"/>
            </td>
            <td><form:errors path="sportType" cssClass="error"/></td>
        </tr>
        <tr>
            <td colspan="3"><input type="submit"/></td>
        </tr>

    </form:form>

</html>
```

Tag `<form:select>` obezbeđuje način za generisanje padajuće liste vrednosti koje kontroler prosleđuje pogledu. Otuda, forma prikazuje polje `sportType` kao skup HTML `<select>` elemenata, umesto prethodno definisanih polja `<input>` koja zahtevaju da korisnik unese tekst vrednost. Sada će biti razmotreno kako kontroler pridružuje `sportType` polje kao atribut modela. Obrada se razlikuje od slučaja sa prethodnim poljima.

Prvo će biti definisana metoda `getAllSportTypes()` u interfejsu `ReservationService` za vraćanje svih dostupnih tipova sporta:

```
/**
 *
 * @author Vlada
 */
public interface ReservationService {

    public List<Reservation> query(String courtName);

    void make(Reservation reservation) throws ReservationNotAvailableException;

    public List<SportType> getAllSportTypes();
}
```

Ova metoda se implementira na **hard - code** način:

```
/**
 *
 * @author Vlada
 */
@Service
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");
    private final List<Reservation> reservations = new ArrayList<>();

    ****

    public List<SportType> getAllSportTypes() {
        return Arrays.asList(new SportType[]{TENNIS, SOCCER});
    }
}
```

## OBRADA LISTE KONTROLEROM

*Dalje će biti istraženo kako kontroler obrađuje kreiranu listu da bi je vratio u pogled forme.*

Dalje će biti istraženo kako kontroler obrađuje ovu listu da bi je vratio u pogled forme.

```
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {

    ****

    @ModelAttribute("sportTypes")
    public List<SportType> populateSportTypes() {

        return reservationService.getAllSportTypes();
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(
        @RequestParam(required = false, value = "username") String username,
        Model model) {
        Reservation reservation = new Reservation();
        reservation.setPlayer(new Player(username, null));
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") Reservation reservation,
        BindingResult result, SessionStatus status) {
        reservationService.make(reservation);
        return "redirect:reservationSuccess";
    }
}
```

Primećuje se da je `setupForm` rukovalac metoda obuhvatila prazan povratni `Reservation` objekat da bi pogled forme ostao nepromenjen. U nastavku, metoda obeležena anotacijom `@ModelAttribute("sportTypes")` je odgovorna za prosleđivanje liste `SportType` atributu modela za pogled forme. Ova anotacija definiše globalne attribute modela dostupne bilo kojem pogledu kojeg koriste metode rukovaoci. Na isti način, rukovalac metoda deklariše objekat `Model` kao ulazni parametar i pridružuje attribute da im je moguće pristupiti u vraćenom pogledu.

Na kraju je moguće primetiti da će `SportType` objekti **TENNIS** i **SOCCER** biti pridruženi atributu modela `sportTypes`.

## POVEZIVANJE OSOBINA KREIRANIH TIPOVA

*Osobine kreiranih tipova ne mogu biti konvertovane bez posebnog konvertera osobina za njih.*

Kada je popunjena forma potvrđena, kontroler povezuje vrednosti polja forme sa osobinama objekata modela sa istim nazivom, u ovom slučaju objektom `Reservation`. Međutim osobine

kreiranih tipova ne mogu biti konvertovane kontrolerom ukoliko nije specificiran poseban editor (konverter) osobina za njih.

Na primer, polje za izbor tipa sporta, dozvoljava samo ID izabranog tipa sporta, jer je to način na koji HTML `<select>` polje radi. Otuda je nophodno konvertovati ovaj ID u objekat tipa ***SportType***, primenom odgovarajućeg editora. Prvo, neophodna je metoda `getSportType()` u *ReservationService* za prihvatanje *SportType* objekta na osnovu njegovog ID-a:

```
public interface ReservationService {

    ****

    public SportType getSportType(int sportTypeId);
}
```

U svrhu testiranja, metoda može biti implementirana sa **switch - case** strukturom:

```
package com.metropolitan.service;
...
public class ReservationServiceImpl implements ReservationService {
...
    public List<SportType> getAllSportTypes() {
        return Arrays.asList(new SportType[]{TENNIS, SOCCER});
    }

    public SportType getSportType(int sportTypeId) {
        switch (sportTypeId) {
            case 1:
                return TENNIS;
            case 2:
                return SOCCER;
            default:
                return null;
        }
    }
}
```

Tada je moguće kreirati klasu, ***SportTypeConverter***, za konverziju ID tipa u objekat klase *SportType*.

```
package com.metropolitan.domain;

import com.metropolitan.service.ReservationService;
import org.springframework.core.convert.converter.Converter;
/**
 *
 * @author Vlada
 */
public class SportTypeConverter implements Converter<String, SportType> {

    private ReservationService reservationService;
```

```

public SportTypeConverter(ReservationService reservationService) {
    this.reservationService = reservationService;
}

@Override
public SportType convert(String source) {
    int sportTypeId = Integer.parseInt(source);
    SportType sportType = reservationService.getSportType(sportTypeId);
    return sportType;
}
}

```

## ANGAŽOVANJE KONTROLERA

*Neophodno je omogućiti angažovanje kontrolera primenom metode `addFormatters()`.*

Pošto je kreirana klasa `SportTypeConverter` neophodna za povezivanje osobina forme sa osobinama koje odgovaraju kreiranim klasama, npr. `SportType`, neophodno je omogućiti angažovanje odgovarajućeg kontrolera. U ovu svrhu, moguće je upotrebiti metodu `addFormatters()` interfejsa `WebMvcConfigurer`.

```

package com.metropolitan.config;

import com.metropolitan.domain.SportTypeConverter;
import com.metropolitan.service.ReservationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

/**
 *
 * @author Vlada
 */
@Configuration
@EnableWebMvc
@ComponentScan("com.metropolitan")
public class WebConfiguration implements WebMvcConfigurer {

    @Autowired
    private ReservationService reservationService;

    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addConverter(new SportTypeConverter(reservationService));
    }
}

```



Jedino polje u klasi, odgovara objektu `reservationService`, koristi se za pristup zrnju `ReservationService` unutar aplikacije. Takođe, moguće je primetiti upotrebu anotacije `@Autowired` koja omogućava automatsko umetanje zrna. Nakon toga, implementirana je metoda `addFormatters()` za povezivanje klasa `Date` i `SportTypeConverter`. Ovde je moguće videti dva poziva kojima se registruju konverter i formater (linije koda 25 i 26). Ove metode pripadaju objektu `FormatterRegistry`, koji je prosleđen kao ulazni parametar metodi `addFormatters()`.

Prvi poziv je iskorišćen za povezivanje klase `Date` sa klasom `DateFormatter`. Klasu `DateFormatter` obezbeđuje radni okvir `Spring`. Navedena klasa obezbeđuje funkcionalnosti parsiranja i štampanja `Date` objekata. Drugim pozivom registrovana je klasa `SportTypeConverter`.

## DEMONSTRACIJA UPRAVLJANJA FORMAMA

*Nakon dorade inicijalnog projekta sledi demonstracija njegovog izvršavanja.*

Sada je moguće izvršiti ponovno prevođenje programa, preporučuje se *Build with Dependencies*, i pratiti rezultate njegovog izvršavanja.

Pozivom linka:

```
http://localhost:8080/court/reservationForm
```

otvara se stranica sa sledećim sadržajem:



Slika 8.1 Učitavanje stranice reservationForm.jsp [izvor:autor]

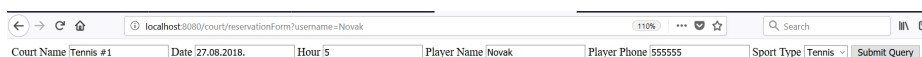
Moguće je uraditi i sledeći poziv:

```
http://localhost:8080/court/reservationForm?username=Novak
```



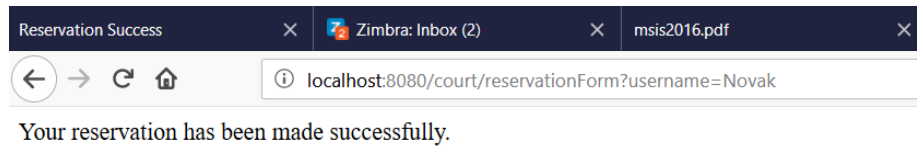
Slika 8.2 Učitavanje stranice reservationForm.jsp sa predefinisanim osobinom [izvor:autor]

Neka je forma popunjena na sledeći način:



Slika 8.3 Popunjena forma [izvor:autor]

Klikom na dugme "*Submit*" potvrđuju se podaci uneti u formu i vrši se angažovanje kontrolera forme. Nakon obrade, vrši se preusmeravanje na stranicu *reservationSuccess.jsp*, a to je moguće videti na sledećoj slici.



Slika 8.4 Uspešno obavljena rezervacija - stranica reservationSuccess.jsp [izvor:autor]

## PROVERA PODATAKA FORME

*Objekti validatori implementiraju Validator interfejs.*

*Spring MVC* podržava validaciju u smislu primene objekata validatora koji implementiraju **Validator** interfejs. Sledeći validator proverava da li su tražena polja forme popunjena:

```
package com.metropolitan.domain;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

/**
 *
 * @author Vlada
 */
@Component
public class ReservationValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Reservation.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Court name is required.");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
            "required.playerName", "Player name is required.");
        ValidationUtils.rejectIfEmpty(errors, "sportType",
            "required.sportType", "Sport type is required.");
    }
}
```

```
}  
}
```

Ovaj validator koristi metode alate `rejectIfEmptyOrWhitespace()` i `rejectIfEmpty()` klase **ValidationUtils**, za validaciju polja forme. Ukoliko je bilo koje odgovarajuće polje prazno, metode će kreirati grešku **field error** koju će povezati sa odgovarajućim poljem. Drugi argument ovih metoda je naziv osobine, a treći i četvrti su kodovi grešaka i podrazumevana poruka o grešci.

Budući da je validator klasa obeležena anotacijom **@Component**, Spring pokušava da instancira klasu kao zrna u skladu sa nazivom klase, a u ovom slučaju **reservationValidator**.

## DATOTEKA SA PORUKAMA

*Validatori mogu da kreiraju greške u proveru, neophodno je definisati poruke, za kodove grešaka.*

Budući da validatori mogu da kreiraju greške tokom provere, neophodno je definisati poruke, za kodove grešaka, koje će biti prikazane korisnicima. Ukoliko postoji definisan **ResourceBundleMessageSource**, moguće je uključiti sledeće poruke grešaka u datoteku resursa (na primer, `messages.properties`) za odgovarajuću lokaciju:

```
required.courtName=Court name is required  
required.date=Date is required  
required.hour=Hour is required  
required.playerName=Player name is required  
required.sportType=Sport type is required  
invalid.holidayHour=Invalid holiday hour  
invalid.weekdayHour=Invalid weekday hour
```

## VALIDACIJA

*Validacijom upravlja kontroler.*

Za primenu ovog validatora, neophodno je izvesti sledeće modifikacije u kontroleru:

```
package com.metropolitan.controller;  
  
//ovde idu importi  
  
/**  
 *  
 * @author Vlada  
 */  
@Controller
```

```

@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {

    private final ReservationService reservationService;
    //registrovanje validatora
    private ReservationValidator reservationValidator;

    @Autowired
    public ReservationFormController(ReservationService reservationService,
        ReservationValidator reservationValidator) {
        this.reservationService = reservationService;
        this.reservationValidator = reservationValidator;
    }

    @ModelAttribute("sportTypes")
    public List<SportType> populateSportTypes() {

        return reservationService.getAllSportTypes();
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(
        @RequestParam(required = false, value = "username") String username,
        Model model) {
        Reservation reservation = new Reservation();
        reservation.setPlayer(new Player(username, null));
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") @Validated Reservation reservation,
        BindingResult result, SessionStatus status) {
        if (result.hasErrors()) {
            return "reservationForm";
        } else {
            reservationService.make(reservation);
            return "reservationSuccess";
        }
    }

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.setValidator(reservationValidator);

        //koristi se zbog izbacivanja greške konverzije datum - string (11
        karaktera ima string datuma)
        binder.registerCustomEditor(Date.class, new CustomDateEditor(new
        SimpleDateFormat("dd.MM.yyyy"), true, 11));
    }
}

```

```
}  
}
```

Prvi dodatak kontroleru je *ReservationValidator* polje koje daje pristup kontroleru instanci validator zrna. Na onovu *@Autowired* anotacije, *ReservationValidator* zrno je umetnuto zajedno sa postojećim zrnom

*ReservationService*.

Sledeća modifikacija je u metodi rukovaocu za *HTTP POST* zahtev, koja se uvek poziva kada korisnik popuni formu. Odmah pored anotacije *@ModelAttribute* sada se nalazi anotacija *@Validated* koja omogućava pokretanje procesa validacije objekata. Nakon obavljene provere, rezultujući parametar (objekat *result* klase *BindingResult*) čuva rezultate procesa validacije. Zatim, vrši se proveravanje uslova *result.hasErrors()*. Ukoliko klasa validator otkrije grešku, ova vrednost će biti *true*.

U slučaju greške, rukovalac metoda vraća pogled *reservationForm*, koji odgovara početnoj formi pa korisnik ponovo može uneti podatke. U slučaju da tokom provere nisu otkrivene greške, vrši se poziv metode za realizovanje rezervacije - *reservationService.make(reservation)*; - praćen preusmeravanjem na pogled *uspeha reservationSuccess*.

Registrowanje validatora je obavljeno u metodi obeleženoj anotacijom *@InitBinder*, a validator je podešen pomoću *WebDataBinder* tako da može da bude korišćen nakon obrađenog povezivanja podataka sa forme. Za registrowanje validatora neophodno je upotrebiti metodu *setValidator()*. Takođe, moguće je registrovati veći broj validatora primenom metode *addValidators()* koja kao argumente uzima jednu ili više instanci tipa *Validator*.

## DEMONSTRACIJA VALIDACIJE

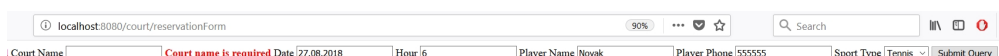
### *Demonstacija primene validatora na polja kreirane forme.*

Prema uslovima validacije, polje za naziv terena je obavezno. Neka je forma popunjena tako da je ovo polje ostavljeno praznim, kao na sledećoj slici.



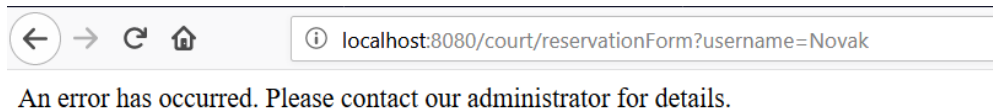
Slika 8.5 Nekompletno popunjena forma [izvor:autor]

Klikom na dugme za potvrdu, forma se proverava i javlja se greška validacije. Navedeno je prikazano sledećom slikom.



Slika 8.6 Greška validacije [izvor:autor]

Konačno, ukoliko u programu dođe do bilo kakvih grešaka koje za posledicu imaju nemogućnost učitavanja željene stranice, doći će do učitavanja poznatog pogleda *error*, odnosno njemu odgovarajuće stranice *error.jsp*.



Slika 8.7 Stranica greške [izvor:autor]

## NAPOMENE U VEZI SA TRAJANJEM SESIJE

*Neophodno je pronaći idealno vreme za isticanje sesije korisnika.*

Sa ciljem da se podrži mogućnost da forma bude popunjena više puta i da se ne izgube podaci koje je obezbedio korisnik, između dva popunjavanja, kontroler se oslanja na primenu anotacije *@SessionAttributes*. Tada, referenca na polje *reservation*, reprezentovana kao objekat *Reservation*, je sačuvana između dva zahteva. Međutim kada se uspešno popuni forma i kada je uspešno obavljena rezervacija, nema svrhe čuvati *Reservation* objekat u sesiji korisnika. Zapravo, ako u kratkom vremenskom periodu korisnik ponovo poseti formu, postoji mogućnost da će se ostaci starog *Reservation* objekta pojaviti ukoliko se ne uklone.

Vrednosti pridružene primenom anotacije *@SessionAttributes*, mogu biti uklonjene pomoću objekta *SessionStatus*. Ovaj objekat može biti pridružen kao ulazni parametar rukovalac metode. Sledeći listing pokazuje kako ističe podatak o kontolerovoj sesiji.

```
package com.metropolitan.controller;
...
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {
...
@RequestMapping(method = RequestMethod.POST)
public String submitForm(
@ModelAttribute("reservation") Reservation reservation,
BindingResult result, SessionStatus status) {
    reservationValidator.validate(reservation, result);
    if (result.hasErrors()) {
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    } else {
        reservationService.make(reservation);
        status.setComplete();
        return "reservationSuccess";
    }
}
```

Kada rukovalac metoda izvrši rezervaciju pozivom ***reservationService.make(reservation)*** ; i neposredno pre nego što je korisnik prosleđen na stranicu *uspeha*, idealno je vreme za isticanje podataka sesije kontrolera. Ovo se obavlja pozivom metode ***setComplete()*** objekta ***SessionStatus*** .

## RAD SA FORMAMA U SPRING MVC - VIDEO MATERIJAL

### *JSP Form binding using Model Attribute - video*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

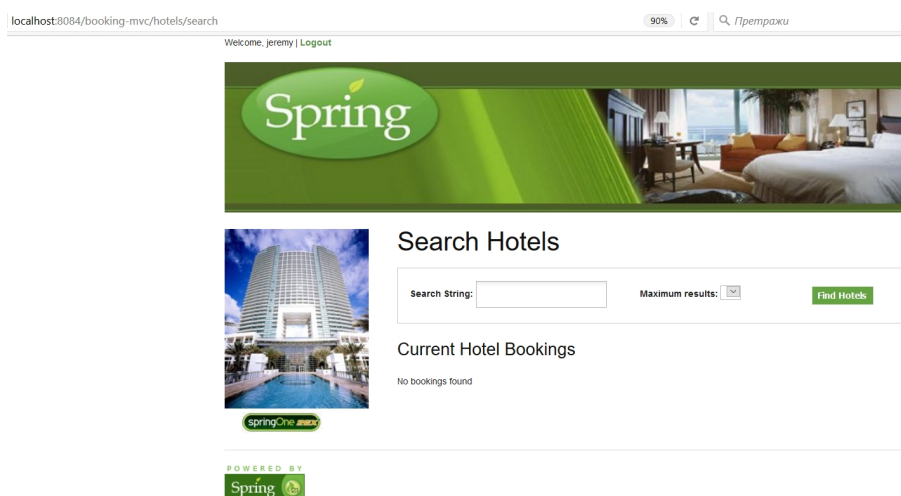
## ▼ Poglavlje 9

# Spring MVC - pokazna vežba

## STRANICE SA FORMOM ZA PRETRAGU I REZULTATIMA PRETRAGE (45 MIN)

*Nakon uspešnog prijavljivanja na sistem otvara se stranica sa odgovarajućom formom*

Pokretanjem programa otvara se početna stranica na kojoj je moguće obaviti prijavljivanje na sistem pretrage hotela. Uspešnim prijavljivanjem na sistem, vrši se preusmeravanje na stranicu koja sadrži formu pomoću koje je moguće uneti ključnu reč za pretragu dostupnih hotela (Slika 1). Forma je upravljana odgovarajućim kontrolerom i nakon unosa i potvrđivanja vrši se preusmeravanje na stranicu koja prikazuje rezultate pretrage.



Slika 9.1 Stranica sa formom za pretragu hotela [izvor:autor]

Sledećom slikom je prikazana stranica koja prikazuje rezultate pretrage hotela.





Slika 9.2 Rezultat pretrage [izvor:autor]

## IZBOR IZ REZULTATA PRETRAGE I FORMA ZA REZERVACIJU

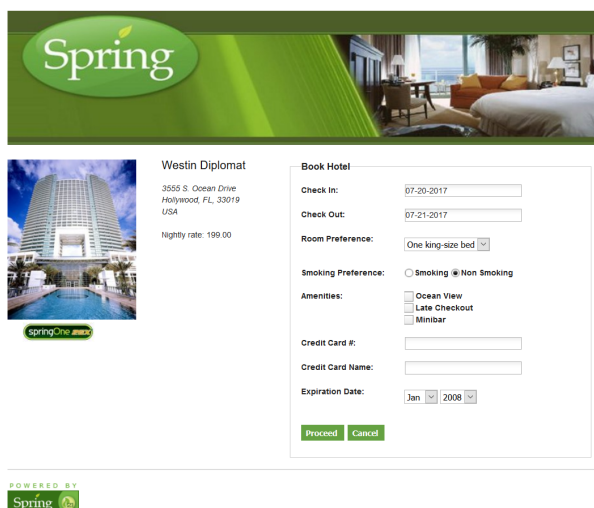
*Sledi dalja demonstracija funkcionisanja Spring MVC aplikacije*

Iz liste rezultata pretrage biraju se pojedinačni hoteli, kao na sledećoj slici.



Slika 9.3 Izbor iz rezultata pretrage [izvor:autor]

Ukoliko je izabran konkretan hotel, otvara se stranica sa formom za rezervaciju hotela, kao na sledećoj slici.



**Spring**

**Westin Diplomat**  
3555 S. Ocean Drive  
Hollywood, FL, 33019  
USA  
Nightly rate: 199.00

**Book Hotel**

Check In: 07-20-2017  
Check Out: 07-21-2017  
Room Preference: One king-size bed  
Smoking Preference: ☐ Smoking ☒ Non Smoking  
Amenities: ☐ Ocean View ☐ Late Checkout ☐ Minibar  
Credit Card #:   
Credit Card Name:   
Expiration Date: Jan 2008  
**Proceed** **Cancel**

POWERED BY  
**Spring**

Slika 9.4 Forma za rezervaciju [izvor:autor]

Prethodna forma sadrži i neka obavezna polja (videti sledeću sliku).

**Book Hotel**

Check In: 07-20-2017  
Check Out: 07-21-2017  
Room Preference: One king-size bed  
Smoking Preference: ☐ Smoking ☒ Non Smoking  
Amenities: ☐ Ocean View ☐ Late Checkout ☐ Minibar  
Credit Card #:   
Credit Card Name:   
Expiration Date: Jan 2008  
**Proceed** **Cancel**

Slika 9.5 Primena obaveznih polja [izvor:autor]

Kompletno urađen i testran primer, koji je ovde bio demonstriran, dostupan je na kraju ovog objekta učenja kao aktivnost Shared Resources.

## ▼ Poglavlje 10

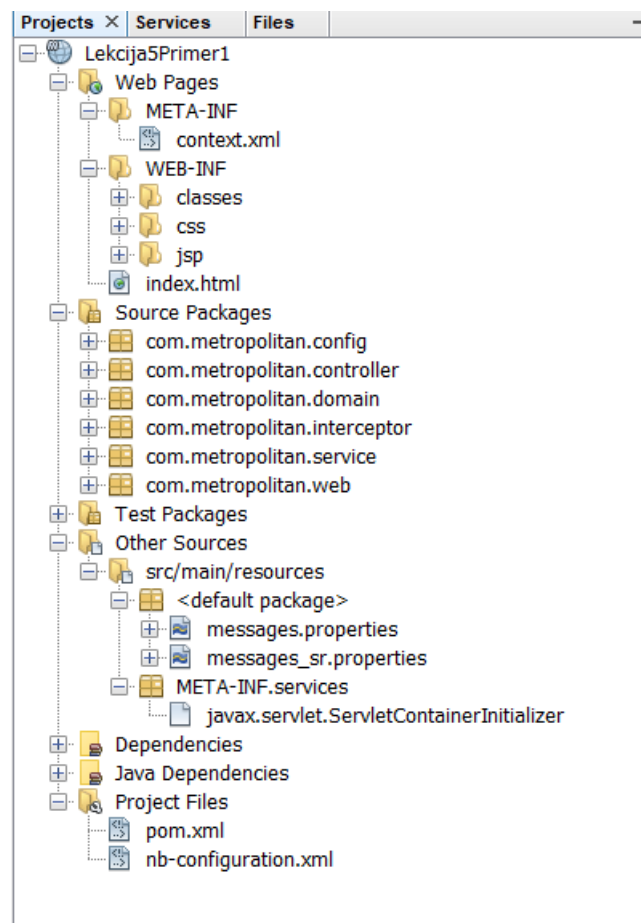
# Spring MVC - Individualna vežba 5

## ZAHTJEVI (90 MINUTA - PREDVIĐENO VREME)

### *Kreiranje Spring MVC aplikacije*

Kreiranje Spring MVC aplikacije po sledećim zahtevima:

1. Kompletira se primer parcijalno prezentovan na predavanjima u kompletnu i potpuno funkcionalnu Spring MVC aplikaciju;
2. Kreirajte samostalno vaš Maven projekat koji odgovara veb aplikaciji;
3. Napravite strukturu projekta tako da, na kraju, ima oblik priložen slikom 1;
4. Dodati neophodne zavisnosti u pom.xml;
5. Kreirati domenski sloj sa svim prikazanim klasama;
6. Kreirati upravljački sloj sa svim obrađenim kontrolerima;
7. Kreirati servise i implementacione klase servisa;
8. Kreirati konfiguraciju za kontekst aplikacije i registrovanje svih neophodnih zrna za funkcionisanje aplikacije;
9. Kreirati i implementirati presretače na obrađeni način;
10. Kreirati resurse sa porukama i primeniti različita lokalna podešavanja za iste poglede;
11. Primeniti validaciju na stranice sa formama;
12. Prevesti aplikaciju i pokrenuti je;
13. Demonstrirati funkcionalnost aplikacije.



Slika 10.1 Tražena struktura projekta [izvor:autor]

## ▼ Poglavlje 11

### Domaći zadatak 5

#### DOMAĆI ZADATAK BROJ 5 (120 MINUTA)

*Cilj domaćeg zadatka je da student napravi samostalno MVC aplikaciju*

Kreirati Spring MVC aplikaciju za iznajmljivanje automobila:

1. Koristiti Maven za dodavanje i upravljanje zavisnostima;
2. Aplikacija mora da bude podeljena na domenski (Korisnik, Automobil...), servisni, konfiguracioni, upravljački (kontrolerski) i prezenatcioni sloj;
3. Aplikacija koristi Java konfiguracije, nikako XML;
4. Za poglede koristiti JSP okvir;
5. Omogućiti prikazivanje sadržaja na dva jezika (Srpski i engleski);
6. Koristiti validaciju podataka koji se unose u formama stranica;

Urađeni zadatak proslediti vašem asistentu na mail: [nikola.dimitrijevic@metropolitan.ac.rs](mailto:nikola.dimitrijevic@metropolitan.ac.rs).

Nakon obaveznog zadatka studenti dobijaju dodatne različite zadatke na email od predmetnog asistenta.

## ▼ Poglavlje 12

# Zaključak

## ZAKLJUČAK

*Lekcija 06 je obradila osnove Spring MVC.*

Savladavanjem ove lekcije, studenti su stekli osnove za uspešno razvijanje JAVA web aplikacija primenom okvira *Spring MVC*. Centralna komponenta Spring MVC je *DispatcherServlet* koji se ponaša kao glavni kontroler za prosleđivanje zahteva ka metodama rukovaocima. Zatim je istaknuto da su kontroleri JAVA klase koje su obeležene anotacijom *@Controller*. Takođe, u svetlu *Spring MVC*, pokazano je kako se koriste, i koja im je namena, druge anotacije poput: *@RequestMapping*, *@Autowired* i *@SessionAttributes*.

Takođe, u posebnom delu lekcije obrađen je koncept presretača, a pokazano je i kao *Spring MVC* rešava problem podrške za lokalizaciju i internacionalizaciju. Dalje, pažnja je usmerena ka specifičnoj *Spring MVC* terminologiji koja se fokusira na problem *rešavanja* pogleda. Posebna tema u ovoj lekciji, bilo je upravljanje izuzecima na *Spring MVC* način. Lekcija završava izlaganje dodatnim razmatranjima u vezi sa zadacima kontrolera.

## LITERATURA

*U pripremi Lekcije 06 korišćena je najnovija literatura.*

Za pripremu lekcije korišćena je najnovija pisana i elektronska literatura:

1. Marten Deinum, Josh Long, and Daniel Rubio, Spring 5 Recipes, Apress (2017)
2. Spring Framework Reference Documentation - <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/>
3. Craig Walls, Spring in Action, Manning
4. Craig Walls, Spring Boot in Action, Manning

Dopunska literatura:

1. <http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/spring-tutorials/>
2. <http://www.tutorialspoint.com/spring/>
3. <http://www.javatpoint.com/spring-tutorial>