Spring MVC

Róbert Novotný

Webové frameworky v Jave

- základom webových frameworkov sú **servlety**
- poskytujú však len minimálnu funkcionalitu
- tvorba výstupu sa redukuje na `out.println("<b>" + tučnýText + "</b>")`
  - vzhľad je napevno v kóde
  - nemožno oddeliť webdizajnéra od programátora
- získavanie parametrov z požiadavky je ťažkopádne

Servlety v Jave

- aplikácie založené na servletoch často pozostávajú z **jediného magického servletu**
- jedna trieda, ktorá rieši všetko
  - získava parametre z požiadavky
  - zabezpečuje aplikačnú logiku
  - rieši výstup
- to vedie k **špagetovému kódu**

Java Server Pages (JSP)

- JSP sú pokusom o priblíženie sa k webovým dizajnérom
- filozofia podobná PHP
- píšeme HTML, v ktorom sú kusy Java kódu
- kompilácia sa deje na serveri = rýchlejší vývoj
- z hľadiska histórie je to však **omyl**
  - vznikajú obrovské JSP, v ktorých sa kombinuje pevné HTML a dynamické HTML
  - dizajnér aj tak musí vedieť Javu

Poučenie z vývoja a pohľad do histórie

- 1979: jazyk SmallTalk
- snaha o jasné **oddelenie**
  - dát (typicky doménové objekty)
  - a spôsobu ich zobrazenia (používateľské rozhranie)
- vzniká návrhový vzor **MVC**
  - **M**odel = dáta
  - **V**iew = zobrazenie dát
  - **C**ontroller = prostredník medzi **M** a **V**

Poučenie z vývoja a pohľad do histórie

- **Model** obsahuje dáta z aplikačnej domény (typicky doménové objekty)
- **View** reprezentuje používateľské rozhranie, zobrazuje dáta poskytované *modelom*.
- **Controller** spracováva používateľov vstup a podľa potreby aktualizuje *model* a odosiela ho do *view* vrstvy.

  + v Spring MVC tiež interaguje s databázovou / servisnou vrstvou

  + v Spring MVC navyše rieši navigáciu medzi viewmi

MVC a webové frameworky

- v servletoch je všetko na kope
  - aplikačná logika lepí HTML (používateľské rozhranie)
  - aplikačná logika vyťahuje dáta z požiadavky
- v škaredom JSP je tiež všetko na kope
  - v rámci definovania prezentačnej vrstvy máme ostrovy aplikačnej logiky

MVC a webové frameworky

- na webe však vieme oddeliť
- vzhľad stránok, ktoré vidí používateľ
  - čiže to, čo prezentuje webový prehliadač
- aplikačnú logiku
  - tá je na serveri
- a dáta odosielané z prehliadača na server a späť

MVC

používateľ si vyžiada stránku zo servera (navštíví URL) a voliteľne dodá parametre

server vytiahne parametre, spracuje, vytiahne dáta, vytvorí stránku a pošle ju používateľovi

Spring MVC

- aplikačný rámec pre vývoj webových aplikácií
- je súčasťou Springu
- abstraktný a flexibilný
- vo verzii 2.5 značne prepracovaný, zjednodušený a sprehľadnený
- postavený na vlastnostiach Javy 5
  - anotácie
  - konfigurácia na základe konvencií

Spring MVC

- kontroléry (logika):
  - kontrolérom sa môže stať ľubovoľná trieda
  - netreba dediť, implementovať
  - stačí dodať pár anotácií
- view (výzor):
  - štandardne sú používané JSP
  - tie však obsahujú minimum kódu
- model (dáta):
  - obaľuje dáta lietajúce medzi viewom a kontrolérom
  - vie obaliť ľubovoľnú inštanciu

Spring MVC – čo potrebujeme

- potrebujeme stiahnuť:
  - spring.jar
  - spring-webmvc.jar
  - commons-logging-1.1.1.jar
  - JSTL knižnice: jstl.jar + standard.jar
- potrebujeme mať nainštalovaný servletový kontajner (Tomcat, Jetty...)

Spring MVC – architektúra

- kontrolérov je zvyčajne veľa
- každý z nich zodpovedá za jednu operáciu, či sadu logických operácií
- typickým je kontrolér spravujúci operácie nad entitou (**CRUD**)
  - **C**reate – vkladanie entity do systému
  - **R**ead – zobrazovanie entity
  - **U**pdate – aktualizácia dát
  - **D**elete – odstraňovanie entity

Príklad kontroléra

```
public class SimpleController {


   public void logCurrentDate() {
      System.out.println(new Date());
   }
}
```

klasická trieda, nič špeciálne

Spring MVC – kód kontroléra

```
@Controller
public class SimpleController {

   @RequestMapping("/date.do")
   public void logCurrentDate() {
      System.out.println(new Date());
   }
}
```

@Controller: trieda je kontrolér
zároveň je springovským beanom
@RequestMapping: špecifikuje príponu URL adresy,
ktorú bude obsluhovať

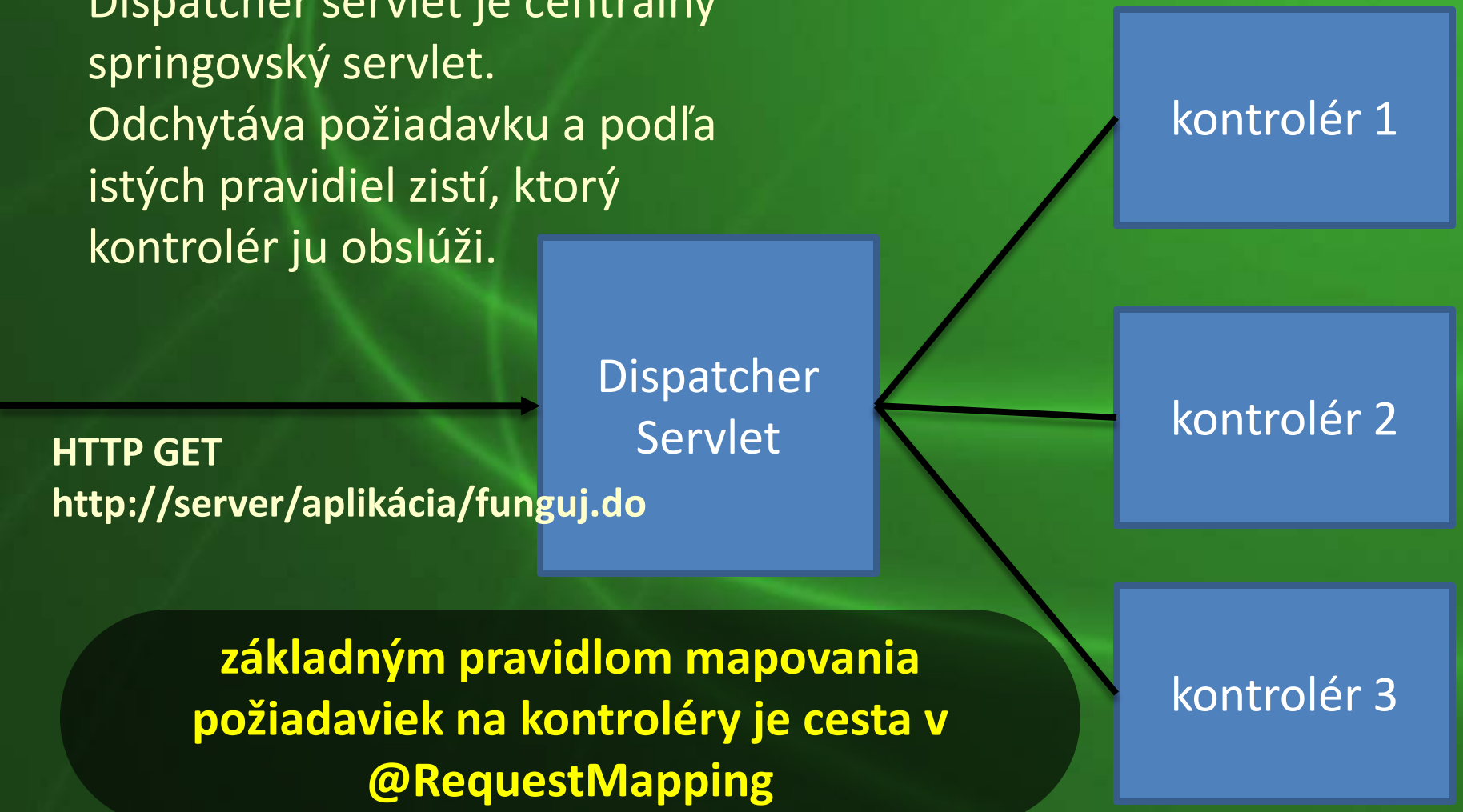
Spring MVC – architektúra

```
@RequestMapping("/date.do")
```

- klasická aplikácia je zvyčajne nasadená na adresu s danou predponou **context path**
- @RequestMapping určuje **adresu**, na ktorej kontrolér počúva.
- tá je relatívna vzhľadom na *context path*
- príklad:
  - webová aplikácia s context path = "**/datetime**"
  - uvedený kontrolér počúva na http://*server:port*/**datetime/date.do**

Spring MVC – architektúra

Dispatcher servlet je centrálny springovský servlet.
Odchytáva požiadavku a podľa istých pravidiel zistí, ktorý kontrolér ju obslúži.



základným pravidlom mapovania požiadaviek na kontroléry je cesta v @RequestMapping

Konfigurácia DispatcherServletu

- rieši sa vo web.xml ako v prípade akéhokoľvek iného servletu

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Servlet obslúži všetky URL končiace sa na *.do

Konfigurácia aplikačného kontextu

- ďalej potrebujeme nakonfigurovať aplikačný kontext pre Spring

Konvencia!

pre server s názvom **spring-mvc** (viď predošlý slajd)
hľadáme **springmvc-servlet.xml** vo WEB-INF

Konfigurácia aplikačného kontextu

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
>
   <context:component-scan
                base-package="sk.spring.mvc" />
</beans>
```

Zapne automatické vyhľadávanie tried v CLASSPATH. Všetky triedy anotované ako **@Controller** v balíčku **sk.spring.mvc** sú zaradené do aplikačného kontextu a sú považované za kontroléry.

Konvencie pre kontroléry

```
<context:component-scan
        base-package="sk.spring.mvc" />
```

Konvencia!
Triedy anotované ako @Controller sa pokladajú za triedy kontrolérov.

Konvencia!
Triedy anotované ako @Controller sú automaticky zaradené do aplikačného kontextu Springu.
Nemusíme ich teda deklarovať ako <bean> v XML.

Metódy s parametrami

- metódy kontrolérov môžu mať parametre
- ich hodnoty sa automaticky prevezmú z parametrov v URL

```
@Controller
public class SimpleController {
  @RequestMapping("/date.do")
  public void logCurrentDate(String locale) {
    Date date = getDateFromLocale(locale)
    System.out.println(date);
  }
}
```

Metódy s parametrami

```
http://server:port/dateTime/date.do?locale=en
```

```
@Controller
public class SimpleController {
  @RequestMapping("/date.do")
  public void logCurrentDate(String locale) {
    // locale má hodnotu "en"
  }
}
```

Metódy s parametrami

- v prípade, že parameter v URL nebol špecifikovaný, hodnota je **null**
- podporované sú všetky základné dátové typy
- Spring automaticky zabezpečí konverziu zo Stringov
  - namiesto primitívov je niekedy lepšie používať objektové typy: **Integer**, **Boolean**...
  - ak je parameter neprítomný, vieme to zistiť testom na **null**

Metódy s návratovou hodnotu

- metódy kontrolérov môžu vracať objekty
- tie sa obalia do modelu a odošlú do view vrstvy

```
@Controller
public class StudentController {
  @RequestMapping("/showStudent.do")
  public Student getStudent(int id) {
    return findStudentById(id);
  }
}
```

Model a návratová hodnota

- model prenáša dáta medzi kontrolérmi a view
- objekty, ktoré vracajú metódy kontrolérov sa **obalia** do modelu a odošlú do view vrstvy
- v Spring MVC je model implementovaný ako mapa reťazcov na objekty (modelová mapa, **ModelMap**)

Konvencia!

Ak vložíme objekt do modelu, kľúč v mape sa odvodí z názvu triedy.

- ,,vložiť do modelu" = ,,vložiť do modelovej mapy"

Konvencie pre vkladanie do modelu

- sk.spring.mvc.**Student** -> "**student**"
- kolekcie a polia: dátový typ + "list"
  - **List<Student>** -> "**studentList**"
  - **Student[]** -> "**student**List"
  - **Set<Student>** -> "**studentList**"
- **Map**<Student, Integer> -> **map**
  - mapa nie je kolekcia, berie sa len názov!
- **HashMap<Student, Integer> -> hashMap**

Konvencie pre vkladanie do modelu

Konvencia!
Ak metóda vracia objekt, ten sa vloží do modelu s automaticky odvodeným kľúčom

```
public Student getStudent(Integer id) {
    …
    return nájdenýŠtudent;
}
```

- mapa obsahuje "student" -> nájdenýŠtudent

Povolené parametre

- kontroléry majú flexibilné parametre a návratové hodnoty
- možno pristupovať k objektom požiadaviek, objektom pre manuálny zápis dát, sessionom...
- to isté sa týka návratových typov
- niektoré ukážky si ukážeme na príkladoch
- viď dokumentácia
  http://static.springframework.org/spring/docs/2.5.x/reference/mvc.html

Ako sa zobrazí view vrstva?

- *view* je v Springu abstraktný pojem
  - pripomeňme si, že je to konkrétny spôsob / forma, akým sa zobrazujú dáta z modelu
- klasický view je JSP stránka
  - ale je možné robiť aj PDF/Excel/RSS view
- každý view má **logické meno**
- podľa mena možno nájsť konkrétnu reprezentáciu (PDF...)

Ako sa zobrazí view vrstva?

Konvencia!

Logické meno view, ktorý sa zobrazí po návrate z metódy, sa odvodí z URL adresy, ktorú obsluhuje kontrolér.

```
@RequestMapping("/displayStudent.do")
public Student getStudent(int id) {
  return findStudentById(id);
}
```

Odsekneme príponu, zrušíme lomku na začiatku
=> view s menom **displayStudent**

- vieme teda, ktoré view zobraziť
- a vieme aké budú dáta (model), ktoré sa zobrazia

Ako sa zobrazí view vrstva?

- vieme logické meno *view,*
- lenže ako ho namapovať na JSP stránku?
- **view resolver!**
- deklarujeme v aplikačnom kontexte:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet
             .view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

zobrazíme súbor /WEB-INF/jsp/**displayStudent**.jsp

JSP stránka

- vytvoríme JSP stránku /WEB-INF/jsp/displayStudent.jsp

```
<h1>Detaily o studentovi</h1>
<b>ID:</b> ${student.id} <br />
<b>Meno:</b> ${student.firstName} <br />
<b>Priezvisko:</b> ${student.lastName} <br />
<b>Rocnik:</b> ${student.year} <br />
```

- máme v podstate HTML, kde používame špeciálne premenné
- JSP predstavuje šablónu (template)

Model a šablóna

- hodnoty premenných sa prevezmú z modelu!
- **šablóna** + **dáta** z modelu => výsledná **stránka**
- názvy premenných sú kľúče v modelovej mape
- hodnota premennej = hodnota v modelovej mape
  - ${student}: objekt uložený v modelovej mape pod kľúčom student
  - ${student.id}: získa z mapy hodnotu pre kľúč student a zavolá na nej getId()
- syntax je definovaná v jazyku **JSP EL**

Kontrolér vracajúci zoznam

Kontrolér môže mať aj viac metód!

```
@Controller
public class StudentController {
  @RequestMapping("/listStudents.do")
  public List<Student> listStudents() {
    return findStudents();
  }
}
```

Do modelovej mapy sa vloží
["studentList", zoznam študentov]
a zobrazí sa view **listStudents**

JSP stránka pre zoznam študentov

- vytvoríme JSP stránku /WEB-INF/jsp/listStudents.jsp
- filozofia:
  - v cykle prejdeme zoznam študentov
  - každého študenta vypíšeme na samostatný riadok
- v JSP máme dve možnosti pre zoznam:
  - buď vložíme Java kód priamo
    - neprehľadné, zle udržiavateľné
  - alebo použijeme špeciálne značky

JSP stránka pre zoznam študentov

- v JSP existuje sada špeciálnych značiek
- Java Standard Tag Library (**JSTL**)
- podporuje typické operácie
  - **podmienky** („ak *niečo* potom zobraz elementy")
  - **cykly** („prelez cez zoznam a niečo sprav")
  - **výpis obsahu premenných** z modelu do stránky
  - a mnoho iného
- sady značiek je nutné zaviesť do stránky spolu s *prefixom*

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

JSP stránka pre zoznam študentov

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
<table>
  <tr><th>Meno</th><th>Priezvisko</th><th>Ročník</th>

  <c:forEach items="${studentList}" var="student">
    <tr>
      <td>${student.firstName}</td>
      <td>${student.lastName}</td>
      <td>${student.year}</td>
    </tr>
  </c:forEach>
</table>
```

forEach: značka pre cyklus
items: meno premennej z modelu obsahujúcej zoznam, ktorý sa iteruje
var: premenná obsahujúca aktuálny prvok zoznamu

Zadávanie dát cez formuláre

- HTML formuláre sú štandardný (a v podstate jediný) spôsob získavania dát od používateľa
- reprezentované sadou ovládacích prvkov

Ovládací prvok	Anglický názov	HTML kód
textové pole	textfield	<**input** type="text">
viacriadkové textové pole	textarea	<**textarea**
začiarkávacie pole	checkbox	<**input** type="checkbox">
zoznam	list	**<select>** pre zoznam, <**option>** pre položky
rozbaľovací zoznam	combobox	
tlačidlá	button	<**input type="submit"**> pre odoslanie formulára

Postupnosť krokov pri práci s formulármi

1. používateľ navštívi stránku	• zavolá sa metóda kontroléra
2. je mu prezentovaný prázdny formulár	• zobrazí sa view s prázdnym modelovým objektom, ktorý sa naviaže na ovládacie prvky
3. vyplní ho	• dáta sa odošlú
4. odošle dáta na server	• naviažu na modelový objekt, t. j. na parameter metódy
5. ak sú dáta vyplnené nesprávne, zobrazí sa pôvodný formulár a prejde sa na krok 3	• ak sú dáta nesprávne, zobrazí sa view s čiastočne vyplneným modelovým objektom, ktorý sa naviaže na ovládacie prvky
6. inak sa dáta spracujú	

Implementácia odosielania dát

- dáta sa odosielajú HTTP protokolom
- reprezentované dvojicami ***kľúč = hodnota***
- formulárové dáta: ***názov ovl. prvku = hodnota***
- príkaz GET
  - dvojice odosielané v rámci URL
  - jednoduchý prístup, navštívením URL možno priamo zadávať dáta
  - dáta je rovno vidieť v adrese – bezpečnostný problém
  - URL adresy majú ohraničenie na dĺžku
  - špeciálne znaky je nutné kódovať. Diakritika sa prenáša zle.
- príkaz POST
  - dvojice odosielané v tele požiadavky
  - žiadne ohraničenie na dĺžku, menej problémov s kódovaním

Implementácia odosielania dát

- dáta sa odosielajú HTTP protokolom
- reprezentované dvojicami ***kľúč = hodnota***
- formulárové dáta: ***názov ovl. prvku = hodnota***

HTTP GET	HTTP POST
dvojice odosielané v URL: ?meno=John&priezvisko=Doe&vek=25	dáta odosielaná v tele požiadavky
navštívením URL možno priamo zadávať dáta	-
URL je priamo vidieť = bezpečnostné riziko	dáta nie sú ,,na očiach" (ale ich zistenie je otázkou správneho nástroja!)
URL majú ohraničenie na dĺžku (cca 512 znakov, niekde 255)	limit dát je obmedzovaný servermi z bezpečnostných dôvodov (Tomcat: 8 MB)
zložité kódovanie špeciálnych znakov, veľké problémy s diakritikou	je možné špecifikovať kódovanie dát

Zadávanie dát cez formuláre

- Spring uľahčuje tvorbu formulárov vlastnou **sadou značiek**
- priamo podporujú naväzovanie (**binding**) ovládacích prvkov na inštančné premenné v modeli
- rieši množstvo typických problémov
  - automatická typová konverzia
  - odlíšenie zaslania formulára od zobrazenia
  - ponechanie vyplnených dát v prípade opravy formulára
  - problém s neprítomnosťou parametra v prípade checkboxov
  - ...

Zadávanie dát cez formuláre

- sadu značiek je potrebné zaviesť do JSP stránky

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

<form:form modelAttribute="book">
<b>ISBN:</b>  <form:input path="isbn"/> <br />
<b>Autor:</b> <form:input path="autor"/> <br />
<b>Názov:</b> <form:input path="nazov"/> <br />
<input type="submit" />
</form:form>
```

tri textové políčka + jedno tlačidlo na odoslanie

Väzba ovládacích prvkov na model

- vo formulári je nutné určiť väzbu medzi ovládacími prvkami a inštančnými premennými objektu v modeli

```
<form:form modelAttribute="book">
<b>ISBN:</b>
<form:input path="isbn" /> <br />
...
</form:form>
```

```
class Book {
  String isbn;
  …
}
```

- **modelAttribute:** premenná objektu v modeli, na ktorý sa namapujú ovládacie prvky
- **path** v ovládacom prvku: cesta k inštančnej premennej objektu, ktorej hodnota je naviazaná na prvok

Parameter *modelAttribute*

```
<form:form modelAttribute="book">
<b>ISBN:</b>
<form:input path="isbn"/> <br />
...
</form:form>
```

- určuje objekt v modeli, ktorý prichádza z kontroléra
  - takto možno predvypĺňať ovládacie prvky
  - použitie ako v bežnom prípade zobrazovania dát
- určuje objekt v modeli, na ktorý sa naviažu hodnoty ovládacích prvkov pri odosielaní formulára
  - mapovanie na parametre metódy

Parameter *modelAttribute*

```
<form:form modelAttribute="book">
<b>ISBN:</b>
<form:input path="isbn"/> <br />
...
</form:form>
```

```
public String createBook(Book book) {
    validator.validate(book, errors);
    if(errors.hasFieldErrors()) {
       return null;
    } else {
       bookService.add(book);
       return "redirect:listBooks.do";
    }
}
```

Realizácia v kontroléri

- **Fáza 1**: používateľ navštívi adresu s kontrolérom a je mu vrátená stránka s prázdnym formulárom

```
@RequestMapping("/createStudent.do");
public Student createStudent() {
  return new Student();
}
```

```
<form:form modelAttribute="student">
<b>Meno:</b>
<form:input path="firstName"/> <br />
...
</form:form>
```

Zobrazí sa view **createStudent** s prázdnymi textovými políčkami – inštancia **Studenta** má prázdne hodnoty

Realizácia v kontroléri

- **Fáza 2**: používateľ vyplní formulár, odošle ho cez HTTP POST. Dáta sa naviažu na objekt v modeli.

```
<form:form modelAttribute="student">
  <form:input path="firstName"/>
</form:form>
```

Model:

```
student =>  Student:
                firstName = ...
                lastName = ...
```

POST

inštancia študenta

Realizácia v kontroléri

- **Fáza 2**: dáta z objektu v modelu sú dostupné v parametri metódy kontroléra

```
student =>  Student:
                firstName = ...
                lastName = ...
```

inštancia študenta v modeli

```
@RequestMapping(method=RequestMethod.POST,
                value="/createStudent.do")
public String createStudent(Student student) {
  // spracuj študenta
  return "redirect:listStudents.do";
}
```

Po zadaní knihy chceme zobraziť ich zoznam. Prehliadač presmerujeme na kontrolér obsluhujúci **/listStudents.do** (detaily neskôr).

Sumarizácia

- máme **dve** metódy **createStudent()**
- obe namapované na URL končiacu na /**createStudent.do**
- jedna bez parametra, volaná cez HTTP **GET**, zobrazuje prázdny formulár

```
@RequestMapping("/createStudent.do")
```

alebo úplný zápis

```
@RequestMapping(method=RequestMethod.GET , value="/createStudent.do")
```

- druhá s parametrom, volaná cez HTTP **POST**, spracováva odoslané dáta

```
@RequestMapping(method=RequestMethod.POST , value="/createStudent.do")
```

Ďalší príklad: editácia študenta

- editáciu študenta vyriešime podobne ako zadávanie
- vyrobíme dve metódy **editStudent()**
- jedna pre zobrazovanie formulára s dátami o upravovanom študentovi
- druhá pre spracovanie zasielaných dát

Editácia: realizácia v kontroléri

- **Fáza 1**: používateľ navštívi adresu s kontrolérom a je mu vrátená stránka s údajmi o študentovi
- vytvoríme **editStudent.jsp** s formulárom
- vytvoríme **metódu** v kontroléri
- metóda sa zavolá cez **GET**

```
@RequestMapping("/editStudent.do");
public Student editStudent(Integer id) {
  return findStudentById(id);
}
```

```
http://...//editstudent.do?id=25
```

Zobrazí sa view **editStudent** s vyplnenými textovými políčkami zodpovedajúcimi hodnotám v inštancii **Studenta**

Editácia: realizácia v kontroléri

- **Fáza 2**: používateľ odošle dáta, tie sa naviažu na modelový objekt
- analogicky k príkladu s vytváraním
- vytvoríme novú metódu, parametrom je **Student**
- metóda sa zavolá cez **HTTP POST**

```
@RequestMapping(method=RequestMethod.POST,
                value="/editStudent.do")
public String editStudent(Student student) {
  // odošli študenta do databázy
  return "redirect:listStudents.do";
}
```

Aký view sa zobrazí?

Realizácia v kontroléri

- doteraz sme mali konvenciu ,,názov view sa odvodí z URL adresy"
- čo ak chceme špeciálny prípad?
- teda zobraziť iný view, než indikuje URL?

Konvencia!

Ak metóda vracia String, ten sa považuje za logické meno view, ktorý sa zobrazí

```
public String editStudent(Student student) {
   return "dashboard";
}
```

Zobrazí sa view "**dashboard**", t. j. **dashboard.jsp**

Realizácia v kontroléri

- návratová hodnota obsahuje logické meno view
- žiadne lomítka, žiadna koncovka *.do!
- jediná **výnimka**
  - reťazec začínajúci na **`redirect:`**
  - prehliadač sa presmeruje na danú URL

```
public String editStudent(Student student) {
return "redirect:listStudents.do";
}
```

Klient sa presmeruje na URL adresu http:/.../**listStudents.do**
Zavolá sa teda kontrolér sediaci
na tejto adrese a zobrazí sa príslušný view,

Prekrývanie konvencií

- štandardná konvencia:
  - návratová hodnota sa do modelu vloží s kľúčom odvodeným z triedy.
  - view sa odvodí z URL adresy
- ak chceme vlastný view:
  - metóda vráti String
  - nemáme možnosť vkladať dáta do modelu!
- čo ak chceme vlastný view a vlastné dáta?

Manuálne vkladanie do modelu, view podľa konvencie

- do metódy pridáme premennú typu **Model**
- do nej vložíme objekty modelu
  - vkladanie pomocou **addAttribute()**
  - kľúč sa odvodí z názvu triedy

```
@RequestMapping("/editStudent.do");
public void editStudent(Integer id, Model model) {
  Student student = findStudentById(id);
  model.addAttribute(student)
}
```

void = zobrazí sa view podľa konvencie (**editStudent**)

do modelu sa vloží dvojica ["student" -> student]

Manuálne vkladanie do modelu, vlastný view

```
@RequestMapping("/editStudent.do");
public String editStudent(Integer id, Model model) {
  Student student = findStudentById(id);
  model.addAttribute(student);
  return "studentForm";
}
```

Vraciame reťazec, t. j. zobrazí sa view **studentForm**

do modelu sa vloží dvojica ["student" -> student]

Čo ak potrebujeme viac modelov?

- zatiaľ sme mali v modeli len jedinú dvojicu
- tá sa naplnila podľa konvencie
- často však potrebujeme mať viac dvojíc
- príklad:
  - rozbaľovací zoznam (combobox)
  - naviazaný na inštančnú premennú objektu (napr. ročník)
  - odkiaľ zoberie combobox zoznam zobrazovaných hodnôt?

Dáta do comboboxu sa predvyplnia na základe modelu.

Dáta z modelu pre ovládacie prvky

```
<form modelAttribute="student">
  ...
  <form:select path="year" items="${yearList}"/>
</form>
```

```
@RequestMapping("/editStudent.do")
public String editStudent(Integer id, Model model) {
  Student student = findStudentById(id);
  model.addAttribute(student);

  List<Integer> years = Arrays.asList(1, 2, 3, 4, 5);
  model.addAttribute("yearList", years);
  return "editStudent";
}
```

V modeli bude "student" -> student
a "yearList" -> zoznam rokov

Dáta z modelu pre ovládacie prvky

- alternatívna možnosť: metódy napĺňajúce model

```
@ModelAttribute("yearList")
public List<Integer> getYears() {
  return Arrays.asList(1, 2, 3, 4, 5);
}
```

- metódu anotujeme **@ModelAttribute**
- Jej výsledok sa vloží do modelu pod daným **kľúčom**
  - ten možno vynechať, potom sa použije konvencia
- všetky takéto metódy sa zavolajú **pred** zavolaním metód obsluhujúcich požiadavky

Problémy s nejednoznačnými parametrami

- formulár pre úpravu študenta sa nachádza na http://.../editStudent.do?**id=*66***
- lenže vo formulári sa tiež nachádza ovládací prvok pre ***id***
- formulár POSTujeme na URL s parametrami
- parameter ***id*** sa teda zjaví dvakrát
  - raz pre prvok z formulára, raz pre dvojicu v URL adrese
- Spring zlúči tieto dve hodnoty do poľa
- do ***id*** sa teda vloží dvojprvkové pole
- lenže id je typu Integer – chyba!

Problémy s nejednoznačnými parametrami

- riešenie 1:
  - premenujeme parameter v metóde kontroléra

```
@RequestMapping("/editStudent.do");
public String editStudent(Integer studentId, Model model)
```

```
http://.../editStudent.do?studentId=66
```

- riešenie 2:
  - anotáciou **@RequestParam** premenujeme parameter

```
public String
  editStudent(@RequestParam("studentId") Integer id,
              Model model)
```

Validácia dát

- do formulárových prvkov možno zadať hocičo
- text do číselných políčok
- možno ich nevyplniť
- možno do nich vložiť nepovolené hodnoty
- ...
- **validácia**: overenie dát od používateľa

Pravidlo!

Všetky dáta od používateľa sú nesprávne a nebezpečné!

- nevalidované dáta môžu viesť k narušeniu bezpečnosti a k napadnutiu systému!

Validácia dát

- základným mechanizmom je interfejs **Validator**
- overíme rozsahy / platnosť premenných inštancie **target**
- ak je premenná neplatná, pridáme hlásenie do inštancie **Errors.** **Errors** je zoznam hlásení

```
public class StudentValidator implements Validator {
  public void validate(Object target, Errors errors) {
     // prebehne validácia
  }

  // určí triedy, ktoré dokáže tento validátor zvalidovať
  public boolean supports(Class clazz) {
    return Student.class.isAssignableFrom(clazz);
  }
}
```

Validácia dát

```
public void validate(Object target, Errors errors) {
  Student student = (Student) target;
  ValidationUtils.rejectIfEmptyOrWhitespace(
      errors, "firstName", "", "Meno nie je vyplnené.");
  if(student.getYear() < 1 || student.getYear() > 5) {
    errors.rejectValue("year", "",
                       "Rok musí byť medzi 1 a 5!");
  }
}
```

- **errors.rejectValue()** odmietne inštančnú premennú, ktorej cesta je v parametri ako nesprávnu
- druhý parameter: kód chyby pre dohľadanie v prípade i18n
- tretí parameter: implicitná hláška

Validácia dát

```
public void validate(Object target, Errors errors) {
  ValidationUtils.rejectIfEmptyOrWhitespace(
      errors, "firstName", "", "Meno nie je vyplnené.");
  ...
}
```

- **rejectIfEmptyOrWhitespace()** odmietne inštančnú premennú ak je prázdna alebo **null**
- prvý parameter je zoznam chýb
- druhý cesta k premennej
- tretí kód chyby
- štvrtý: implicitná hláška

Validácia po odoslaní dát

```
public String editStudent(Student student, Errors errors)
{
  studentValidator.validate(student, errors);
  if(errors.hasErrors()) {
    return "createStudent";
  }
  aktualizuj(student);
  return "redirect:listStudents.do";
}
```

- ak dodáme do metódy parameter typu **Errors**, získame automaticky zoznam chýb, ktorý odovzdáme validátoru
- **Errors** musí nasledovať **hneď** za parametrom, ktorý chceme validovať
- metóda **hasErrors()** vracia true, ak je zoznam chýb neprázdny

Validácia po odoslaní dát

- validátor môžeme do kontroléra dostať cez dependency injection
- ale môžeme použiť automatické vkladanie závislostí cez anotácie
- anotácia **@Component** nad triedou zavedie triedu do aplikačného kontextu ako bean
  - je to podobné ako v prípade @Controllera

```
@Component
public class StudentValidator implements Validator {
  ...
}
```

Validácia po odoslaní dát

- závislosť vložíme do triedy použitím anotácie **@Autowired** nad inštančnou premennou

```
public class StudentController {
  @Autowired
  private StudentValidator studentValidator;
  ...
}
```

```
@Component
public class StudentValidator implements Validator {
  ...
}
```

Ošetrovanie výnimiek

- ak nastane v systéme výnimka, používateľ uvidí celý stack trace
- to nie je ideálne chovanie, pretože
  - mätie používateľa technickými informáciami
  - ukazuje vnútro systému
- Spring MVC ponúka mapovanie výnimiek na viewy
- **handler exception resolver**
- stačí deklarovať vhodný bean

Mapovanie výnimiek na viewy

```
<bean id="exceptionHandler"
      class="org.springframework.web.servlet.handler.
             SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="IllegalArgumentException">
        unknownEntity
      </prop>
    </props>
  </property>
  <property name="defaultErrorView" value="error" />
</bean>
```

- mapuje **IllegalArgumentException** na view **unknownEntity**
- všetky ostatné výnimky zobrazia view **error**