Róbert Novotný

13. 4. 2010, seminár Bez(a)DiS

Java a web

- Java je **prosto** dobrá voľba na web
- moderné technológie na tepe doby
- štandardne používaná na enterprise projekty
- výborná platforma
- plejáda technológií na výber

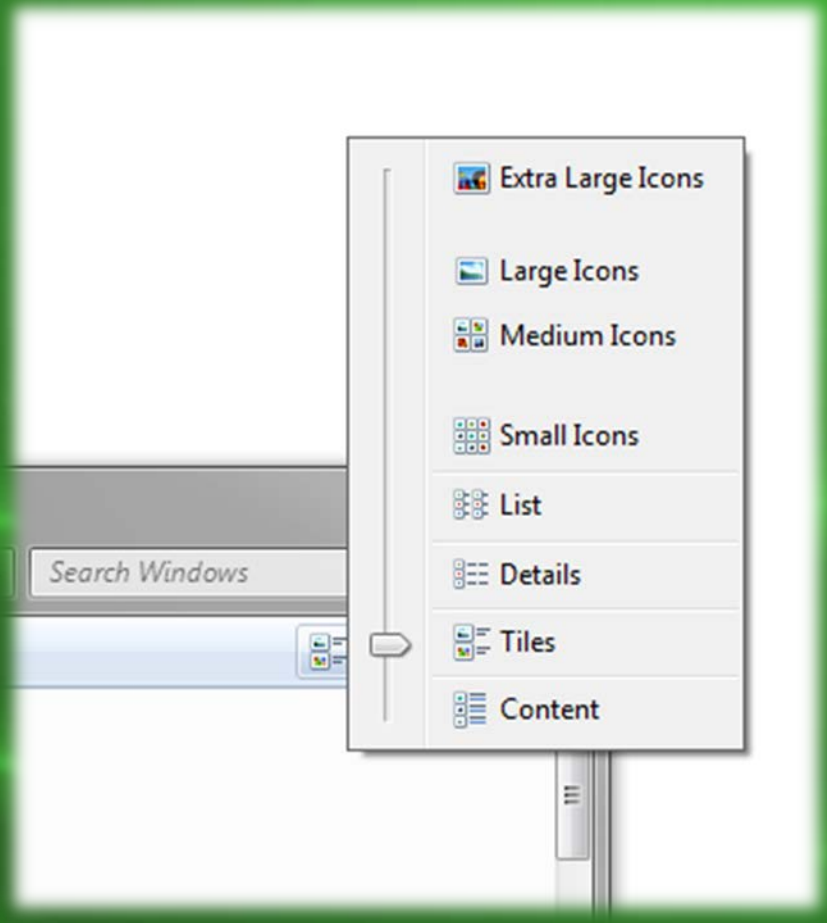
Syndróm nedeľných New York Times

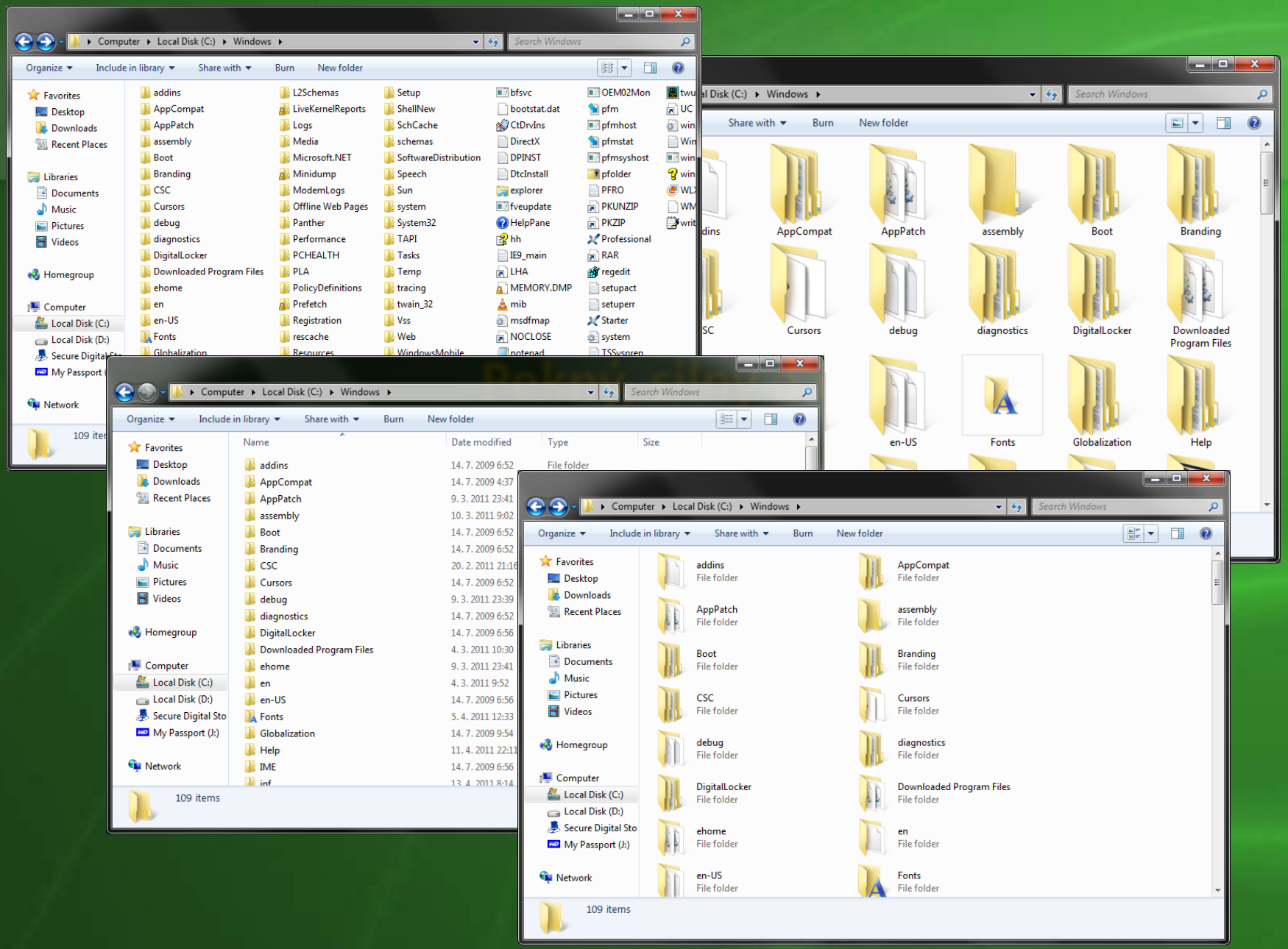
Poučenie z vývoja a pohľad do histórie

- 1987: návrh rozhraní v jazyku SmallTalk
- snaha o jasné **oddelenie**
  - prezentačnej vrstvy (používateľské rozhranie)
  - aplikačnej logiky
  - dát
- vzniká návrhový vzor **MVC**
  - **M**odel = dáta
  - **V**iew = prezentácia
  - **C**ontroller = aplikačná logika

Pekný, silný a bystrý

- základom sú dáta
  - objekty, zoznamy objektov
- dáta sa dajú zobraziť pestrým spôsobom
- spôsob záleží od formy prezentácie




Computer ▸ Local Disk (C:) ▸ Windows ▸
Search Windows
Organize
Include in library
Share with
Burn
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Homegroup
Computer
Local Disk (C:)
Local Disk (D:)
Secure Digital Sto
My Passport (J:)
Network
addins
AppCompat
AppPatch
assembly
Boot
Branding
CSC
Cursors
debug
diagnostics
DigitalLocker
Downloaded Program Files
ehome
en
en-US
Fonts
Globalization
Help
IME
L2Schemas
LiveKernelReports
Logs
Media
Microsoft.NET
Minidump
ModemLogs
Offline Web Pages
Panther
Performance
PCHEALTH
PLA
PolicyDefinitions
Prefetch
Registration
rescache
Setup
ShellNew
SchCache
schemas
SoftwareDistribution
Speech
Sun
system
System32
TAPI
Tasks
Temp
tracing
twain_32
Vss
Web
WindowsMobile
bfsvc
bootstat.dat
CtDrvIns
DirectX
DPINST
DtcInstall
explorer
fveupdate
HelpPane
hh
IE9_main
LHA
MEMORY.DMP
mib
msdfmap
NOCLOSE
notepad
OEM02Mon
pfm
pfmhost
pfmstat
pfmsyshost
pfolder
PFRO
PKUNZIP
PKZIP
Professional
RAR
regedit
setupact
setuperr
Starter
system
TSSysprep
Name
Date modified
Type
Size
File folder
14. 7. 2009 6:52
14. 7. 2009 4:37
9. 3. 2011 23:41
10. 3. 2011 9:02
20. 2. 2011 21:16
9. 3. 2011 23:39
14. 7. 2009 6:56
4. 3. 2011 10:30
4. 3. 2011 9:52
14. 7. 2009 6:56
5. 4. 2011 12:33
14. 7. 2009 9:54
11. 4. 2011 22:11
109 items

Piliere MVC

- **Model** manažuje dáta z aplikačnej domény, zasiela informácie o stave vrstve *view* a upravuje tieto dáta na základe akcií vyvolaných v *controlleri*.
- **View** reprezentuje používateľské rozhranie, ktoré zobrazuje dáta poskytované *modelom*.
- **Controller** spracováva používateľov vstup a podľa potreby aktualizuje *model* a odosiela ho do *view* vrstvy.

MVC a HTTP: dve kamarátske filozofie

požiadavka na server

odpoveď:

1. server vytiahne dáta z biznis logiky
2. vloží ich do modelu
3. vybuduje view: model zlúči so šablónou stránky
4. výsledok zapíše do HTTP odpovede

MVC a HTTP: dve kamarátske filozofie

požiadavka na server:

1. server určí kontrolér, ktorý spracuje požiadavku
2. vytiahne parametre a hodnoty z HTTP požiadavky
3. namapuje ich na model
4. model pošle kontroléru

Spring MVC

- aplikačný rámec pre vývoj webových aplikácií
- je súčasťou Spring Frameworku
- typické črty
  - anotácie
  - minimum XML
  - konvencia nad konfiguráciou

Spring MVC

kontroléry:

- kontrolérom sa môže stať ľubovoľná trieda
- netreba dediť, implementovať
- stačí dodať pár anotácií

view:

- štandardne sú používané JSP
- tie však obsahujú minimum kódu

model:

- dáta lietajúce medzi viewom a kontrolérom
- modelom sa môže stať ľubovoľná inštancia

Spring MVC – čo potrebujeme

- potrebujeme stiahnuť:
  - JAR súbory z projektu Spring Framework
    - http://www.springframework.com
  - 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

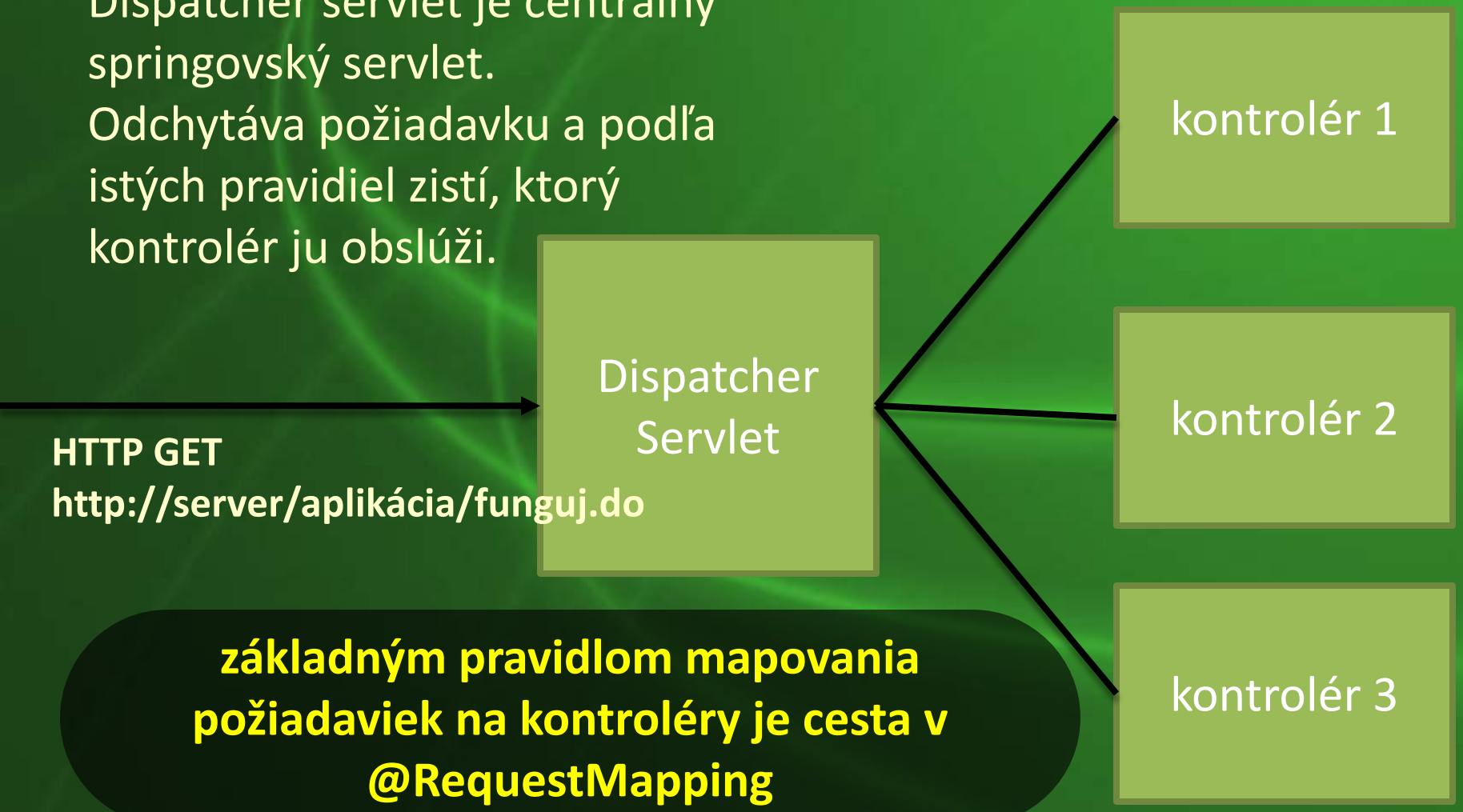
Spring MVC – architektúra

```
@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 **spring-mvc-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.

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 odlíšiť

Metódy s návratovou hodnotu

- metódy kontrolérov môžu vracať objekty
- tie sa vyhlásia za model a odošlú do view vrstvy

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

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.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-ann-requestmapping-arguments

Ako sa zobrazí view vrstva?

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

Ako sa zobrazí view vrstva?

- anotácia @RequestMapping navyše určuje meno view, ktorý sa má zobraziť po návrate z metódy

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

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

- 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/**lshowStudent.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**
- model je vlastne mapa
  - názov premennej -> hodnota
- návratové hodnoty metód v kontroléroch sa vkladajú do modelovej mapy
- konvencie:
  - sk.spring.mvc.**Student** -> "**student**"
  - **List**<**Student**> -> "**studentList**"

Model a šablóna

- ak metóda vráti **List<Student>**, do modelovej mapy sa vloží dvojica ("studentList", objekt z metódy)
- v JSP sa potom môžeme odkazovať cez bodkovú notáciu
- ${student.id}: získa z mapy hodnotu pre kľúč student a zavolá na nej getId()
- syntax zodpovedá syntaxi jazyka 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** a **option**
rozbaľovací zoznam	combobox	**select** a **option**
tlačidlá	button	<**input**>

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***

HTTP GET	HTTP POST
dvojice odosielané v URL: ?meno=John&priezvisko=Doe&vek=25	dáta odosielaná v tele požiadavky
bookmarkable URL: adresa nesie všetky dáta potrebné pre zobrazenie stránky	-
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 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, ktorého dáta sa vrátia do kontroléra
  - 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;
    }

        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 modelový objekt.

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

```
<form:form modelAttribute="student">
...
</form:form>
```

Po zadaní knihy chceme zobraziť ich zoznam. Nezobrazíme štandardný view **createStudent**, ale presmerujeme požiadavku na kontrolér obsluhujúci **/listStudents.do**

Sumarizácia

- máme **dve** metódy **createStudent()**
- obe namapované na URL končiacu na **/createStudent.do**
- jedna bez parametra, volaná cez HTTP **GET**, zobrazuje prázdny formulár
- druhá s parametrom, volaná cez HTTP **POST**, spracováva odoslané dáta

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

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

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

Realizácia v kontroléri

- pokojne môžeme využiť existujúci view **createStudent**, obsahuje totiž identické formulárové položky
  - v reálnej aplikácii by to bolo asi naopak: vytváranie by používalo ten istý formulár ako úprava
- štandardná konvencia je
  - názov view sa odvodí z URL adresy
  - model sa vytvorí automaticky, kľúč podľa mena triedy, hodnota podľa návratovej hodnoty

Prechádzanie na vlastný view

- Možnosť 1:
  - metóda **vráti String** s logickým názvom view
  - metóda obsahuje **parameter** typu **Model**, do ktorého vložíme objekty
- Možnosť 2:
  - metóda vráti **ModelAndView**
  - trieda, v ktorej nastavíme logický názov view a vieme do nej vkladať objekty

Realizácia v kontroléri: možnosť 1

- alternatíva k inštancii ModelAndView
- návratovou hodnotou bude String s názvom view
- do metódy pridáme premennú typu Model
- do nej vložíme objekty modelu

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

Realizácia v kontroléri: možnosť 2

```
@RequestMapping("/editStudent.do");
public ModelAndView editStudent(Integer id) {
  Student student = findStudentById(id);
  ModelAndView mav = new ModelAndView("createStudent")
  mav.addObject(student)
  return mav;
}
```

- zobrazí sa view **createStudent** (= createStudent.jsp)
- vložíme doň inštanciu študenta **student**
  - názov premennej sa odvodí z názvu triedy ako v klasickom prípade
  - názov premennej je možné špecifikovať aj ručne

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**

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

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 pomocou JSR-303

- špecifikácia JSR-303 generalizuje API pre validáciu beanov

Idea!

Dodajme anotácie k inštančným premenným beanov!

- validátory spĺňajúce špecifikáciu sú použiteľné vo viacerých projektoch
- jedna z implementácií JSR-303: **Hibernate Validator**

Validácia pomocou JSR-303

```
public class Student {
  @NotEmpty private String firstName;
  @NotEmpty private String lastName;
  @Min(1) @Max(5) private int year;
  @Past private Date birthDate;
  /…/
}
```

- anotácia zodpovedá obmedzeniu (constraint)
- preddefinované v balíčku **javax.validation.constraints**
- možnosť definovať vlastné obmedzenia

Spring MVC a zapnutie JSR-303

```
<mvc:annotation-driven />
```

- do aplikačného kontextu dodáme jediný riadok
- a validácia je zapnutá!
  - samozrejme, musíme stiahnuť a pridať do projektu JARy Hibernate Validatora

Kontroléry a JSR-303

validuj bean!

```
@RequestMapping(method=RequestMethod.POST,
                value="/editStudent.do")
public void saveStudent(@Valid Student student,
                        BindingResult bindingResult)
{
  if(bindingResult.hasErrors()) {
    return "createStudent";
  }
  aktualizuj(student);
  return "redirect:listStudents.do";
}
```

zoznam validačných chýb

- metóda **hasErrors()** vracia **false**, ak validácia uspela
  - t. j. zoznam validačných chýb je prázdny
- parameter **BindingResult** musí nasledovať hneď za validovaným beanom!

Formuláre a zobrazenie validačných chýb

```
<form:form commandName="student">
      <form:errors path="*" />
    ...
</form:form>
```

zobrazenie všetkých validačných chýb!

```
<form:input path="firstName" />
<form:error path="firstName" />
```

zobrazenie chyby pre konkrétne políčko

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**

Rekapitulácia

- vieme **veľmi rýchlo** vybudovať webovú aplikáciu
- relatívne **málo kódu**
- relatívne **veľa konvencií**
- množstvo vecí sa deje „**magicky**" na pozadí
- ak to prestane fungovať, začiatočníci môžu mať problém zistiť, kde je skutočná príčina chyby

Rozšírenia frameworku

- podpora **REST** architektúry
  - priamo zabudovaná do frameworku
- podpora **Ajaxu**
  - na strane servera možno jednoducho posielať JSON
  - komponenty si však treba urobiť po svojom
- **Spring Flow**:
  - podpora komplexných tokov
  - sprievodcovia, transakcie

Načo sa hodí Spring MVC?

- vhodný na portály, kde sa informácie viac prezentujú než zadávajú
- na miestach, kde nie je príliš komplexné používateľské rozhranie
- zrejme **najvyšľachtenejší kôň** stajne klasických Java web frameworkov typu Struts [2], Stripes

Otázky?