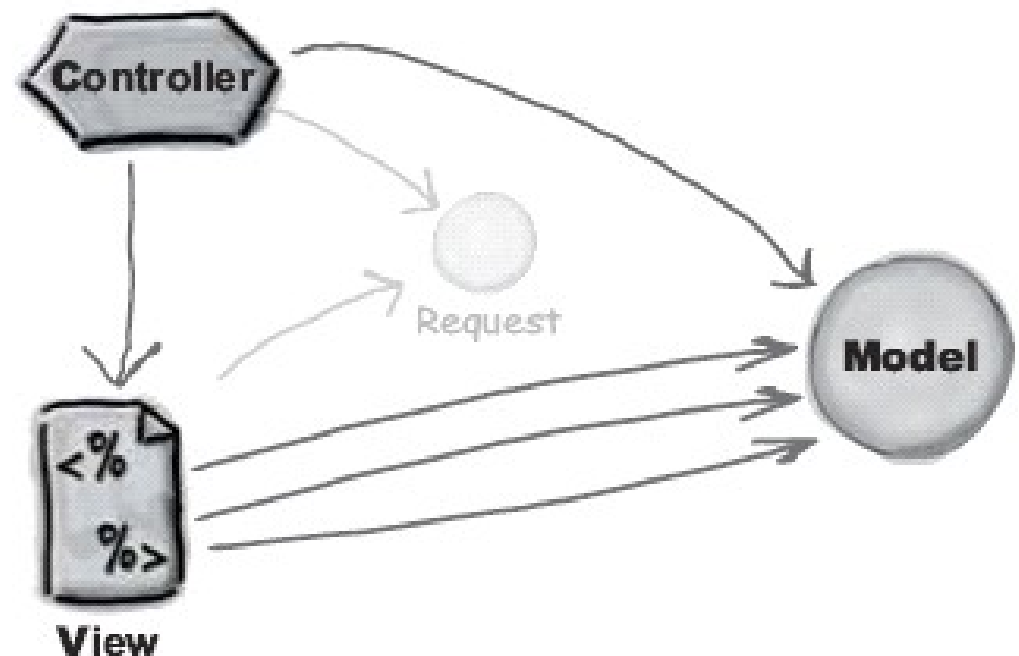


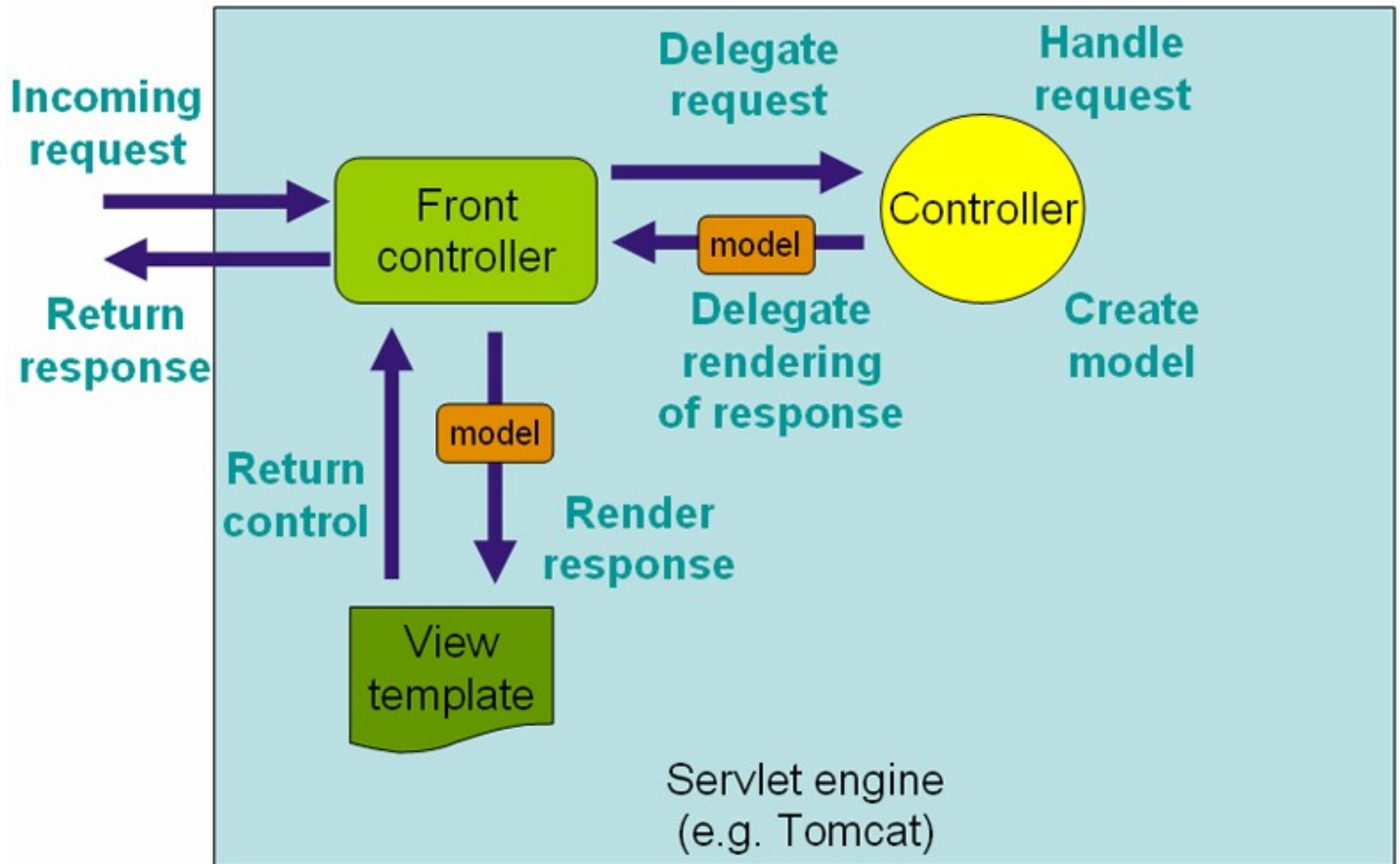
# Spring Web MVC

# Spring MVC

- Spring poskytuje vlastní implementaci návrhového vzoru MVC:
  - **Spring Web MVC** pro tvorbu klasických webových aplikací.
- **Návrhový vzor MVC (Model View Controller)** umožňuje rozdělit webovou aplikaci do tří částí, které jsou na sobě vzájemně nezávislé (Model, View a Controller).



# Front Controller



# Spring Boot

- Se Spring Boot stačí přidat dependency `spring-boot-starter-web` a máte Spring Web MVC nakonfigurované. Dříve se to konfigurovalo ručně (popsané na následujících třech stránkách). V principu se ale ve Spring Boot také používá `DispatcherServlet`, jenom ho nemusíme konfigurovat explicitně :-)

# Legacy: DispatcherServlet I.

- DispatcherServlet se jako každý servlet, na který jsou přímo zasílány požadavky, musí konfigurovat ve web.xml:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>

  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.html</url-pattern>
  <url-pattern>*.htm</url-pattern>
  <url-pattern>*.xml</url-pattern>
  <url-pattern>*.json</url-pattern>
</servlet-mapping>
```

Na jakých příponách webových stránek  
bude Servlet naslouchat

- Tento dispatcher servlet bude mít svůj konfigurační soubor /WEB-INF/dispatcher-servlet.xml.

# Legacy: DispatcherServlet II.

- Další často používaná konfigurace DispatcherServlet:

```
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/app/servlet-context.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```



Konfigurační soubor tohoto Servletu je definovaný zde

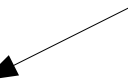
- Poznámka: V souboru web.xml můžete mít také víc Servletů.

# Legacy: DispatcherServlet III.

- Konfigurace bez XML:

```
public class MyWebAppInitializer implements WebApplicationInitializer {  
    @Override public void onStartup(ServletContext container) {  
        AnnotationConfigWebApplicationContext rootContext  
            = new AnnotationConfigWebApplicationContext();  
        rootContext.register(SpringConfiguration.class);  
        container.addListener(new ContextLoaderListener(rootContext));  
        ServletRegistration.Dynamic dispatcher =  
            container.addServlet("DispatcherServlet", new DispatcherServlet(rootContext));  
        dispatcher.setLoadOnStartup(1);  
        dispatcher.addMapping("/spring/*");  
    }  
}
```

Na tuto třídu nastavte anotaci `@EnableWebMvc`



# Spring @MVC

- Základem Spring @MVC je Controller. Jedná se o třídu, která zachytává požadavky od klienta, na jejich základě volá metody servisní vrstvy a přeposílá požadavek klienta na prezentační vrstvu.
- Nejjednodušší controller:

```
@Controller  ← Jedná se o controller
public class IndexController {
    @RequestMapping("/index") ← Zavolá se při
    public String showIndex() { požadavku
        return "/WEB-INF/jsp/index.jsp"; ← Přesune řízení na
    }                                       /WEB-INF/jsp/index.jsp
}
```

- Aby Spring tento controller našel, je nutné nastavit `<context:component-scan>` tak, aby prohledával balíček, ve kterém se tento controller nachází.



# View resolver

- Controller uvedený na předcházejícím snímku je možné ještě více zjednodušit. Je zbytečné psát všude, kde chcete přesunout řízení na JSP stránku prefix „/WEB-INF/jsp/“ a suffix „.jsp“. Z toho důvodu existuje View resolver. Stačí jeho uvedení v XML konfiguraci:

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

Nebo pomocí Java Config:

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

# Model, @RequestParam

- Objekt typu Model je mapa, do které je možné vložit atributy, ke kterým je poté možné z JSP stránky přistoupit. Spring automaticky před zavoláním metody uvozené anotací @RequestMapping nastaví hodnotu parametru Model. Pomocí anotace @RequestParam se z HTTP requestu získá parametr, který se uloží do proměnné.

HelloController.java:

```
public class HelloController {  
    @RequestMapping("/hello")  
    public String showHello (Model model,  
        @RequestParam String name) {  
        model.addAttribute("hellotxt", "Hello " + name);  
        return "hello";  
    }  
}
```

hello.jsp:

```
<div>${hellotxt}</div>
```

Vše, co jde dát dovnitř metody s anotací RequestMapping:  
<http://docs.spring.io/spring-data/jpa/docs/1.7.0.RELEASE/reference/html/#core.web>

# @PathVariable

- Část URI může být tzv. URI template:

URI template: `http://www.example.com/users/{userid}`

- `{userid}` je URI template, je možné do ní uložit nějakou hodnotu a poté ji přímo získat v controlleru:

```
public class CustomerController {  
    @RequestMapping("/customers/{customerId}")  
    public String showCustomer (@PathVariable String customerId,  
        Model model) {  
        Customer cust = customerService.findCustomer(customerId);  
        model.addAttribute(cust);  
        return "displayCustomer";  
    }  
}
```

Poznámka: Uvnitř složených zárovek může být i regulární výraz:

<http://www.logicbig.com/how-to/code-snippets/jcode-spring-mvc-requestmapping-regex/>  
<http://stackoverflow.com/questions/32791179/spring-request-mapping-with-extensions>

# Typické použití @RequestMapping

```
@Controller
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private EShopService eShopService;

    @RequestMapping
    public String listProducts(Model model) {
        model.addAttribute("products", eShopService.listProducts());
        return "products";
    }

    @RequestMapping("/{productId}")
    public String showProduct(Model model, @PathVariable Integer
        productId) {
        model.addAttribute(eShopService.getProduct(productId));
        return "product";
    }
}
```



NEBO:

```
@RequestMapping(params = { "productId" })
public String showProduct(Model model, @RequestParam
    Integer productId) {
```

# Získání objektu z formuláře

- Ve Springu je možné přetransformovat formulář na objekt. Jak na to?
- V JSP stránce je nutné nejprve přidat taglib "form":

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

- Poté se pomocí tohoto tagu vytvoří formulář:

```
<form:form action="" commandName="userOrder" method="post">
```

```
    Name: <form:input path="name" />
```

```
    <input type="submit" value="Odeslat" />
```

```
</form:form>
```

action a method atributy  
není nutné uvádět,  
v příkladu je uvedeno  
jejich výchozí nastavení

- A nakonec je nutné v controlleru zapojit pomocí anotace @ModelAttribute tento formulář.

# @ModelAttribute I.

- Anotace @ModelAttribute má dvě použití v controllerech:
  - 1) V případě, že tuto anotaci použijete před definicí metody, pak tato metoda bude sloužit pro předpřipravení objektu.
  - 2) Když ji použijete před parametrem metody, pak tato anotace uloží do parametru metody instanci objektu s daty z vyplněného formuláře.

```
@ModelAttribute("userOrder")
public UserOrder create() {
    return new UserOrder();
}

@RequestMapping(method=RequestMethod.POST)
public String saveOrder(@ModelAttribute UserOrder userOrder) {
    userOrderService.save(userOrder);
    return "redirect:/order.html";
}
```

# @ModelAttribute II.

- Poznámka: První způsob použití anotace @ModelAttribute navíc nemusíte používat, tyto dva způsoby jsou ekvivalentní:
- První způsob je preferovaný

(1)

```
@ModelAttribute("userOrder")
public UserOrder create() {
    return new UserOrder();
}

@RequestMapping
public String show() {
    return "order";
}
```

Při požadavku na zobrazení stránky s formulářem se vloží objekt do modelu ručně

(2)

```
@RequestMapping
public String show(Model model) {
    model.addAttribute("userOrder", new UserOrder());
    return "order";
}
```

# Redirect Attributes (Flash Scope)

- Po odeslání formuláře je best practice zobrazit informaci o výsledku. Můžete to udělat buď pomocí request parametrů, nebo od Spring 3.1 můžete použít redirect atributy (známé také jako flash scope):
- Controller:

```
@RequestMapping(method=RequestMethod.POST)  
public String saveOrder(@ModelAttribute UserOrder userOrder,  
                        RedirectAttributes redirectAttributes) {  
    userOrderService.save(userOrder);  
    redirectAttributes.addFlashAttribute("success", "true");  
    return "redirect:/order.html";  
}
```

- JSP:

```
<c:if test="${success eq true}">  
    <div class="alert alert-success">Order saved!</div>  
</c:if>
```



Order saved!



# @SessionAttributes I.

- Když potřebujeme uložit atribut do session scope, tak také můžeme použít přístup jako na předcházejícím obrázku, jenom musíme před třídu přidat anotaci @SessionAttributes, která specifikuje, které atributy bude Spring Web MVC ukládat do session.
- Tímto způsobem bychom si mohli vylepšit práci s košíkem:

```
@Controller
@RequestMapping("/basket")
@SessionAttributes("basket")
public class BasketController {

    @Autowired
    private ItemService itemService;
```

**Pokračování**



# @SessionAttributes II.

```
@ModelAttribute("basket")
```

```
public Basket construct() { return new Basket(); }
```

```
@RequestMapping
```

```
public String showBasket() { return "basket"; }
```

```
@RequestMapping("/add")
```

```
public String add(@RequestParam int id, @RequestParam int quantity,  
                  @ModelAttribute Basket basket) {  
    basket.add(itemService.findOne(id), quantity); return "redirect:/items.html";  
}
```

```
@RequestMapping("/remove")
```

```
public String remove(@RequestParam int id, @ModelAttribute Basket basket) {  
    basket.remove(id); return "redirect:/basket.html";  
}
```

```
}
```

# @SessionAttributes III.

- Třída Basket:

```
public class Basket {  
    private Map<Integer, OrderedItem> orderedItems = new HashMap<Integer, OrderedItem>();  
    public void add(Item item, int quantity) {  
        int itemId = item.getItemId();  
        if (orderedItems.containsKey(itemId)) {  
            OrderedItem orderedItem = orderedItems.get(itemId);  
            orderedItem.setQuantity(orderedItem.getQuantity() + quantity);  
        } else {  
            OrderedItem orderedItem = new OrderedItem();  
            orderedItem.setQuantity(quantity);  
            orderedItem.setItem(item);  
            orderedItems.put(orderedItem.getItem().getItemId(), orderedItem);  
        }  
    }  
}
```

**Pokračování**



# @SessionAttributes IV.

```
public void remove(int itemId) {  
    orderedItems.remove(itemId);  
}
```

```
public Iterable<OrderedItem> getItems() {  
    return orderedItems.values();  
}  
}
```

# SessionAttributes V.

- Kdekoli chcete pracovat s atributy ze session přidejte k definici controlleru anotaci @SessionAttributes:

```
@Controller
```

```
@RequestMapping("/order")
```

```
@SessionAttributes("basket")
```

← Tento objekt není výsledek formuláře,  
ale je získán ze session!

```
public class OrderController {
```

```
    public String save(@ModelAttribute @Valid UserOrder userOrder,  
                      BindingResult bindingResult, @ModelAttribute Basket basket) {
```

```
        // pokračovani ...
```

```
    }
```

```
    // pokračovani ...
```

```
}
```

# @ExceptionHandler I.

- V případě vyhození výjimky v controlleru je možné tuto výjimku zachytit pomocí anotace @ExceptionHandler:

```
public class ItemNotFoundException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
    public ItemNotFoundException(String message) {  
        super(message);  
    }  
}  
  
@ExceptionHandler  
@ResponseStatus(HttpStatus.NOT_FOUND)  
public void handleINFException(ItemNotFoundException ex) {  
  
}
```

**Pokračování**




# @ExceptionHandler II.

```
@RequestMapping("/{id}")
```

```
@ResponseBody
```

```
public Item detail(@PathVariable int id) {  
    Item item = itemService.findOne(id);  
    if (item == null) { throw new ItemNotFoundException("Item not found!"); }  
    return item;  
}
```

Poznámka: ještě lepší je  
dát to do servisní vrstvy



- Metoda, před kterou se nachází tato anotace, je obdobně flexibilní jako metoda s anotací @RequestMapping. Může vrátet String, ModelAndView, nebo také může přímo nastavovat návratový error kód apod.

# @ExceptionHandler III.

- Je navíc možné ExceptionHandler specifikovat ve třídě s anotací @ControllerAdvice, čímž ošetřuje výjimky nejenom v jednom Controlleru, ale v jejich skupině (ve výchozím nastavení pro všechny Controllery):
  - <https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc>



# REST architektura I.

- REST architektura byla navržena v roce 2000 Roy Fieldingem, jedním z tvůrců HTTP protokolu.
- REST je použitelné pro jednotný a jednoduchý přístup ke zdrojům (resources).
- REST přístup je alternativa vůči XML-RPC a SOAP.
- Všechny zdroje mají vlastní identifikátor URI a REST definuje čtyři základní metody pro přístup k nim:
  - GET – získání dat
  - POST – přidání dat
  - DELETE – mazání dat
  - PUT – editace dat

# REST architektura II.

- Při práci s REST architekturou si klient a server vyměňují plain JSON (XML) soubory.
- Pro práci s REST architekturou je zapotřebí pouze HTTP klient. GET požadavky jednoduše zvládá i webový prohlížeč.
- Naprogramování REST architektury na serverové straně je rozšířením stávajícího přístupu pomocí Spring Web MVC, kdy se z controlleru z metody s anotací `@RequestMapping` nevrací JSP stránky, ale přímo objekty.
- Pro mapování mezi objekty a XML soubory se používá JAXB, pro mapování JSONů se používá Jackson
- Na následujících stránkách je kompletní příklad REST architektury včetně klienta, který pracuje s REST serverem pomocí třídy `RestTemplate`

# RestItem

@XmlElement

← Musí být (pokud chcete vracet XML)!

```
public class RestItem {  
    private String name; private String description; private int itemId;  
    public RestItem() { }  
    public RestItem(Item item) {  
        name = item.getName(); description = item.getDescription(); itemId = item.getItemId();  
    }  
    public Item getItem() {  
        Item item = new Item(); item.setName(name); item.setDescription(description);  
        item.setItemId(itemId); return item;  
    }  
    @XmlElement(name = "item-id")  
    public int getItemId() { return itemId; }  
    // dalsi gettery a settery  
}
```

Mapování Item → RestItem

Mapování RestItem → Item

Jeden z getterů, je možné změnit  
výchozí mapování atributů na XML

## Poznámka:

Tento objekt je tzv. DTO (Data Transfer Object).  
Mapování entity na DTO je ručně v kódu.  
Pokud byste používali DTO objekty víc, pak  
se podívejte na projekt DOZER:  
<http://dozer.sourceforge.net/>

# RestItemList

```
@XmlElement(name = "items")  
  
public class RestItemList {  
    private List<RestItem> restItems;  
  
    public RestItemList() { }  
  
    public RestItemList(Iterable<Item> items) {  
        restItems = new ArrayList<RestItem>();  
        for (Item item : items) {  
            restItems.add(new RestItem(item));  
        }  
    }  
  
    @XmlElement(name = "item")  
    public List<RestItem> getRestItems() { return restItems; }  
    public void setRestItems(List<RestItem> items) { this.restItems = items; }  
}
```

← Musí být (pokud chcete vracet XML)!

← Mapování kolekce Items na RestItemList

# RestController I.

```
@Controller
```

```
@RequestMapping("/rest/items")
```

```
public class RestItemController {
```

```
    @Autowired
```

```
    private ItemService itemService;
```

Vrací se objekt a na výstupu je anotace `ResponseBody`

```
    @RequestMapping
```

```
    @ResponseBody public RestItemList list() {
```

```
        return new RestItemList(itemService.findAll());
```

```
    }
```

```
    @RequestMapping("/{itemId}")
```

```
    @ResponseBody public RestItem item(@PathVariable int itemId) {
```

```
        return new RestItem(itemService.findOne(itemId));
```

```
    }
```

← Výchozí operace GET

← Používá se `PathVariable`

**Pokračování**



# RestController II.

```
@RequestMapping(method = RequestMethod.POST)  
@ResponseBody public RestItem add(@RequestBody RestItem restItem) {  
    return new RestItem(itemService.save(restItem.getItem()));  
}  
  
@ResponseStatus(HttpStatus.OK)  
@RequestMapping(value =("/{itemId}", method = RequestMethod.DELETE)  
public void remove(@PathVariable int itemId) { itemService.remove(itemId); }  
  
@ResponseStatus(HttpStatus.OK)  
@RequestMapping(value =("/{itemId}", method = RequestMethod.PUT)  
public void edit(@RequestBody RestItem restItem, @PathVariable int itemId) {  
    itemService.save(restItem.getItem());  
}  
}
```

Operace POST pro přidání záznamu,  
vrací nově vytvořený záznam

Získává se objekt, na vstupu  
je anotace RequestBody

Je možné nastavit  
návratovou hodnotu

Operace DELETE  
maže záznam

Operace PUT  
edituje záznam

# RestController & Get/PostMapping ...

- Od Spring 4.0 existuje anotace `@RestController`. Pokud ji použijete místo `@Controller`, pak nebudete muset u jednotlivých metod psát anotace `@RequestBody` / `@ResponseBody`.
- Od Spring 4.3 existují anotace jako `@GetMapping` a `@PostMapping`, které jsou náhradou pro:
  - `@RequestMapping(method = RequestMethod.GET)`
  - `@RequestMapping(method = RequestMethod.POST)`
  - ...

# OpenAPI (Swagger)

- Pro dokumentování REST WS existuje OpenApi (Swagger).
- Jak ho použít se Spring Boot?
  - Buď SpringFox:
    - <https://springfox.github.io/springfox/docs/current/>
  - Nebo SpringDoc:
    - <https://springdoc.org/>
- Pro testování REST WS se obvykle používá:
  - Postman
  - SoapUI



# Použití na straně klienta I.

- Posílají se plain XML soubory, tudíž není zapotřebí nic speciálního, jenom HTTP klient.
- Na straně klienta je nutné vytvořit Java soubory, které odpovídají tomu, co posílá server.
- Spring obsahuje třídu `RestTemplate`, která výrazně zjednodušuje práci se serverem pracujícím na REST principech:

```
RestTemplate restTemplate = new RestTemplate();
```

- Získání jednoho záznamu:

```
RestItem restItem = restTemplate.getForObject(  
    "http://localhost:8080/rest/items/1.xml", RestItem.class);
```

- Získání všech záznamů:

```
RestItemList restItemList = restTemplate.getForObject(  
    "http://localhost:8080/rest/items.xml", RestItemList.class);
```

# Použití na straně klienta II.

- Přidání záznamu:

```
RestItem restItem = new RestItem();  
restItem.setDescription("new description");  
restItem.setName("new name");  
RestItem restItem2 = restTemplate.postForObject(  
    "http://localhost:8080/rest/items.xml", restItem, RestItem.class);  
System.out.println("Newly added item: " + restItem2.getItemId());
```

- Odebrání záznamu:

```
restTemplate.delete("http://localhost:8080/rest/items/9.xml");
```

- Editace záznamu:

```
RestItem restItem= restTemplate.getForObject(  
    "http://localhost:8080/rest/items/1.xml", RestItem.class);  
restItem.setDescription("new description v2");  
restTemplate.put("http://localhost:8080/rest/items/1.xml", restItem);
```

# RestTemplate & authorization

```
private static HttpHeaders createHeaders(String username, String password) {  
    return new HttpHeaders() {  
        { String auth = username + ":" + password;  
          byte[] encodedAuth = Base64.getEncoder()  
              .encode(auth.getBytes(Charset.forName("US-ASCII")));  
          String authHeader = "Basic " + new String(encodedAuth);  
          set("Authorization", authHeader); }  
    };  
}  
  
public static void main(String[] args) {  
    RestTemplate restTemplate = new RestTemplate();  
    restTemplate.exchange("URL", HttpMethod.POST,  
        new HttpEntity<String>(createHeaders("USER", "PASSWORD")), String.class);  
}
```

# RestTemplate & List

- Jak získat ze serveru list záznamů:
  - <http://stackoverflow.com/questions/23674046/get-list-of-json-objects-with-spring-resttemplate>

# WebClient

- Spring 5 přišel s novou náhradou RestTemplate : WebClient
  - <https://www.baeldung.com/spring-5-webclient>

# Bean Validation

- Spring 3 obsahuje plnou podporu pro Bean Validation API, které standardizuje validaci doménových objektů. Pomocí tohoto API je možné pomocí anotací deklarativně definovat omezení atributů objektů a nechat runtime toto omezení kontrolovat:

**Předtím:**

```
public class Person {  
    private String name;  
    private int age;  
}
```

**Potom:**

```
public class Person {  
    @NotNull  
    @Size(max=20)  
    private String name;  
  
    @Min(0)  
    private int age;  
}
```

- Anotací není mnoho, všechny jsou v balíčku `javax.validation.constraints`, nejpokročilejší je anotace `@Pattern`, která umožňuje validovat na základě výsledku regulárního výrazu.

# Rozchození Bean Validation

- Nejprve je nutné do classpath přidat implementaci JSR-303, jako je Hibernate validator:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>
```

**Pokračování**



# Zobrazení chyb JSR-303 validace

- Každá validující anotace má výchozí chybovou zprávu. Tuto zprávu je možné v JSP stránce vypsat:

```
<form:errors path="name" />
```

- Tuto výchozí zprávu je vhodné předefinovat. Můžeme to udělat buď jednoduše v kódu:

```
@Size(min = 5, max = 20,  
      message="Jméno musí být v rozsahu {min} - {max} znaků!")  
  
private String name;
```

- Nebo v případě požadavku na internacionalizaci aplikace můžeme uvést `message="{klic_chybove_hlasky}"` a přidat do classpath soubor s názvem `ValidationMessages.properties`, ve kterém budou překlady chybových hlášek.



# Validace bez Spring Web MVC

- Knihovna JSR 303 není závislá na Spring Web MVC, můžete také validovat tímto způsobem:

```
@Service
public class MyService {
    @Autowired private Validator validator;

    public void myMethod() {
        Set<ConstraintViolation<UserOrder>> set = validator.validate(object);
        if (!set.isEmpty()) {
            for (ConstraintViolation<UserOrder> constraintViolation : set) {
                System.out.println(constraintViolation.getPropertyPath() + " has error: "
                    + constraintViolation.getMessage());
            }
        }
    }
}
```

javax.validation.Validator

Objekt, který obsahuje JSR 303 anotace

# Hibernate Validator

- Je také možné tvořit vlastní validace, viz. dokumentace:
  - <http://docs.spring.io/spring/docs/3.2.4.RELEASE/spring-framework-reference/html/validation.html#validation-beanvalidation>
- Pro další pokročilejší témata se podívejte na Hibernate Validator dokumentaci:
  - <http://www.hibernate.org/subprojects/validator/docs>

# Spring taglib

- Spring obsahuje Spring taglib:

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

- V tomto taglibu je několik užitečných tagů.

- Například:

```
<spring:url value="/" var="contextUrl"/>
```

Uloží URL do atributu, který je přístupný přes EL: `${contextUrl}`

Vygeneruje URL včetně kontextu web. aplikace.

Náhrada: `<%= getServletContext().getContextPath() + "/" %>`

# i18n (internationalization) I.

- Do src/main/resources přidejte soubory:
  - messages.properties: Překladové texty pro češtinu
  - messages\_en.properties: Překladové texty pro angličtinu
- Přidejte do Servlet kontextu:

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/*" />
    <bean id="LocaleChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
      <property name="paramName" value="language" />
    </bean>
  </mvc:interceptor>
</mvc:interceptors>
```

**Poznámka:**  
Překladové texty  
jsou ve formátu:  
klic=hodnota

**Pokračování**



# i18n (internationalization) II.

```
<bean id="LocaleResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
    <property name="defaultLocale" value="cs" />
</bean>

<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="messages" />
</bean>
```

messages.properties v classpath



- Nyní když na jakékoli stránce přejdete na URL ?language=en, tak přepnete jazyk do angličtiny. Jakýkoli jiný ?language= přepne jazyk do češtiny.
- V JSP stránkách překladové texty použijete následovně:

```
<spring:message code="klic" />
```

# i18n (internationalization) III.

- Klíče můžete získávat i programově:

```
@Autowired
```

```
private MessageSource messageSource;
```

```
public void myMethod() {
```

```
    String message = messageSource.getMessage("klic", null, new Locale("en"));
```

```
}
```

- Jak na internacionalizaci dynamických textů které vkládá do databáze klient? Na to není jednoduchá odpověď (řešení se liší projekt od projektu), každopádně výše uvedeným způsobem to nelze.

# Přidání atributu do každého modelu všech Controllerů

- Přidejte do konfigurace Servlet contextu:

```
<mvc:interceptors>
    <bean class="cz.jiripinkas.abcvideos.interceptors.SettingsInterceptor" />
</mvc:interceptors>
```

- Vytvořte třídu:

```
public class SettingsInterceptor extends HandlerInterceptorAdapter {
    @Autowired private SettingsService settingsService;
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
                           Object handler, ModelAndView modelAndView) throws Exception {
        if (modelAndView != null) {
            modelAndView.getModelMap().addAttribute("TestKey", "TestValue");
        }
    }
}
```

← Přístup k servisní vrstvě

V jakémkoli JSP souboru, na který se přišlo přes Controller bude nyní k dispozici atribut: `${TestKey}`

# DeferredResult

- DispatcherServlet používá thread pool Tomcatu (ve výchozím nastavení 200 vláken). Pokud máme déle-trvající operaci, která by tento pool zbytečně blokovala, pak můžeme použít jiný (náš) pool tímto způsobem:
  - <https://www.baeldung.com/spring-deferred-result>