Distributed Applications
KU Leuven - Campus Groep T
2014-2015

- *Channak Chhorn*
- *Belay Mulatu*
- *Bunna Kal*

Spring MVC Web Framework

Short description

The Spring Framework is an open source application framework that aims to make J2EE development easier, originally created by Rod Johnson. Spring was created as an alternative to heavier enterprise Java technologies, especially EJB. It empowered plain Old Java Objects (POJOs) with powers previously only available using EJB and other enterprise Java specification. Over time, EJB started offering a simple POJO-oriented programming model of its own. Now EJB employs ideas such as dependency injection (DI) and Aspect-Oriented Programming (AOP), arguably inspired by the success of Spring. Mobile development, social API integration, NoSQL database, cloud computing, and big data are just a few areas where Spring has been and is innovating.

Spring is a framework which helps to connect different components together. There are many modules for IOC, AOP, and Web MVC etc. Spring Framework is an open source application framework and inversion of control container for the Java platform. Spring MVC (Model–view–controller) is one component within the whole Spring Framework, and also a software design pattern for developing web applications. We can use any web containers, so it works fine with Tomcat or any web servers such as Glassfish, JBoss etc. Spring MVC is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest (request, response) method. With this method will handle the response to the client requested which responsible to render the front-end with the data sent by the server. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods. In the new version of Spring, the @Controller mechanism can even allow you to create RESTful Web sites and applications just through the @PathVariable annotation.

Technical features

MVC solution

In Spring MVC web application, it consists of 3 standard MVC (Model, Views, and Controller) components:

- *Model:* The lowest level of the pattern which is responsible for maintaining data. It is a domain objects that are processed by the service layer (business logic) or persistent layer (database operation). If we compare to the EJB, it is Entity that represented data object for view.
- *View:* This is responsible for displaying all or a portion of the data to the user. Normally it's a JSP page written with the Java Standard Tag Library (JSTL).
- *Controller:* Software Code that controls the interactions between the Model and View. It is a URL mapping and interact with service layer for business processing and return a Model.

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.

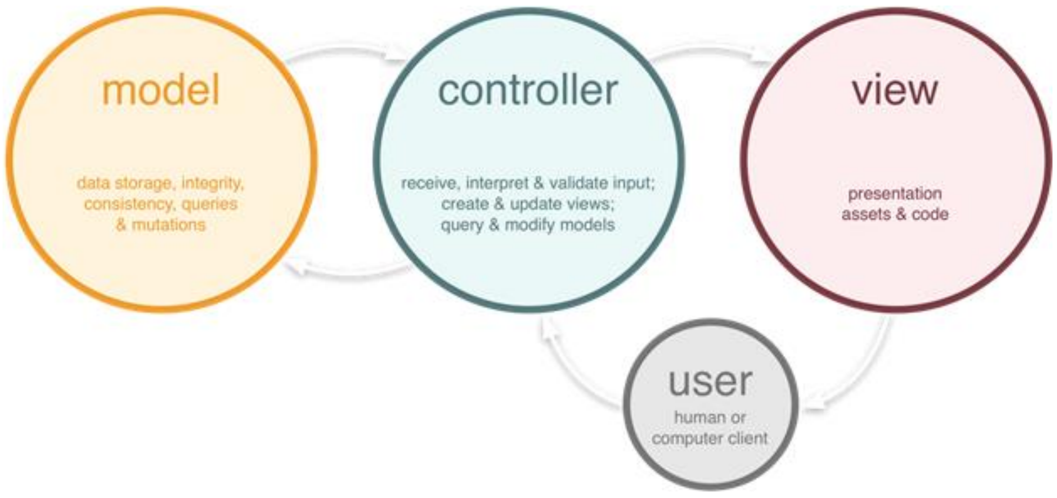

model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client

Figure 1: MVC Architecture

Now let determine the MVC mechanism which corresponding to Spring MVC Framework.

Suppose you want to develop a simple web application with Spring MVC to learn the basic concepts and configurations of this framework.

The central component of Spring MVC is a controller. In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file or a ServletContainerInitializer). A Spring MVC controller—often referred to as a Dispatcher Servlet (see figure 3)—implements one of Sun's core Java EE design patterns called front controller (see figure 2). It acts as the front controller of the Spring MVC framework, and every web request must go through it so that it can manage the entire request-handling process.

When a web request is sent to a Spring MVC application, a controller first receives the request. Then it organizes the different components configured in Spring's web application context or annotations present in the controller itself, all needed to handle the request. Figure 3 shows the primary flow of request handling in Spring MVC.

To define a controller class in Spring 4.0, a class has to be marked with the @Controller annotation. In contrast to other framework controllers or earlier Spring versions, an annotated controller class need not implement a framework-specific interface or extend a framework-specific base class.

When a @Controller annotated class (i.e., a controller class) receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class's methods are decorated with the @RequestMapping annotation, making them handler methods.

You will encounter the various method arguments that can be used in handler methods using the @RequestMapping annotation. The following is only a partial list of valid argument types, just to give you an idea.

- HttpServletRequest or HttpServleResponse
- Request parameters of arbitrary type, annotated with @RequestParam
- Model attributes of arbitrary type, annotated with @ModelAttribute
- Cookie values included in an incoming request, annotated with @CookieValue
- Map or ModelMap, for the handler method to add attributes to the model

- Errors or BindingResult, for the handler method to access the binding and validation result for the command object
- SessionStatus, for the handler method to notify its completion of session processing

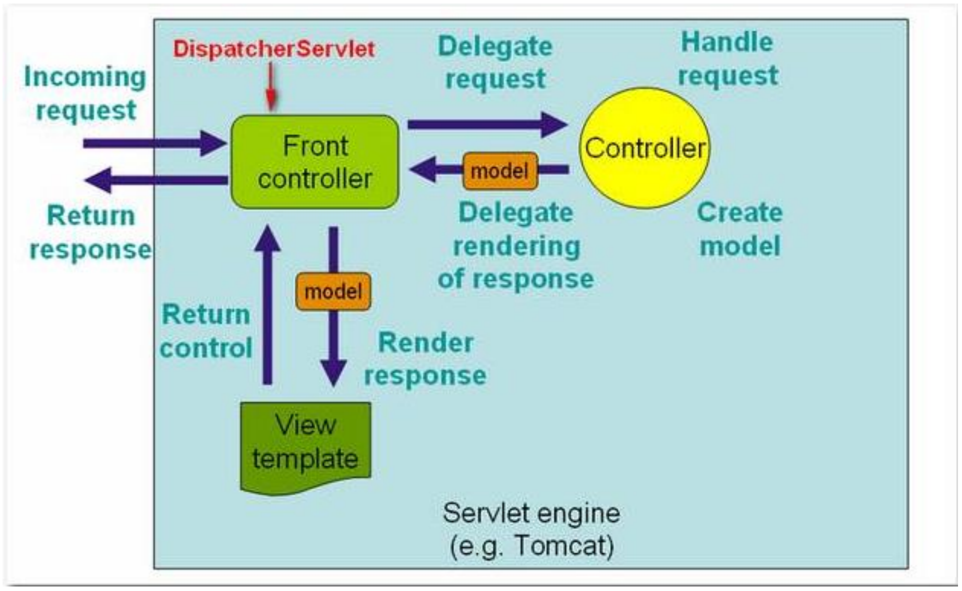


Figure 2: The requesting processing workflow in Spring Web MVC (high level)

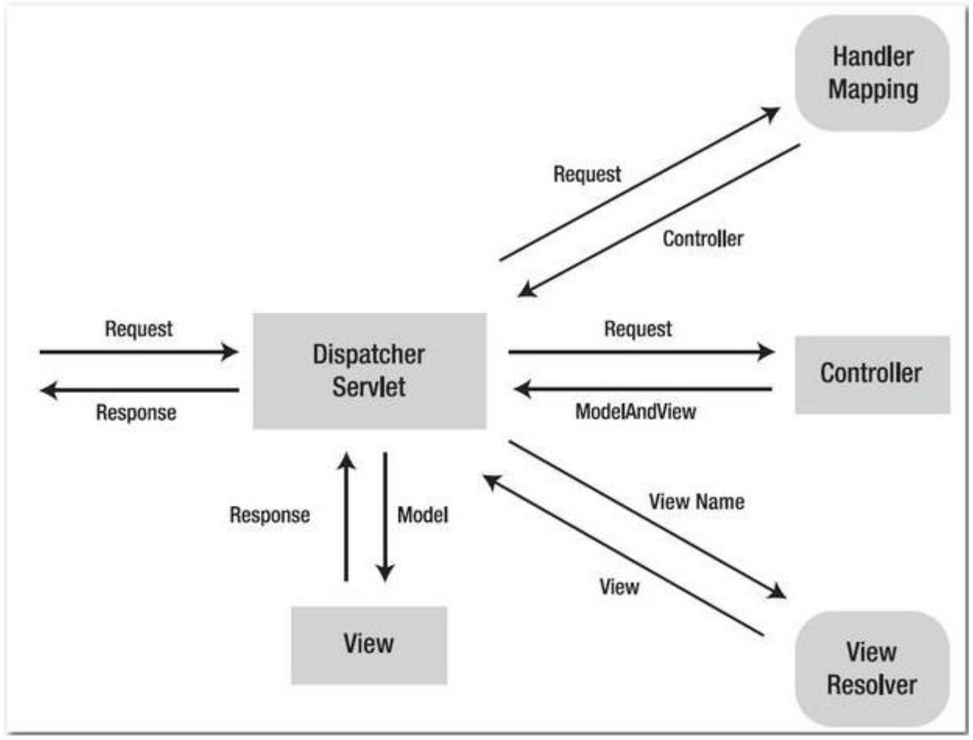


Figure 3: Primary flow of request handling in Spring MVC

Once the controller class has picked an appropriate handler method, it invokes the handler method's logic with the request. Usually, a controller's logic invokes back-end services to handle the request. In addition, a handler method's logic is likely to add or remove information from the numerous input arguments (e.g., HttpServletRequest, Map, Errors, or SessionStatus) that will form part of the ongoing Spring MVC flow.

After a handler method has finished processing the request, it delegates control to a view, which is represented as the handler method's return value. To provide a flexible approach, a handler method's return value doesn't represent a view's implementation (e.g., user.jsp or report.pdf) but rather a logical view (e.g., user or report)—note the lack of file extension.

When the controller class receives a view, it resolves the logical view name into a specific view implementation (e.g., user.jsp or report.pdf) by means of a view resolver. A view resolver is a bean configured in the web application context that implements the ViewResolver interface. Its responsibility is to return a specific view implementation (HTML, JSP, PDF, or other) for a logical view name.

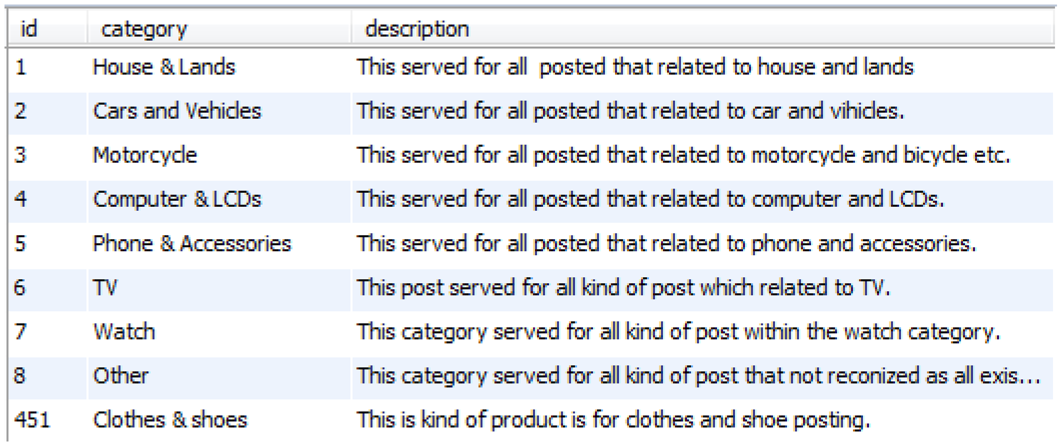
Once the controller class has resolved a view name into a view implementation, per the view implementation's design, it renders the objects (e.g., HttpServletRequest, Map, Errors, or SessionStatus) passed by the controller's handler method. The view's responsibility is to display the objects added in the handler method's logic to the user.

Now have a close look on how it implemented on this three levels of MVC

Model

```java
@Entity
@Table(name = "categories")
public class Category implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String category;

    public String getCategory() {
        return category;
    }

    public String getDescription() {
        return description;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public void setDescription(String description) {
        this.description = description;
    }
    private String description;

    public Long getId() {
        return id;
    }

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

The Category entity class (normal java class) is represented for the below table.



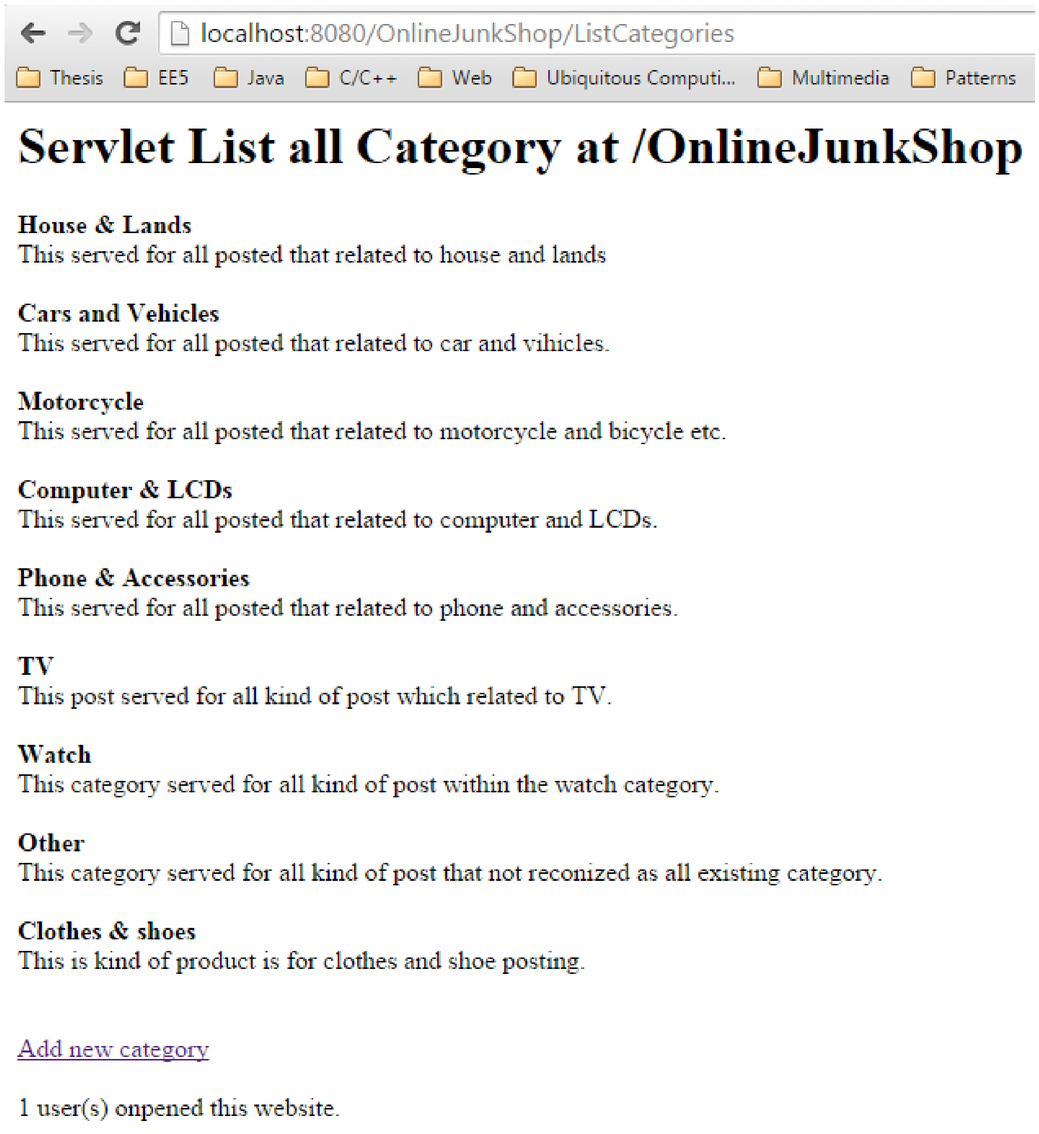
id	category	description
1	House & Lands	This served for all posted that related to house and lands
2	Cars and Vehicles	This served for all posted that related to car and vihicles.
3	Motorcycle	This served for all posted that related to motorcycle and bicycle etc.
4	Computer & LCDs	This served for all posted that related to computer and LCDs.
5	Phone & Accessories	This served for all posted that related to phone and accessories.
6	TV	This post served for all kind of post which related to TV.
7	Watch	This category served for all kind of post within the watch category.
8	Other	This category served for all kind of post that not reconized as all exis...
451	Clothes & shoes	This is kind of product is for clothes and shoe posting.

Table 1: categories table

View (JSP)

Category.jsp file representing as presentation layer in order to rendering the view for user.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@page isELIgnored="false" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Online Auction Junk Shop</title>
    </head>
    <body>
        <h1>Servlet List all Category at " + ${pageContext.request.contextPath} + "</h1>");
            <c:forEach items="${categories}" var="category">
                <b> ${category.getCategory()}</b><br />
                ${category.getDescription()}<br /><br />
            </c:forEach>
            <br/><a href='AddCategory'>Add new category</a>
            <br/><br/>
            <p> ${countViewer} user(s) opened this website </p>
    </body>
</html>
```


localhost:8080/OnlineJunkShop/ListCategories

Thesis EE5 Java C/C++ Web Ubiquitous Computi... Multimedia Patterns

Servlet List all Category at /OnlineJunkShop

House & Lands
This served for all posted that related to house and lands

Cars and Vehicles
This served for all posted that related to car and vihicles.

Motorcycle
This served for all posted that related to motorcycle and bicycle etc.

Computer & LCDs
This served for all posted that related to computer and LCDs.

Phone & Accessories
This served for all posted that related to phone and accessories.

TV
This post served for all kind of post which related to TV.

Watch
This category served for all kind of post within the watch category.

Other
This category served for all kind of post that not reconized as all existing category.

Clothes & shoes
This is kind of product is for clothes and shoe posting.

Add new category

1 user(s) onpened this website.

Figure 4: List all category as view

Controller

By using annotation @Controller this class will act as controller for this project and @RequestMapping is used to handle the matched parameter that passed by the user via URL.

```
@Controller
public class OnlineJunkShopController {
    @RequestMapping(value = "/listCategories",method = {RequestMethod.POST,RequestMethod.GET})
    public ModelAndView handlerRequest(ModelAndView mav) {
       mav.addObject("categories", this.getComponentService().getAllCategory());
       mav.addObject("countViewer", this.getCountViewer());
       mav.setViewName("category");
       return mav;
    }
    public MyComponent getComponentService() {
       ApplicationContext context =
```

```java
               new ClassPathXmlApplicationContext("XMLBeanResources/ejbBeanResources.xml");
            myComponent = (MyComponent) context.getBean("myComponent");
            return myComponent;
        }
        public int getCountViewer() {
            ApplicationContext context =
               new ClassPathXmlApplicationContext("XMLBeanResources/ejbBeanResources.xml");
            SessionBeanManager counter =
                  (SessionBeanManager) context.getBean("mySingleSessionBean");
            return counter.getActiveSessionsCount();
        }
    }
```

This below interface is used to make available call from anywhere outside the project.

```java
@Remote
public interface EntityBeanRemote {
public List<Category> getAllCategory();
}
```

This stateless class below is used to implement the remote interface and actually implement the business logic and manipulate the database.

```java
@Stateless
public class EntityBeanFacade implements EntityBeanRemote{
    @PersistenceContext(unitName = "OnlineJunkShopPU")
    private EntityManager em;
    public List<Category> getAllCategory() {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(Category.class));
        return em.createQuery(cq).getResultList();
    }
}
```

This singleton session bean use to ensure that only one install of count the number of use is created every time the new user open this URL.

```java
@Singleton
@LocalBean
@WebListener
public class SessionBeanManager implements HttpSessionListener {
    private static int counter=0;
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        counter++;
    }
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        counter--;
    }
    public int getActiveSessionsCount() {
        return counter;
    }
}
```

In order to make Spring enable to call the business logic or function which written with EJB module, we have to inject to the spring container by using xml configuration (dispatcher-servlet.xml) as below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jee="http://www.springframework.org/schema/jee"
      xsi:schemaLocation=http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee-4.0.xsd
      http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <bean id="myComponent"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
        <property name="jndiName" value="ejb/myBean"/>
        <property name="businessInterface" value="com.onlinejunkshop.ejb.EntityBeanRemote"/>
    </bean>
    <bean id="myController" class="com.onlinejunkshop.controller.OnlineJunkShopController">
        <property name="myComponent" ref="myComponent"/>
    </bean>
    <!—or use jee module -->
    <!--<jee:remote-slsb id="myComponent" jndi-name="ejb/myBean"
                   business-interface="com.onlinejunkshop.ejb.EntityBeanRemote"/>

   <bean id="myController" class="com.onlinejunkshop.controller.OnlineJunkShopController">
       <property name="myComponent" ref="myComponent"/>
   </bean>-->
    <bean id="mySingleSessionBean" class="com.onlinejunkshop.ejb.SessionBeanManager"/>
</beans>
```

More detail information on how to inject EJB into Spring see this link:
http://docs.spring.io/spring/docs/current/spring-framework-reference/html/ejb.html

As you seen above controller, when the user or client make a requesting by enter the ULR in this case the user might enter http://localhost:8080/OnlineJunkShop/ListCategories which http://localhost:8080/OnlineJunkShop is a root for web application. When it match the @RequestMapping value (i.e. /listCategory) the corresponding in the matched controller will be invoked which accept both URL and form submition (i.e. method = {RequestMethod.POST, RequestMethod.GET}). There are two objects are added to ModelAndView object with its unique identifier (i.e name, recentPosts, allPosts etc.). And the value for these object are get from database by using JPA (Java Persistent API) one of well know mechanism in EJB or Data DAO (Data Access Object) the low level for data access by using pure SQL to manipulate data in the database. For the example above we have use JPA which one feature of EJB.

Navigation

Every routing or navigation in Spring MVC framework has done easily just simply using annotation @RequestMapping which come along controller class. Navigation is done in the controller class with the annotation of @RequestMapping(value="/mapname", method = {RequestMethod.GET, RequestMethod.POST}) (sees Controller section above). The @RequestMapping annotation to map URLs such as /mapname onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

URI Template Patterns

We are not only can route or navigate within a specific URL link, but we can also give one or more parameter via URL. A URI Template is a URI-like string, containing one or more variable names. The URI Template http://www.example.com/users/{userId} contains the variable userId which can be navigated http://www.example.com/users/1. Moreover, we can pass multiple parameter as below:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId, @PathVariable String petId, Model
model) {
    Owner owner = ownerService.findOwner(ownerId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "displayPet";
}
```

When a @PathVariable annotation is used on a Map<String, String> argument, the map is populated with all URI template variables.

A URI template can be assembled from type and path level @RequestMapping annotations. As a result the findPet() method can be invoked with a URL such as /owners/42/pets/21.

There is no optional parameter which offered in Spring 4.x unless you have to create a two different @RequestMapping. For example, if you have to render a view at first time you do not need a filters, but later one you have to make it filter when click on another filer condition (i.e display all product within a categoryId given via RUL). So we have to create a separate mapping.

1. @RequestMapping(value="/home") : There is not parameter passed by the URL (Display all products).
2. @RequestMapping(value="/home/{categoryId}") : This will accept one parameter known as categoryId (display all the product which in categoryId).

However, there are also some existing solution to implement on how to make an optional parameter passed via URL. (Sees this link http://java.dzone.com/articles/spring-3-webmvc-optional-path). It seem that we have to implement everything by ourselves by create a new class which extends AntPathMatcher class.

Form handling

In order to handle the form submission in the JSP f that contains the form with a submit button we have to define a value for action attribute for a form.

By using a pure HTML Form

```
<!DOCTYPE HTML>
<head>
    <title>Handing Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <h1>Form</h1>
    <form action="/myMessage" method="post">
         <p>Id: <input type="text" id="id" /></p>
        <p>Message: <input type="text" id="content" /></p>
        <p><input type="submit" value="Submit" /></p>
    </form>
</body>
</html>
```

When submit button is clicked it will route to check to see any pre-define @RequestMapping which match with it‘s action attribute (i.e. /myMessage). We can define the @RequestMapping as below:

```
@RequestMapping(value="/myMessage", method=RequestMethod.POST)
public void handleFormSubmission(HttpServletRequest request) {
     String id = request.getParameter("id");
     String content = request.getParameter("content");
     System.out.println(“Message ID:” + id + “Message Content:” + content);
}
```

The HttpServletRequest provides methods for accessing parameters of a request. The type of the request determines where the parameters come from. In most implementations, a GET request takes the parameters from the query string, while a POST request takes the parameters from the posted arguments (form submission).

The methods getParameter(), getParameterValues(), and getParameterNames() are offered as ways to access these arguments. For example, in a GET request with a query string of info=intro the call getParameter("info") returns intro.

By using JSP Form

Let’s create the form view reservationForm.jsp. The form relies on Spring’s form tag library, as this simplifies a form’s data binding, display of error messages and the re-display of original values entered by the user in case of errors.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
<title>Reservation Form</title>
<style>
.error {
 color: #ff0000;
 font-weight: bold;
}
</style>
</head>
    <title>Handing Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <form:form method="post" modelAttribute="reservation">
    <form:errors path="*" cssClass="error" />
    <table>
         <tr>
              <td>Court Name</td>
              <td><form:input path="courtName" /></td>
              <td><form:errors path="courtName" cssClass="error" /></td>
         </tr>
         <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>
    </table>
    </form:form>
</body>
</html>
```

The Spring <form:form> declares two attributes. The method="post" attribute used to indicate a form performs an HTTP POST request upon submission. And the modelAttribute="reservation" attribute used to indicate the form data is bound to a model named reservation. The first attribute should be familiar to you since it's used on most HTML forms. The second attribute will become clearer once we describe the controller that handles the form.

The Spring <form:form> declares two attributes. The method="post" attribute used to indicate a form performs an HTTP POST request upon submission. And the modelAttribute="reservation" attribute used to indicate the form data is bound to a model named reservation. The first attribute should be familiar to you since it's used on most HTML forms. The second attribute will become clearer once we describe the controller that handles the form.

Bear in mind the <form:form> tag is rendered into a standard HTML before it's sent to a user, so it's not that the modelAttribute="reservation" is of use to a browser, the attribute is used as facility to generate the actual HTML form.

Next, you can find the <form:errors> tag, used to define a location in which to place errors in case a form does not meet the rules set forth by a controller. The attribute path="*" is used to indicate the display of all errors—given the wildcard *—where as the attribute cssClass="error" is used to indicate a CSS formatting class to display the errors.

Next, you can find the form's various <form:input> tags accompanied by another set of corresponding <form:errors> tags. These tags make use of the attribute path to indicate the form's fields, which in this case are courtName, date and hour.

The <form:input> tags are bound to properties corresponding to the modelAttribute by using the path attribute. They show the user the original value of the field, which will either be the bound property value or the value rejected due to a binding error. They must be used inside the <form:form> tag, which defines a form that binds to the modelAttribute by its name.

Finally, you can find the standard HTML tag <input type="submit" />that generates a 'Submit' button and trigger the sending of data to the server, followed by the </form:form> tag that closes out the form.

In case the form and its data are processed correctly, you need to create a success view to notify the user of a successful reservation. The reservationSuccess.jsp illustrated next serves this purpose.

```
<!DOCTYPE HTML>
<head>
    <title>Reservation Success</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <h3>Your reservation has been made successfully. </h3>
</body>
</html>
```

Creating a form's service processing
This is not the controller, but rather the service used by the controller to process the form's data reservation. First define a make() method in the ReservationService interface:

```
public interface ReservationService {
public void make(Reservation reservation) throws ReservationNotAvailableException;
}
```

Then you implement this make() method by adding a Reservation item to the list that stores the reservations. You throw a ReservationNotAvailableException in case of a duplicate reservation

```
public class ReservationServiceImpl implements ReservationService {
public void make(Reservation reservation) throws ReservationNotAvailableException {
for (Reservation made : reservations) {
        if(made.getCourtName().equals(reservation.getCourtName())&&
made.getDate().equals(reservation.getDate()) && made.getHour() == reservation.getHour()) { throw n
ew ReservationNotAvailableException(reservation.getCourtName(), reservation.getDate(),
reservation.getHour());
}
}
reservations.add(reservation);
}
 }
```

Creating a form's controller

```
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {
        private 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 "redirect:reservationSuccess";
}
}
```

The controller starts by using the standard @Controller annotation, as well as the @RequestMapping annotation that allows access to the controller through the following URL:
http://localhost:8080/court/reservationForm

When you enter this URL in your browser, it will send an HTTP GET request to your web application. This in turn triggers the execution of the setupForm method, which is designated to attend this type of request based on its @RequestMapping annotation. The setupForm method defines a Model object as an input parameter, which serves to send model data to the view (i.e., the form). Inside the handler method, an empty Reservation object is created that is added as an attribute to the controller's Model object. Then the controller returns the execution flow to the reservationForm view, which in this case is resolved to reservationForm.jsp (i.e., the form).

Validation

Bean validation
Spring framework gives validation at the business logic. This bean validation can be included by adapting the validator interface as support setup. An application can choose to enable bean validation once globally and use it exclusively for all validation needs. An application in spring framework can also register additional spring validator instance per data binder instance which may be useful for plugging in validation logic without the use of annotations.
Normally there are merits and demerits for considering validation at business logic, and spring offers a design form validation and data binding that does not exclude either one of therm. specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available.

Considering the above, Spring has come up with a Validator interface that is both basic and fundamentally usable in every layer of an application.

Data binding is important for allowing users input to be dynamically bounded to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-calledDataBinder to do exactly that. The Validator and the DataBinder make up thevalidation package, which is primarily used in the MVC framework.

Validation using Spring's validator Interface

Spring features a validator interface that you can use to validate objects. The Validator interface works using an Errors object so that while validating, validators can report validation failures to the Errors object.

Here is an example of bean validation

```
public class Person {
    private String name;
    private int age;
    // the usual getters and setters...
}
```

We are going to provide validation for the class Person by implementing the following two methods of the **org.springframework.validation.Validator** interface: these are

- `supports(Class)` validates instance of the supplied class.
- `validate(Object, org.springframework.validation.Errors)` validates the given object incase of validation errors, registers those with the given object. The implementation can be done as follows

```
public class PersonValidator implements Validator {
    /**
     * This Validator validates *just* Person instances     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }
    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

As you it can be see, the static *rejectIfEmpty(..)* method on the *ValidationUtils* class is used to reject the 'name' property if it is null or the empty string.

In spring framework it is possible to implement a single validator class to validate each of the nested objects in a reach object, but it may be better to encapsulate the validation logic for each nested class objects in its own validator implementation.

Annotation type of bean validation can also be used to validate a form. The following code snippet shows how to validate forms using bean validation in annotation way. Annotations are used to put the validation requirements. For example

```
@Size        to limit the number of characters to be used
@NotNull     preventing the entry from being null
@Min         to define the minimum character length
```

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
public class Person {
 @Size(min=2, max=30)
 private String name;
 @NotNull
 @Min(18)
 private Integer age;
………………… some getter and setter methods ……….
 public String checkPersonInfo(@Valid Person person, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "form";
         }
 }
```

`@Valid` … to return valid bean **configuring a Bean Validation Provider**
Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303/JSR-349 Bean Validation provider as a Spring bean. This allows for a `javax.validation.ValidatorFactory` or `javax.validation.Validator` to be injected wherever validation is needed in your application.

Use the `LocalValidatorFactoryBean` to configure a default Validator as a Spring bean:

```
<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

The basic configuration above will trigger Bean Validation to initialize using its default bootstrap mechanism.
Injecting a Validator
It is sometimes interesting to inject the validator into a class using reference. Here is how it can be done :-

`LocalValidatorFactoryBean` implements both `javax.validation.ValidatorFactory` and `javax.validation.Validator`, as well as Spring's `org.springframework.validation.Validator`. You may inject a reference to either of these interfaces into beans that need to invoke validation logic.

Inject a reference to `javax.validation.Validator` if you prefer to work with the Bean Validation API directly:

```
import javax.validation.Validator;
@Service
public class MyService {
   @Autowired
   private Validator validator;
```

Inject a reference to `org.springframework.validation.Validator` if our bean requires the Spring Validation API:

```
import org.springframework.validation.Validator;
@Service
public class MyService {
   @Autowired
   private Validator validator;
}
```

Type conversion

Spring 3 introduces a core.convert package that provides a general type conversion system. The system defines an SPI to implement type conversion logic, as well as an API to execute type conversions at runtime. Within a Spring container, this system can be used as an alternative to PropertyEditors to convert externalized bean property value strings to required property types. The public API may also be used anywhere in your application where type conversion is needed.

Converter SPI

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> {
    T convert(S source);
}
```

To create your own converter, simply implement the interface above. Parameterize **S** as the type you are converting from, and **T** as the type you are converting to.

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {
    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

ConverterFactory

When converting from String to java.lang.Enum objects, implement **ConverterFactory**:

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

Parameterize **S** to be the type you are converting from and **R** to be the base type defining the *range* of classes you can convert to. Then implement getConverter(Class<T>), where **T** is a subclass of **R**.

Consider the StringToEnum ConverterFactory as an example:

```
package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);
    }
    private final class StringToEnumConverter<T extends Enum>
                                    implements    Converter<String, T> {
        private Class<T> enumType;
        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }
        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim());
        }
    }
}
```

GenericConverter

GenericConverter supports converting between multiple source and target types.

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
    public Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object source, TypeDescriptor sourceType,
                                      TypeDescriptor    targetType);
}
```

To implement a GenericConverter, have getConvertibleTypes() return the supported source→target type pairs. Then implement convert(Object, TypeDescriptor, TypeDescriptor) to implement your conversion logic. The source TypeDescriptor provides access to the source field holding the value being converted. The target TypeDescriptor provides access to the target field where the converted value will be set.

A good example of a GenericConverter is a converter that converts between a Java Array and a Collection. Such an ArrayToCollectionConverter introspects the field that declares the target Collection type to resolve the Collection's element type.

ConversionService API

The ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind this facade interface:

```
package org.springframework.core.convert;

public interface ConversionService {
    boolean canConvert(Class<?> sourceType, Class<?> targetType);
    <T> T convert(Object source, Class<T> targetType);
    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

To register a default ConversionService with Spring, add the following bean definition with id conversionService:

```
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

A default ConversionService can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converter(s), set the converters property. Property values may implement either of the Converter, ConverterFactory, or GenericConverter interfaces.

```
<bean id="conversionService"
        class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
         <set>
            <bean class="example.MyCustomConverter"/>
        </set>
    </property>
</bean>
```

To work with a ConversionService instance programmatically, simply inject a reference to it like you would for any other bean:

```
@Service
public class MyService {
    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;

    }
    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

Server-side handling of user interface events

Spring MVC provides `@validator` annotation and `@BindingResult` class for server side event handling. We can also implement our own customized server side event handling by implementing the `org.springframework.validation.Validator` interface and set it as validator in the controller class using `@InitBinder` annotation.

Rendering

Spring MVC is an implementation of the Model-View-Controller pattern for the Web service. The view layer in a Web application drives the user experience, and there is no shortage of view technologies to choose from. Spring MVC accommodates this choice with modular support for pluggable view template technologies such as JSP, JSTL, Tiles, Freemarker, velocity. We used JSP on our project, and we will explain jsp page rendering. After a Spring MVC controller has completed its execution, it returns two key pieces of data to Spring-: a model and a view name. Spring uses a configured View Resolver to map the view name onto an actual view page, bind the model to it, and return it as a rendered view to the user through the HTTP response. For simple, direct view resolution, Spring MVC can be configured with a single View Resolver instance which knows where to locate view template files in the dispatcher-servlet.xml (configuration file).

I18N and L10N

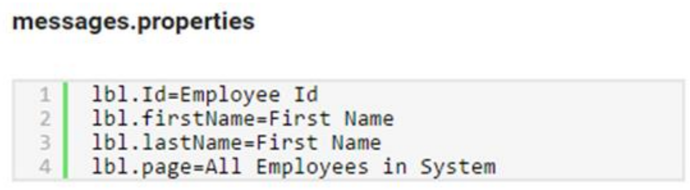
Internationalization is the process of designing software application so that it can be potentially be adapted to various languages religion without engineering changes. Localization is the process of adapting internationalized software for specific region or language by adding locale-specific components and translating text.

Spring framework is shipped with LocaleResolver to support the internationalization and localization as well.

To support internalization and localization in spring framework

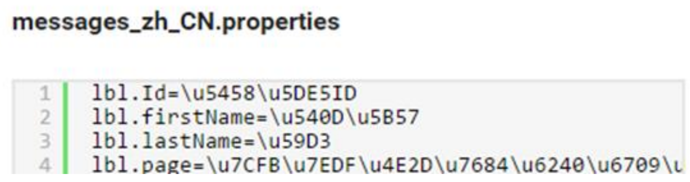
1. Adding locale specific message resources

To support multiple locales we must add as many locale specific resources as we number of locales we need to have. While we are creating locale we must append locale specific short-code at the end.
Example:



messages.properties

```
lbl.Id=Employee Id
lbl.firstName=First Name
lbl.lastName=First Name
lbl.page=All Employees in System
```



messages_zh_CN.properties

```
lbl.Id=\u5458\u5DE5ID
lbl.firstName=\u540D\u5B57
lbl.lastName=\u59D3
lbl.page=\u7CFB\u7EDF\u4E2D\u7684\u6240\u6709\u
```

2. Adding LocaleReSolver configuration in spring context

To support internationalization we need to register two beans

a. Session LocaleResolver : these is used to resolve locales by inspecting the predefined attributes in a user's session. If the session attribute doesn't exist , the locale resolver determines the default locale from accept-language HTTP header.

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

b. Locale Change Interceptor: interceptors are used to detect if special parameter exists in the current HTTP request. The parameters name can be customized with the paramName property of this interceptor. These file is included in application context folder.

```
<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="lang" />
</bean>

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor" />
        </list>
    </property>
</bean>
```

Finally we make view changes to support displaying locale specific text messages.

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

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<html>
<head>
    <title>Spring MVC Hello World</title>
</head>
```

By implementing such small foot print configuration and coding internalization and localization can be fully supported in Spring MVC framework.

JavaScript integration

Spring MVC supports the full integration of JavaScript and JavaScript libraries to enhance user interaction with the web application . The rendering of the JavaScript code into the jsp page can be handled by creating resource sub folder under WEB-INF folder and linking the correct path to the jsp page.

Documentation

Kind of documentation, quality, up-to date

Spring framework is full of documentations that contains guides, Projects, and questions. Each guides, projects has its own explanatory part that describe the area of concern in a well-defined manner. The documentation has also start up guides as well as tutorial for continues long term application learning.
It has a lot of supporting documentations and implementation libraries which are easy to understand and implement. There are also a lot of examples that give the first direction towards development.

Community activity

New blog posts, question and answers are being posted every time. High involvement of experienced and professional programmer as well as learners are increasingly participating in the community which indicates the honor and reputation of the framework.

Future evolution

Amazingly in spring frame work there are continues update, expansions of the supporting library for the framework. There are new releases almost within two days. Sometimes there are two releases are being added to the documentation per single day.
More information about release can be found here: http://spring.io/blog/category/releases

Your perception

Ease of use

Difficult! Since the documentation is too large and there are a lot different type of configuration it needs a lot of effort to be familiar with. Additionally due to the availability of different configuration and libraries styles it is difficult standardize, it is not easy to understand some ones work even if we are able to develop application.

Ease of installation

IDE dependent! Spring Framework is already integrated in the net beans IDE. To install in to Eclipse IDE still needs a lot of effort than it should be. Moving project from one IDE to other IDE is also another challennge. Importing project developed using net beans to Eclipse or form Eclipse to Net beans is not easy. From Intelij version 13 onwards the Intelij IDE developer has included the spring module in to the IDE. These minimizes the effort required to install and integrate the framework with the IDE. Still configuration and sharing project with different Ide such as between intelij and netbeans , or between intelij and eclipse ide is still an issue.

Learning curve

Steep, but can be learnt within a week or less depending on java and other mvc framework background.

Practical information

Website

Interactive and up to date.

Download

Spring for Eclipse can be downloaded from the following link and configured in web.xml file. The Net beans have already integrated the framework as a build in option.
https://www.genuitec.com/products/myeclipse/download/