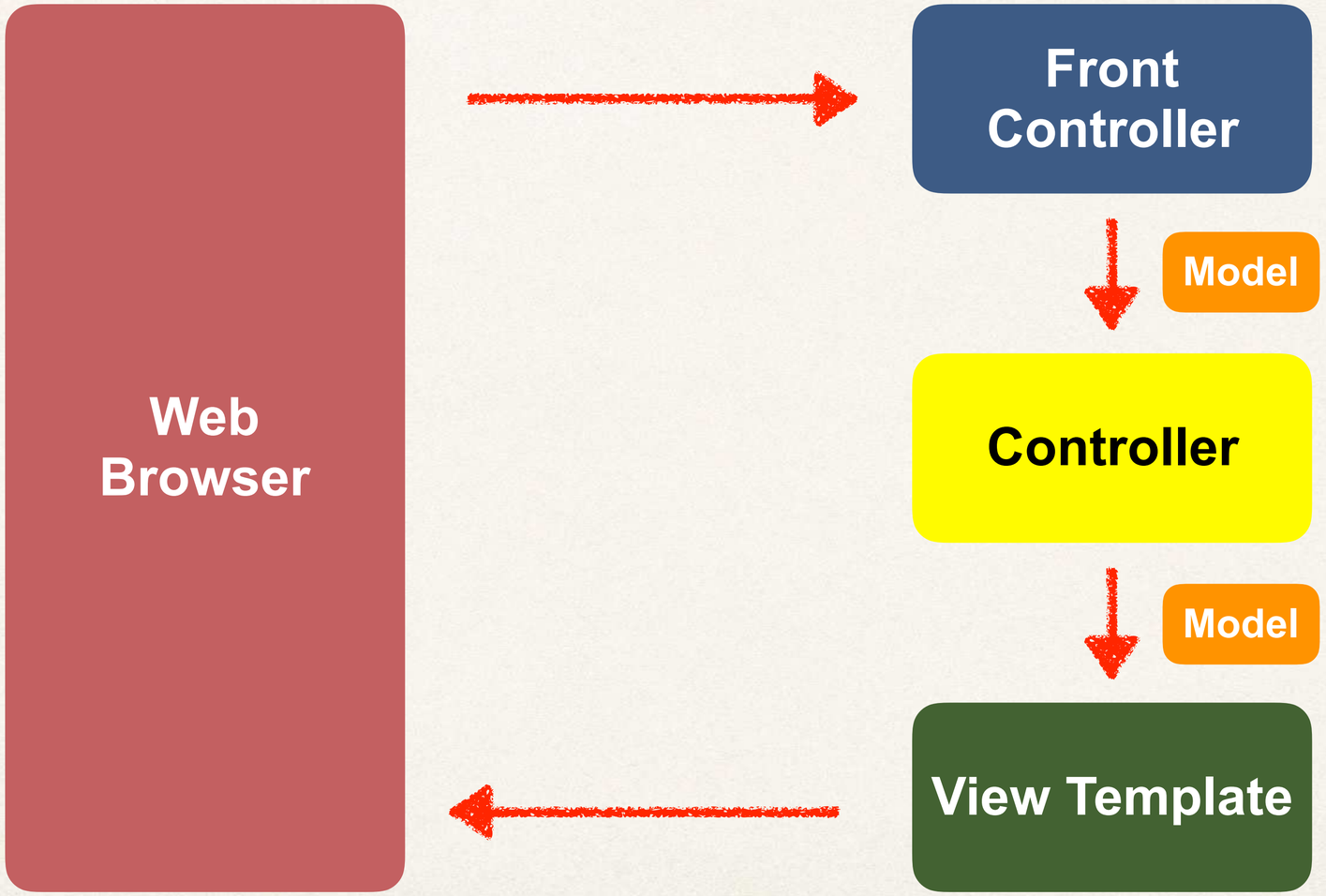
SPRING MVC OVERVIEW

What is Spring MVC?

- Framework for building web applications in Java
- Based on Model-View-Controller design pattern
- Leverages features of the Core Spring Framework (IoC, DI)

Model-View-Controller (MVC)



Spring MVC Benefits

- The Spring way of building web app UIs in Java
- Leverage a set of reusable UI components
- Help manage application state for web requests
- Process form data: validation, conversion etc
- Flexible configuration for the view layer

Spring MVC Documentation

Books

- Plenty of Spring MVC books on Amazon

- Search for “Spring MVC”

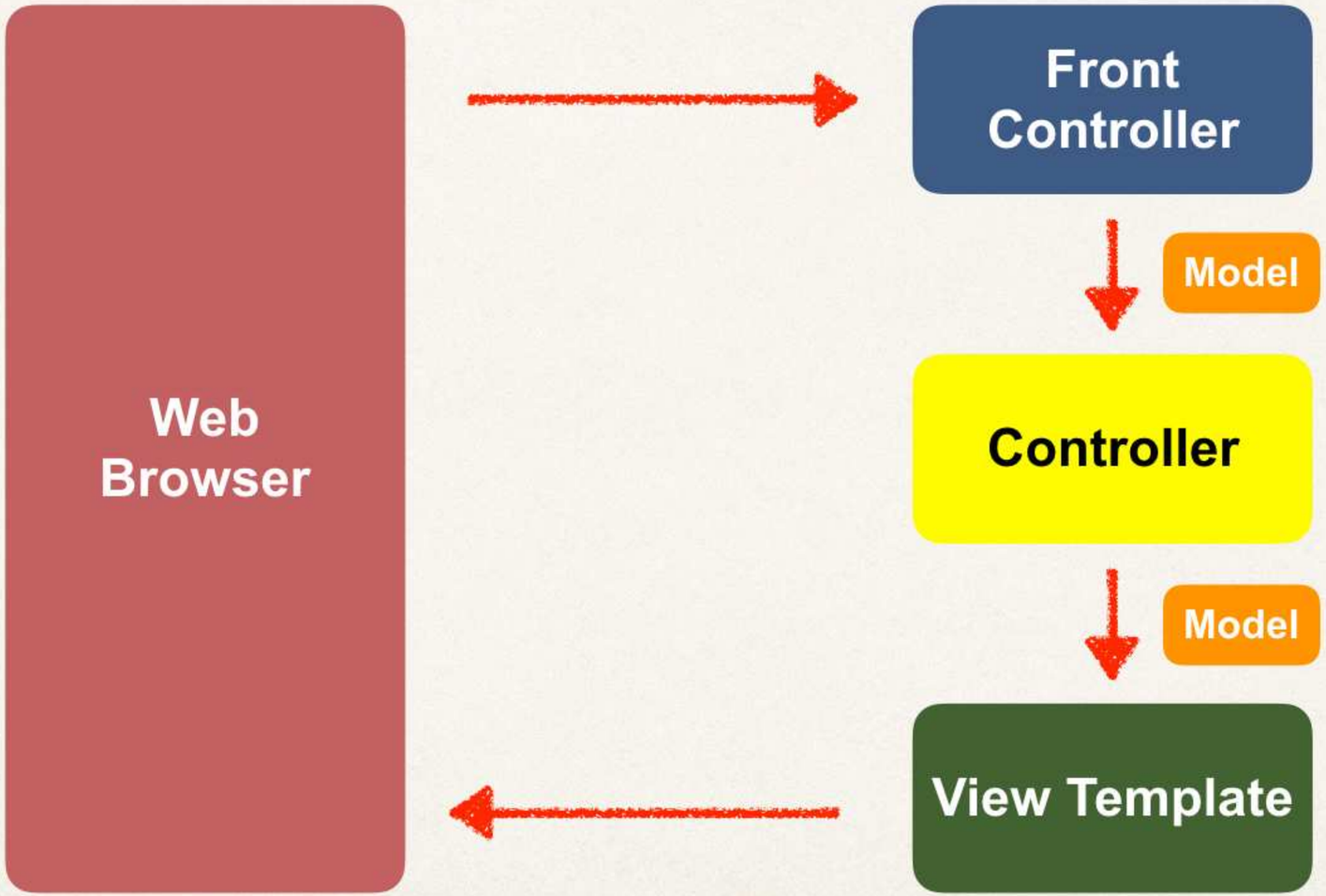
SPRING MVC OVERVIEW

Spring MVC
Behind the Scenes

Components of a Spring MVC Application

- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc…)
- Spring configuration (XML, Annotations or Java)

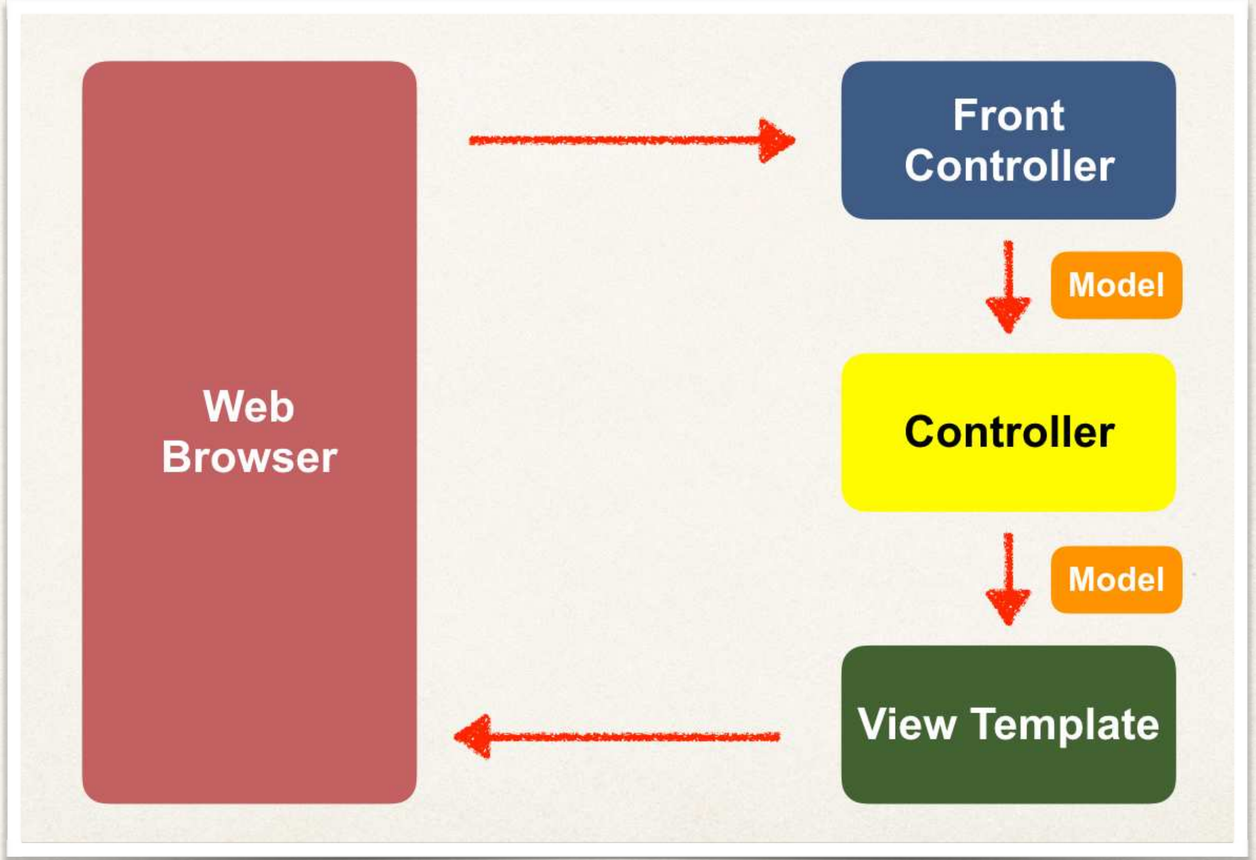
How Spring MVC Works Behind the Scenes



Spring MVC Front Controller

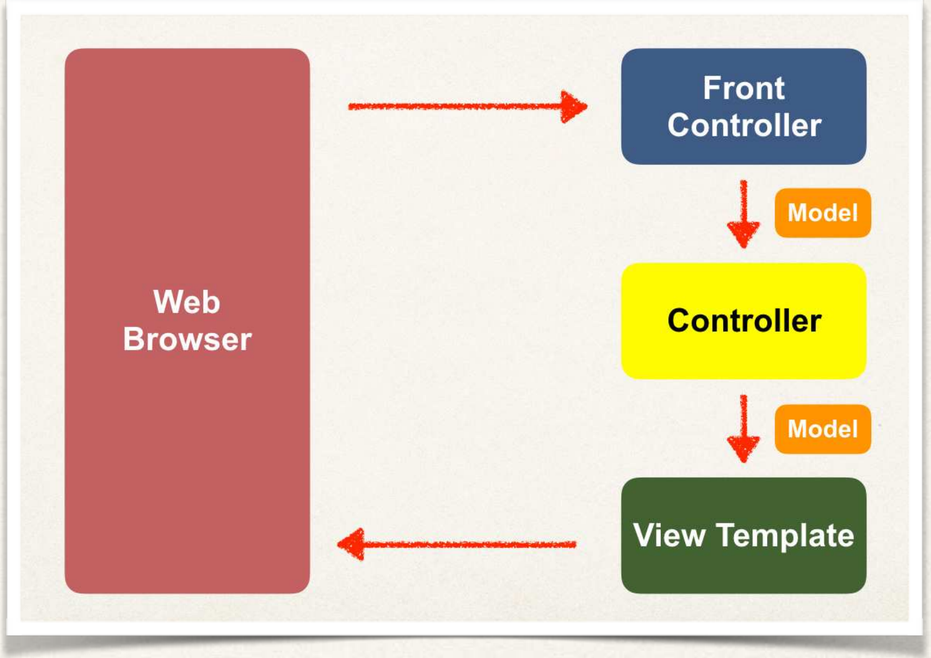
- Front controller known as **DispatcherServlet**
  - Part of the Spring Framework
  - Already developed by Spring Dev Team

- You will create
  - Model objects (orange)
  - View templates (dark green)
  - Controller classes (yellow)



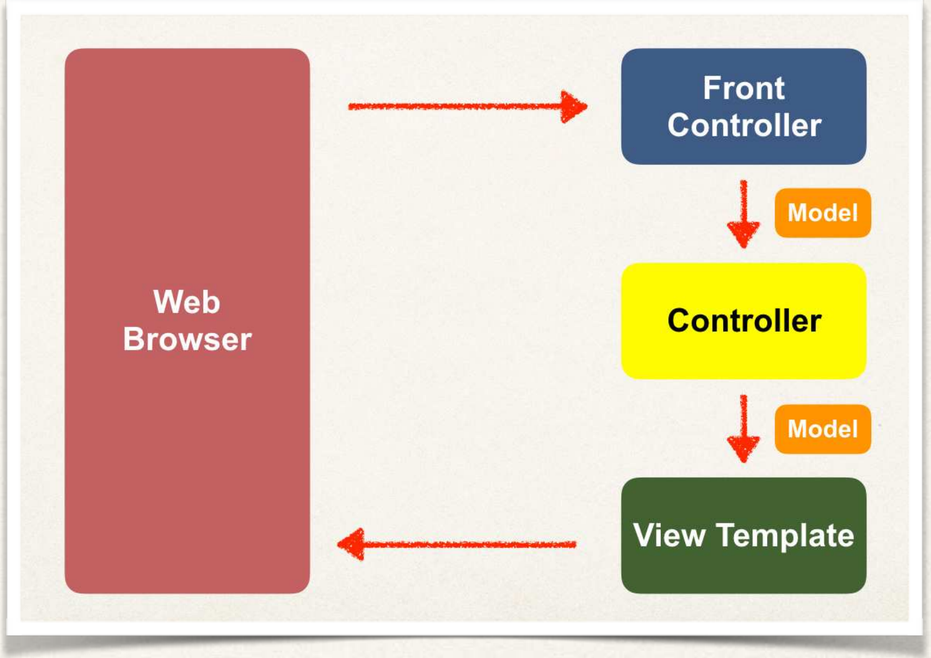
Controller

- Code created by developer
- Contains your business logic
  - Handle the request
  - Store/retrieve data (db, web service…)
  - Place data in model
- Send to appropriate view template



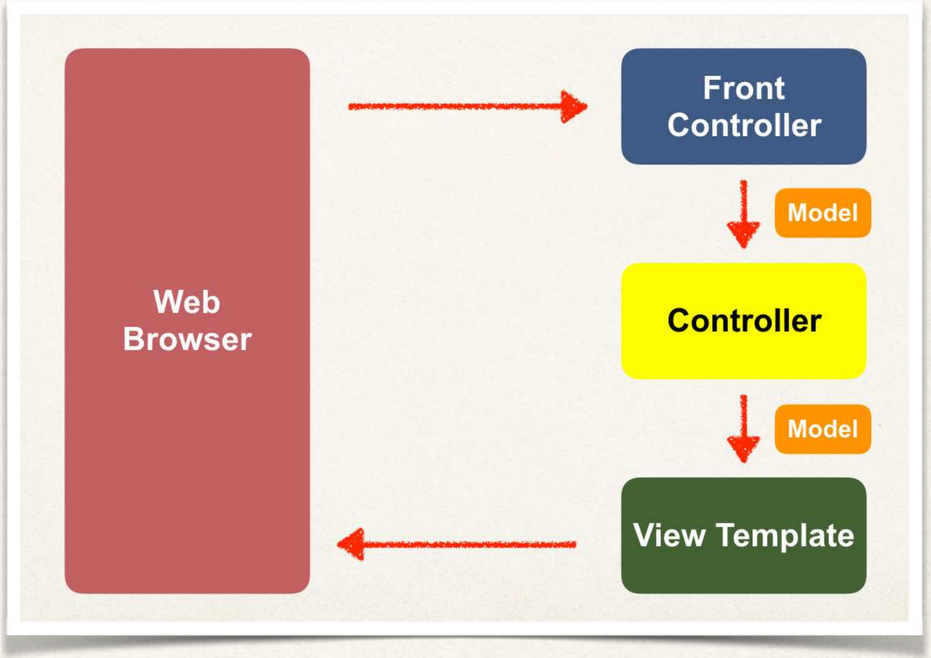
Model

- Model: contains your data
- Store/retrieve data via backend systems
  - database, web service, etc…
  - Use a Spring bean if you like
- Place your data in the model
  - Data can be any Java object/collection



View Template

- Spring MVC is flexible
  - Supports many view templates
- Most common is **JSP** + **JSTL**
- Developer creates a page
  - Displays data



JSP: Java Server Pages
JSTL: JSP Standard Tag Library

View Template (more)

- Other view templates supported
  - Thymeleaf, Groovy
  - Velocity, Freemarker, etc…

- For details, see:

Spring MVC
Behind the Scenes

Dev Environment Setup

Dev Environment Check Point

At this point in the video series you should have installed:

- ☑ Apache Tomcat
- ☑ Eclipse (Java EE version)
- ☑ Connected Eclipse to Tomcat

Tomcat Version

- New version of Tomcat 10 was released to support Jakarta EE 9
  - Renamed packages: `javax.*` to `jakarta.*`
  - This is a breaking change for Java EE apps
- Spring 5 currently does not support the new package renaming Jakarta EE 9
- As a result, Spring 5 does not currently work on Tomcat 10
- Use **Tomcat 9** for your Spring 5 apps

Spring Team is aware of this issue. You can track the status

https://github.com/spring-projects/spring-framework/issues/25354

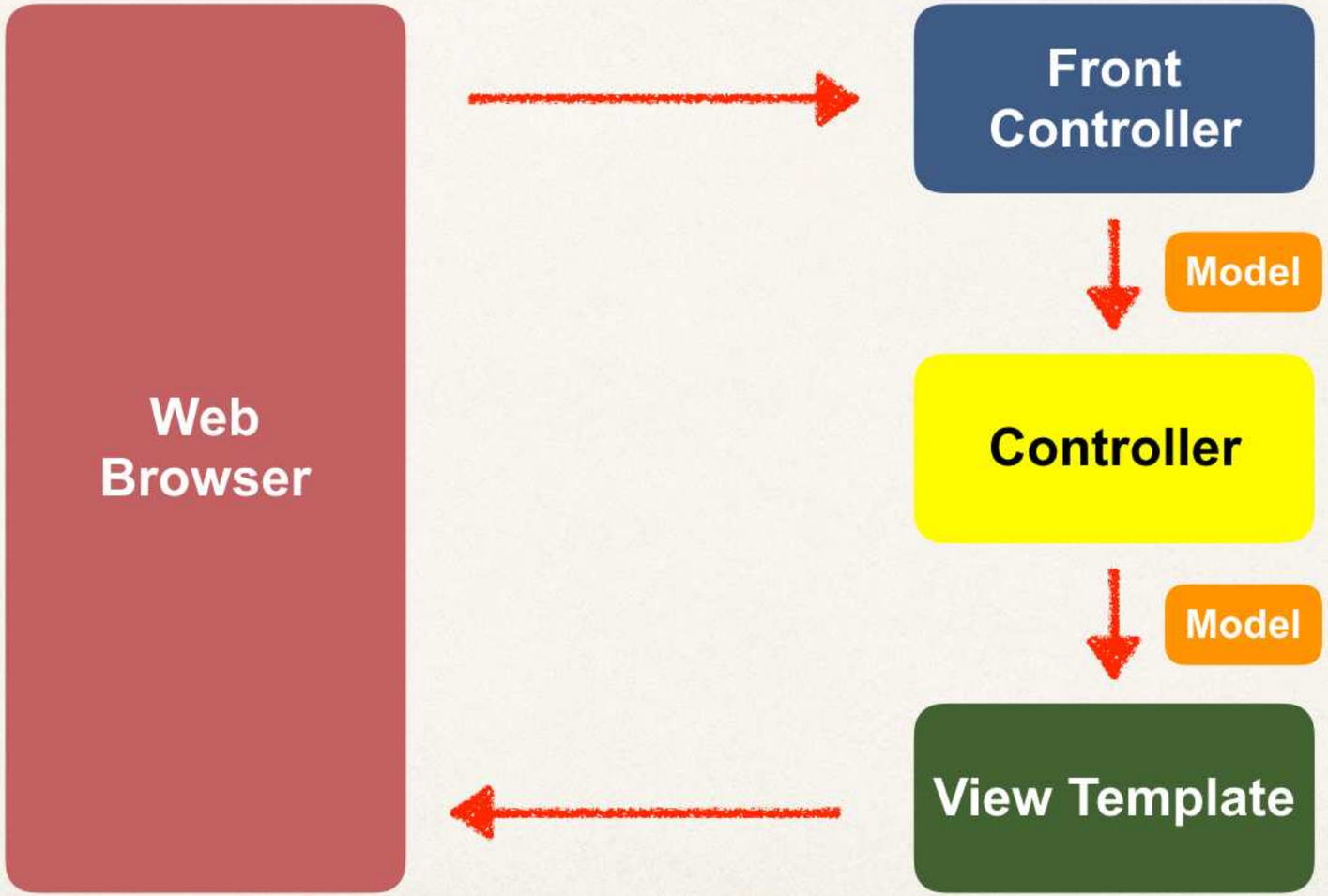
Additional Things To Do

1. Download Spring MVC source code for this training class

2. Download latest Spring JAR files from Spring website

SPRING MVC CONFIGURATION

Model-View-Controller (MVC)



Spring MVC Configuration Process - Part 1

Step-By-Step

Add configurations to file: **WEB-INF/web.xml**

1. Configure Spring MVC Dispatcher Servlet

2. Set up URL mappings to Spring MVC Dispatcher Servlet

Spring MVC Configuration Process - Part 2

Add configurations to file: **WEB-INF/spring-mvc-demo-servlet.xml**

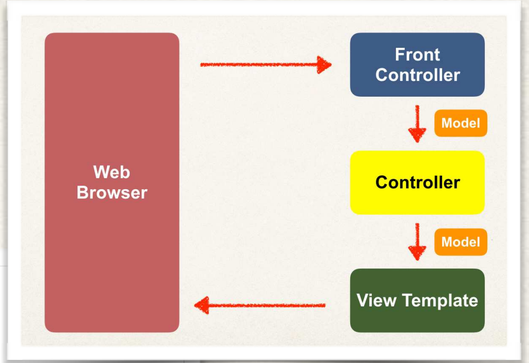
Step-By-Step

3. Add support for Spring component scanning

4. Add support for conversion, formatting and validation

5. Configure Spring MVC View Resolver

Step 1: Configure Spring DispatcherServlet



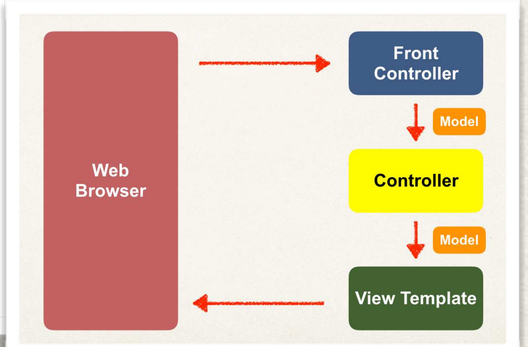
File: web.xml

```
<web-app>

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

    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
    </init-param>

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

</web-app>
```

Step 2: Set up URL mappings to Spring MVC Dispatcher Servlet

File: web.xml

```
<web-app>

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

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```



Step 3: Add support for Spring component scanning

File: spring-mvc-demo-servlet.xml

```
<beans>

    <!-- Step 3: Add support for component scanning -->
    <context:component-scan base-package="com.luv2code.springdemo" />

</beans>
```

Step 4: Add support for conversion, formatting and validation

File: spring-mvc-demo-servlet.xml

```
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

  <!-- Step 4: Add support for conversion, formatting and validation support -->
  <mvc:annotation-driven/>

</beans>
```

Step 5: Configure Spring MVC View Resolver

File: spring-mvc-demo-servlet.xml

```
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

  <!-- Step 4: Add support for conversion, formatting and validation support -->
  <mvc:annotation-driven/>

  <!-- Step 5: Define Spring MVC view resolver -->
  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
  </bean>

</beans>
```

View Resolver Configs - Explained

- When your app provides a “view” name, Spring MVC will
  - prepend the prefix
  - append the suffix

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

View Resolver Configs - Explained

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

/WEB-INF/view/ show-student-list .jsp

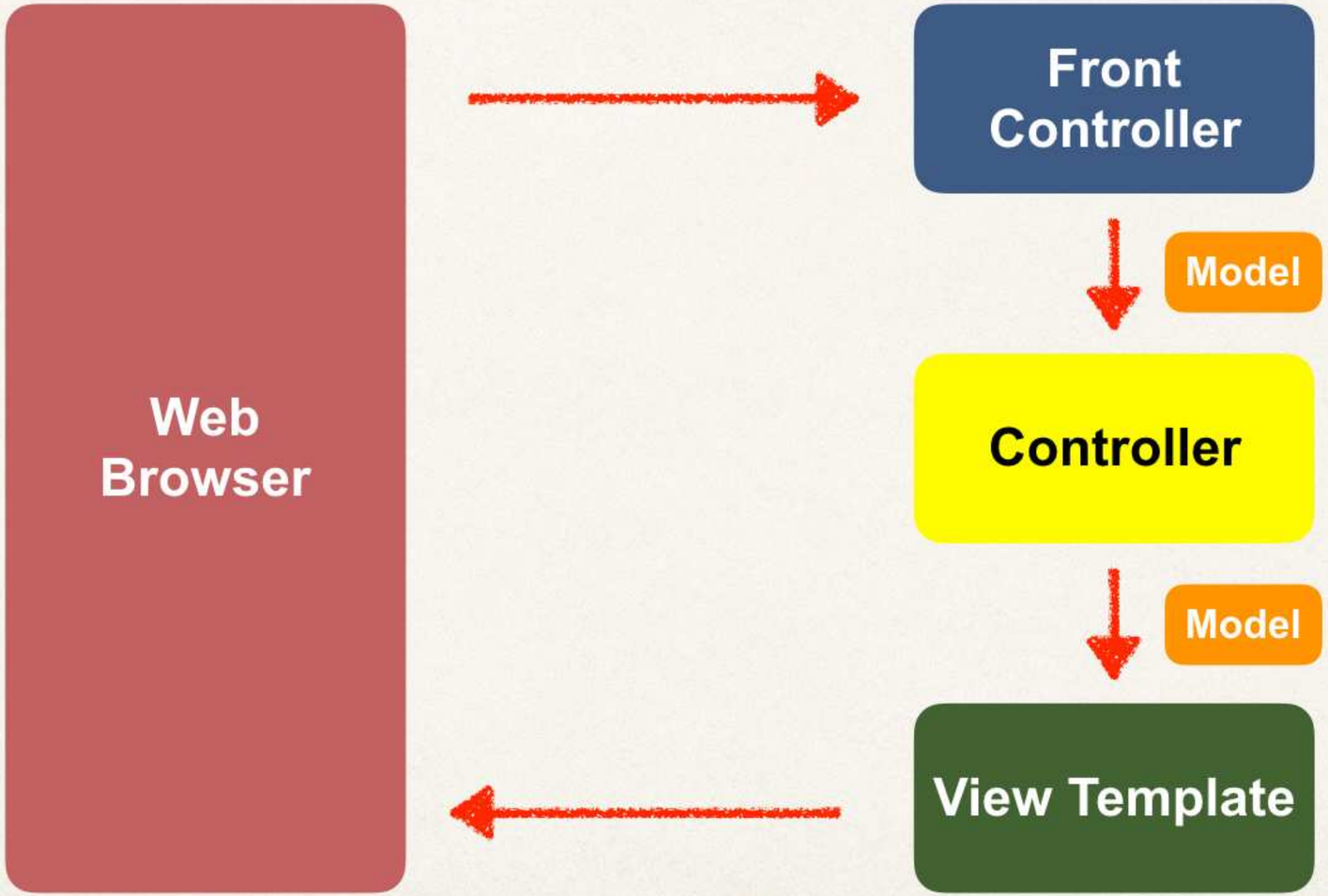
View name

SPRING MVC CONFIGURATION

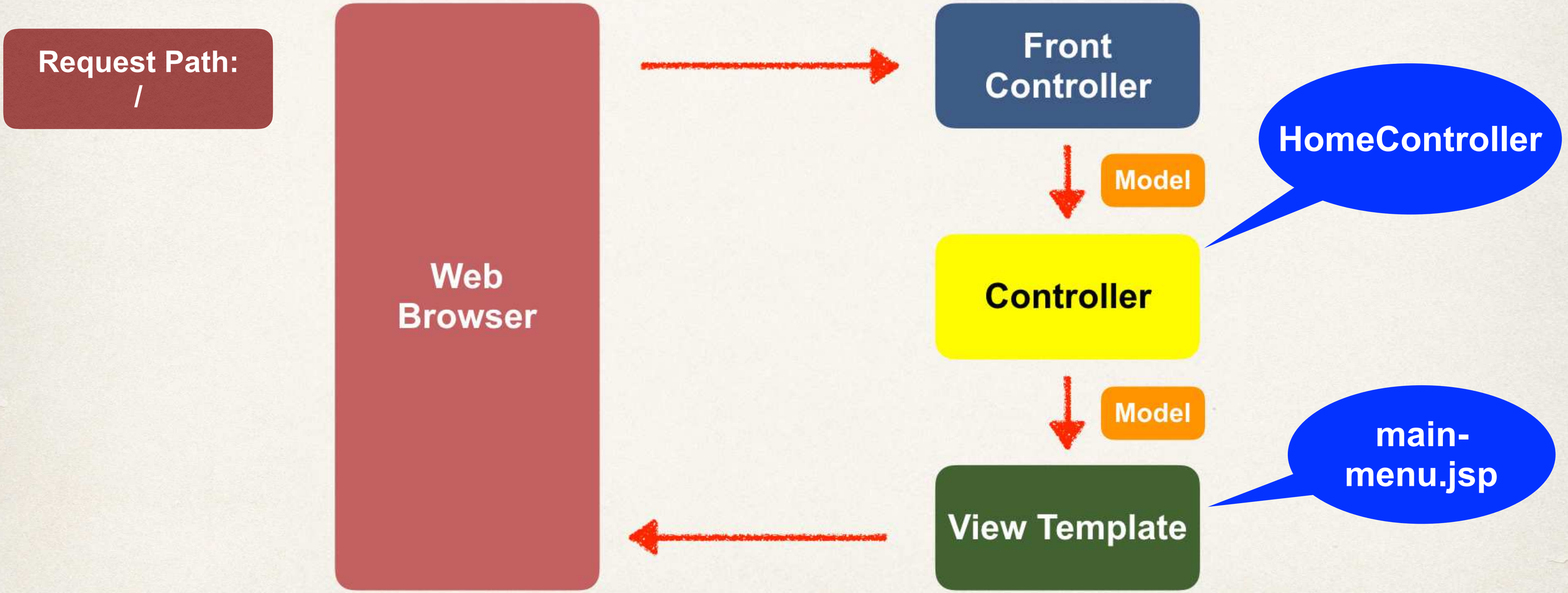
luv2code

Developing
Spring Controllers and Views

Model-View-Controller (MVC)



Our First Spring MVC Example



Development Process

Step-By-Step

1. Create Controller class
2. Define Controller method
3. Add Request Mapping to Controller method
4. Return View Name
5. Develop View Page

Step 1: Create Controller class

- Annotate class with **@Controller**
  - **@Controller** inherits from **@Component** … supports scanning

```
@Controller
public class HomeController {

}
```

Step 2: Define Controller method

```
@Controller
public class HomeController {

  public String showMyPage() {
      ...
  }

}
```

Step 3: Add Request Mapping to Controller method

```
@Controller
public class HomeController {

  @RequestMapping("/")
  public String showMyPage() {
      ...
  }

}
```

Step 4: Return View Name

```
@Controller
public class HomeController {

  @RequestMapping("/")
  public String showMyPage() {
      return "main-menu";
  }

}
```

Finding the View Page

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

/WEB-INF/view/main-menu.jsp

View name

Step 5: Develop View Page

File: /WEB-INF/view/main-menu.jsp

```
<html><body>

<h2>Spring MVC Demo - Home Page</h2>

</body></html>
```

Recap - Development Process

Step-By-Step

1. Create Controller class
2. Define Controller method
3. Add Request Mapping to Controller method
4. Return View Name
5. Develop View Page

luv2code

Reading Form Data with Spring MVC

High Level View

helloworld-form.jsp

What's your name? Submit Query

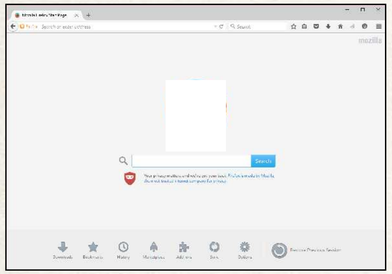
helloworld.jsp

Hello World of Spring!

Student name: John Doe

Application Flow

Application Flow



Request Mapping

/showForm

HelloWorld Controller

helloworld-form.jsp

What's your name? Submit Query

---

helloworld-form.jsp

What's your name? Submit Query

Request Mapping

/processForm

HelloWorld Controller

helloworld.jsp

Hello World of Spring!

Student name: John Doe

Controller Class

```
@Controller
public class HelloWorldController {

   // need a controller method to show the initial HTML form

   @RequestMapping("/showForm")
   public String showForm() {
      return "helloworld-form";
   }

   // need a controller method to process the HTML form

   @RequestMapping("/processForm")
   public String processForm() {
      return "helloworld";
   }
}
```

Development Process

Step-By-Step

1. **Create Controller class**

2. **Show HTML form**

   a. Create controller method to show HTML Form

   b. Create View Page for HTML form

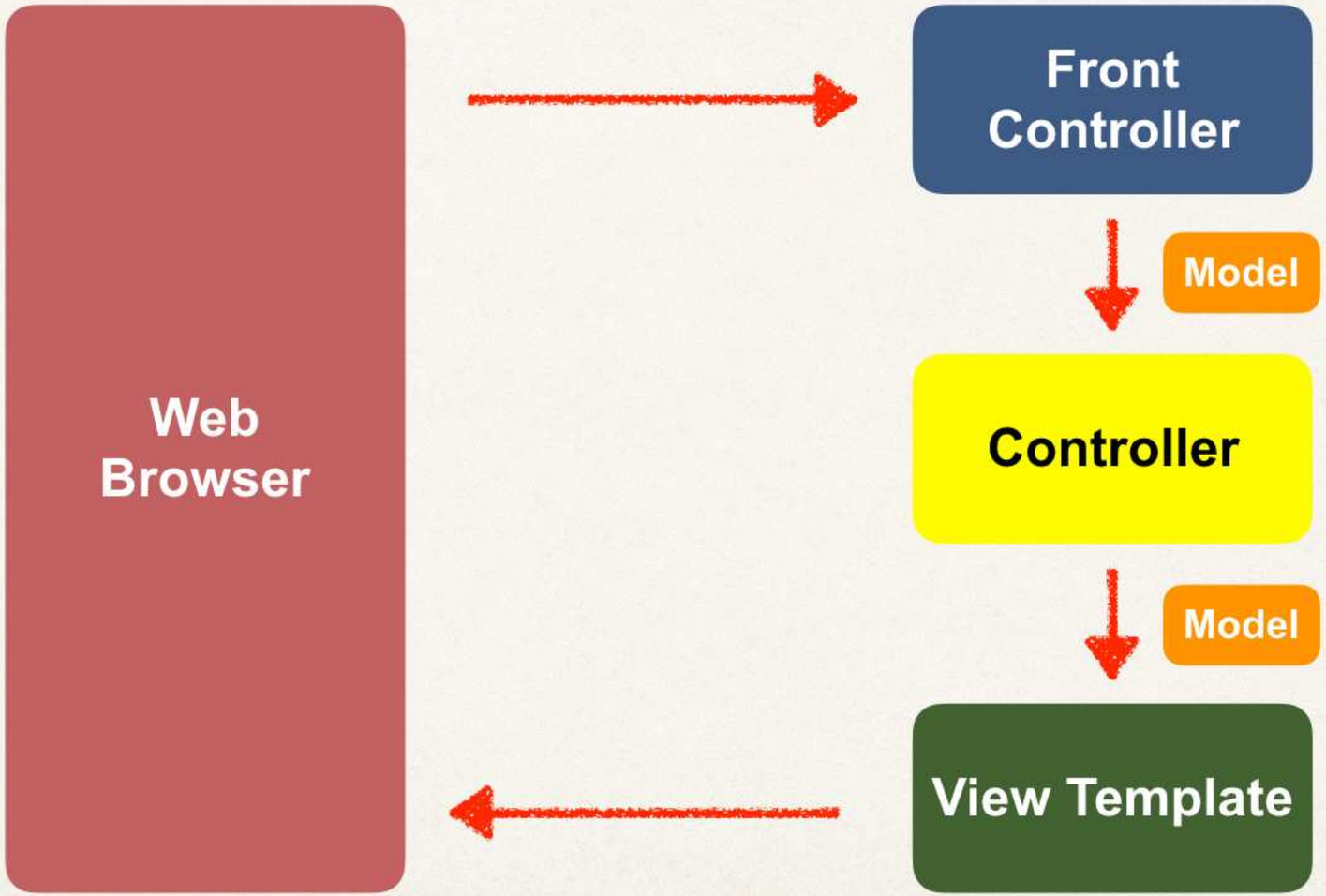
3. **Process HTML Form**

   a. Create controller method to process HTML Form

   b. Develop View Page for Confirmation

Adding Data to Spring Model

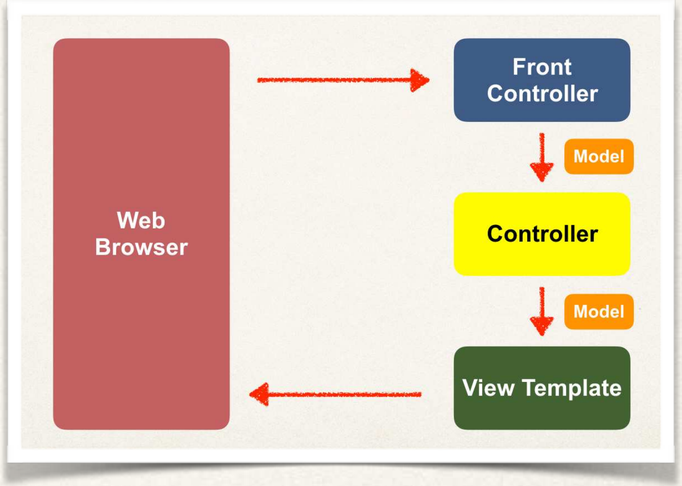
Focus on the Model



Spring Model

- The **Model** is a container for your application data

- In your Controller
  - You can put anything in the **model**
  - strings, objects, info from database, etc…



- Your View page (JSP) can access data from the **model**

Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model

Passing Model to your Controller

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

    // read the request parameter from the HTML form
    String theName = request.getParameter("studentName");

    // convert the data to all caps
    theName = theName.toUpperCase();

    // create the message
    String result = "Yo! " + theName;

    // add message to the model
    model.addAttribute("message", result);

    return "helloworld";
}
```

View Template - JSP

```
<html><body>

Hello World of Spring!
...

The message: ${message}

</body></html>
```

Adding more data to your Model

```
// get the data
//
String result = ...
List<Student> theStudentList = ...
ShoppingCart theShoppingCart = ...

// add data to the model
//
model.addAttribute("message", result);

model.addAttribute("students", theStudentList);

model.addAttribute("shoppingCart", theShoppingCart);
```

Reading HTML Form Data with @RequestParam Annotation

Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model

Instead of using HttpServletRequest

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

    // read the request parameter from the HTML form
    String theName = request.getParameter("studentName");

    ...
}
```

Bind variable using @RequestParam Annotation

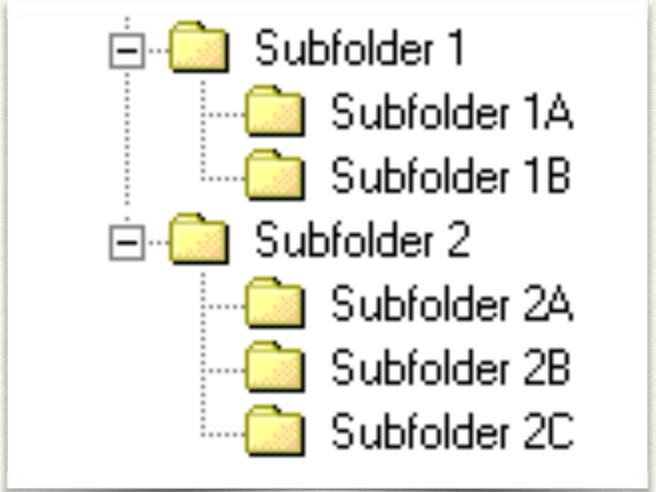
```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
        @RequestParam("studentName") String theName,
        Model model) {

    // now we can use the variable: theName

}
```

- Behind the scenes:
  - Spring will read param from request: studentName
  - Bind it to the variable: theName

Add Controller @RequestMapping

Adding Request Mappings to Controller

- Serves as parent mapping for controller
- All request mappings on methods in the controller are relative
- Similar to folder directory structures



Controller Request Mapping

```
@RequestMapping("/funny")
public class FunnyController {

    @RequestMapping("/showForm")
    public String showForm() {
        ...
    }

    @RequestMapping("/processForm")
    public String process(HttpServletRequest request, Model model) {
        ...
    }

}
```

/funny/showForm

/funny/processForm

Spring MVC Form Tags

Review HTML Forms

- HTML Forms are used to get input from the user

Spring MVC Form Tags

- Spring MVC Form Tags are the building block for a web page
- Form Tags are configurable and reusable for a web page

Data Binding

- Spring MVC Form Tags can make use of data binding
- Automatically setting / retrieving data from a Java object / bean

Spring MVC Form Tags

- Form tags will generate HTML for you :-)

Form Tag	Description
form:form	main form container
form:input	text field
form:textarea	multi-line text field
form:checkbox	check box
form:radiobutton	radio buttons
form:select	drop down list
more ….	

Web Page Structure

- JSP page with special Spring MVC Form tags

```
<html>

  … regular html …

  … Spring MVC form tags …

  … more html …

</html>
```

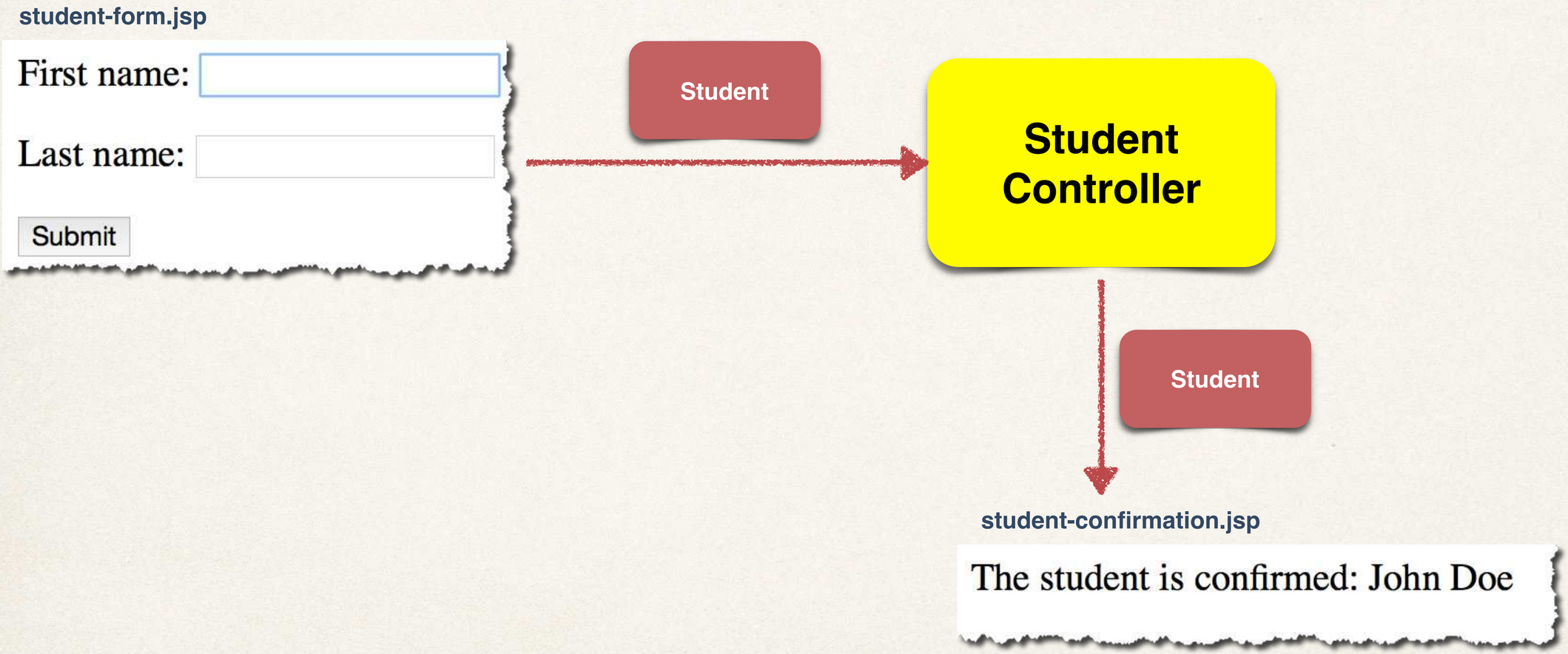
How To Reference Spring MVC Form Tags

- Specify the Spring namespace at beginning of JSP file

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

Spring MVC Form Tag - Text Field

Big Picture


student-form.jsp
First name:
Last name:
Submit
Student
Student Controller
Student
student-confirmation.jsp
The student is confirmed: John Doe

Showing Form

In your Spring Controller

- Before you show the form, you must add a ***model attribute***

- This is a bean that will hold form data for the ***data binding***

Show Form - Add Model Attribute

Code snippet from Controller

```
@RequestMapping("/showForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```

Setting up HTML Form - Data Binding

```
<form:form action="processForm" modelAttribute="student">

  First name: <form:input path="firstName" />

  <br><br>

  Last name: <form:input path="lastName" />

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

When Form is Loaded … fields are populated

```
<form:form action="processForm" modelAttribute="student">

  First name: <form:input path="firstName" />

  <br><br>

  Last name: <form:input path="lastName" />

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

When form is **loaded**,
Spring MVC will call:

student.getFirstName()
…
student.getLastName

When Form is submitted … calls setter methods

```
<form:form action="processForm" modelAttribute="student">

  First name: <form:input path="firstName" />

  <br><br>

  Last name: <form:input path="lastName" />

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

When form is **submitted**, Spring MVC will call:

student.setFirstName(…)
…
student.setLastName(…)

Handling Form Submission in the Controller

Code snippet from Controller

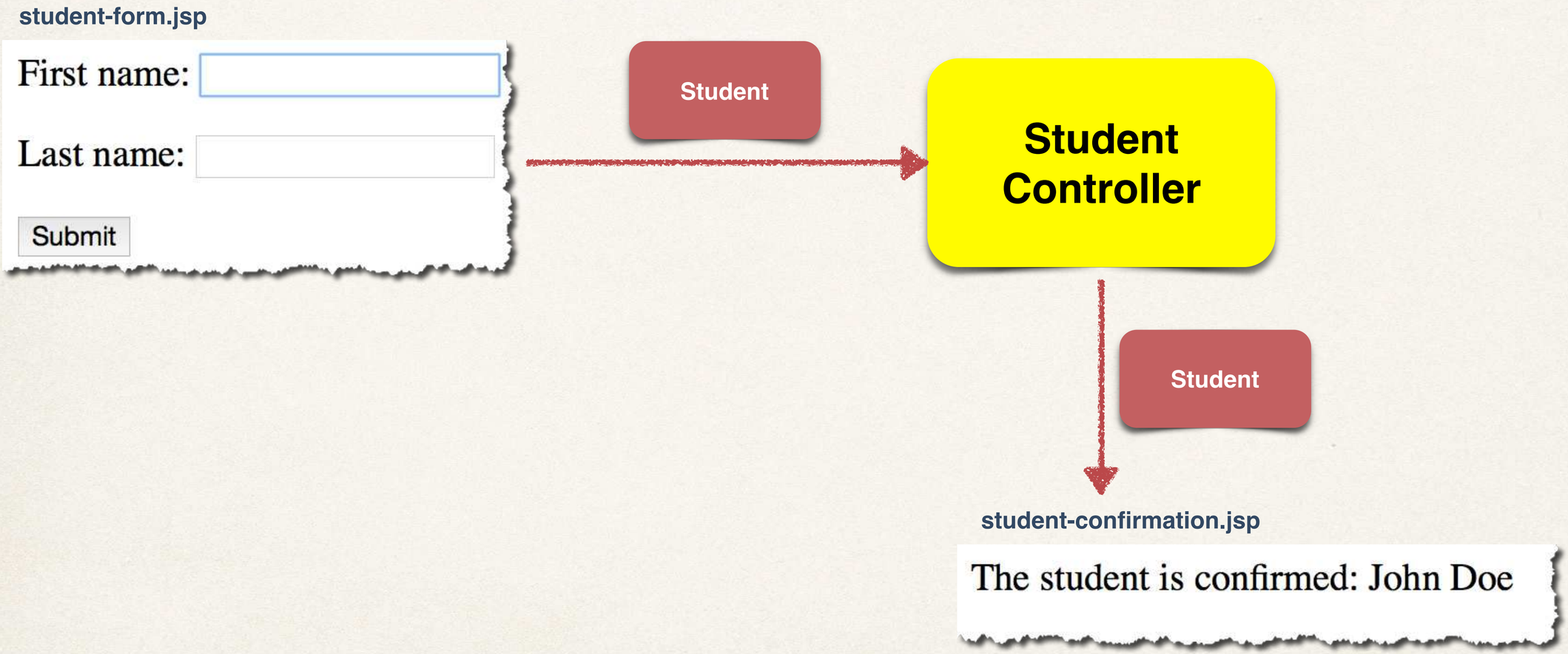
```
@RequestMapping("/processForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

    // log the input data
    System.out.println("theStudent: " + theStudent.getFirstName()
                                + " " + theStudent.getLastName());

    return "student-confirmation";
}
```

Confirmation page

```
<html>

<body>

  The student is confirmed: ${student.firstName} ${student.lastName}

</body>

</html>
```

The student is confirmed: John Doe

Pulling It All Together



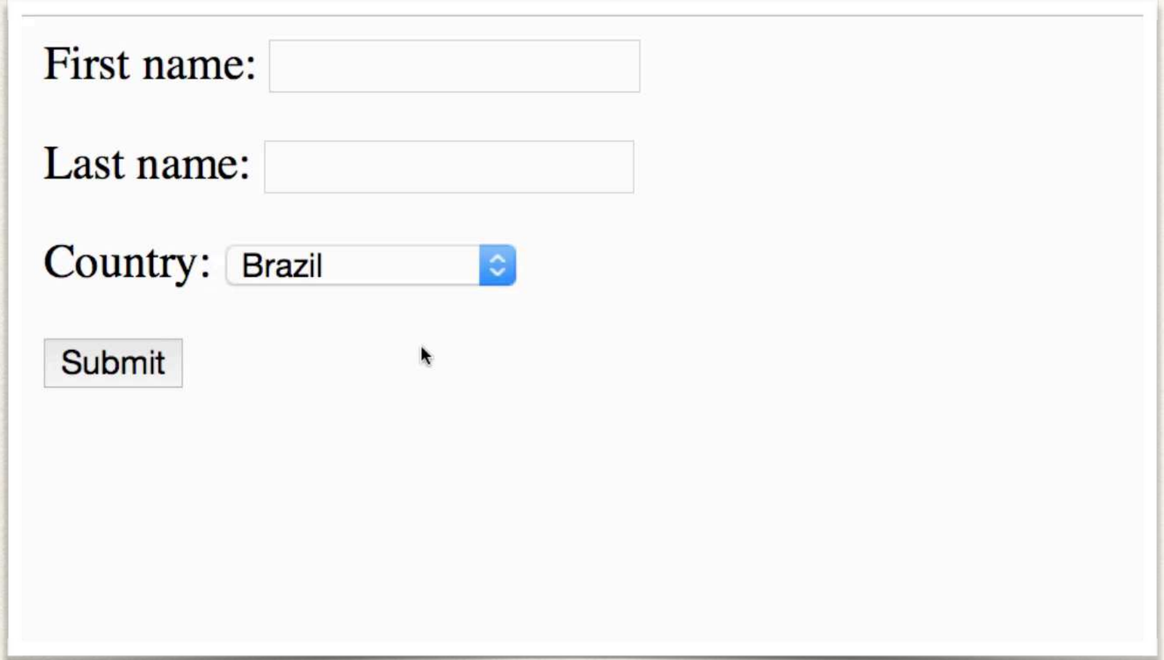
Development Process

Step-By-Step

1. Create Student class
2. Create Student controller class
3. Create HTML form
4. Create form processing code
5. Create confirmation page

Spring MVC Form Tag - Drop Down List

Review of HTML <select> Tag




```
<select name="country">
  <option>Brazil</option>
  <option>France</option>
  <option>Germany</option>
  <option>India</option>
    ...
</select>
```

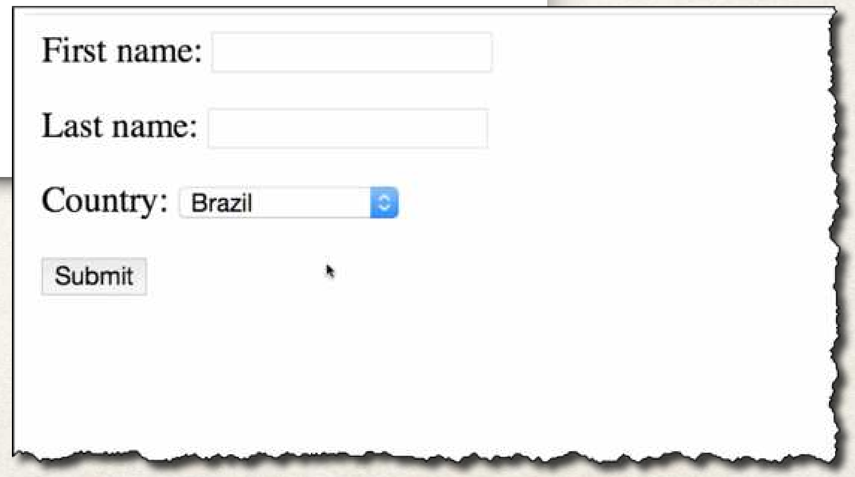
Spring MVC Tag

Drop-Down List is represented by the tag

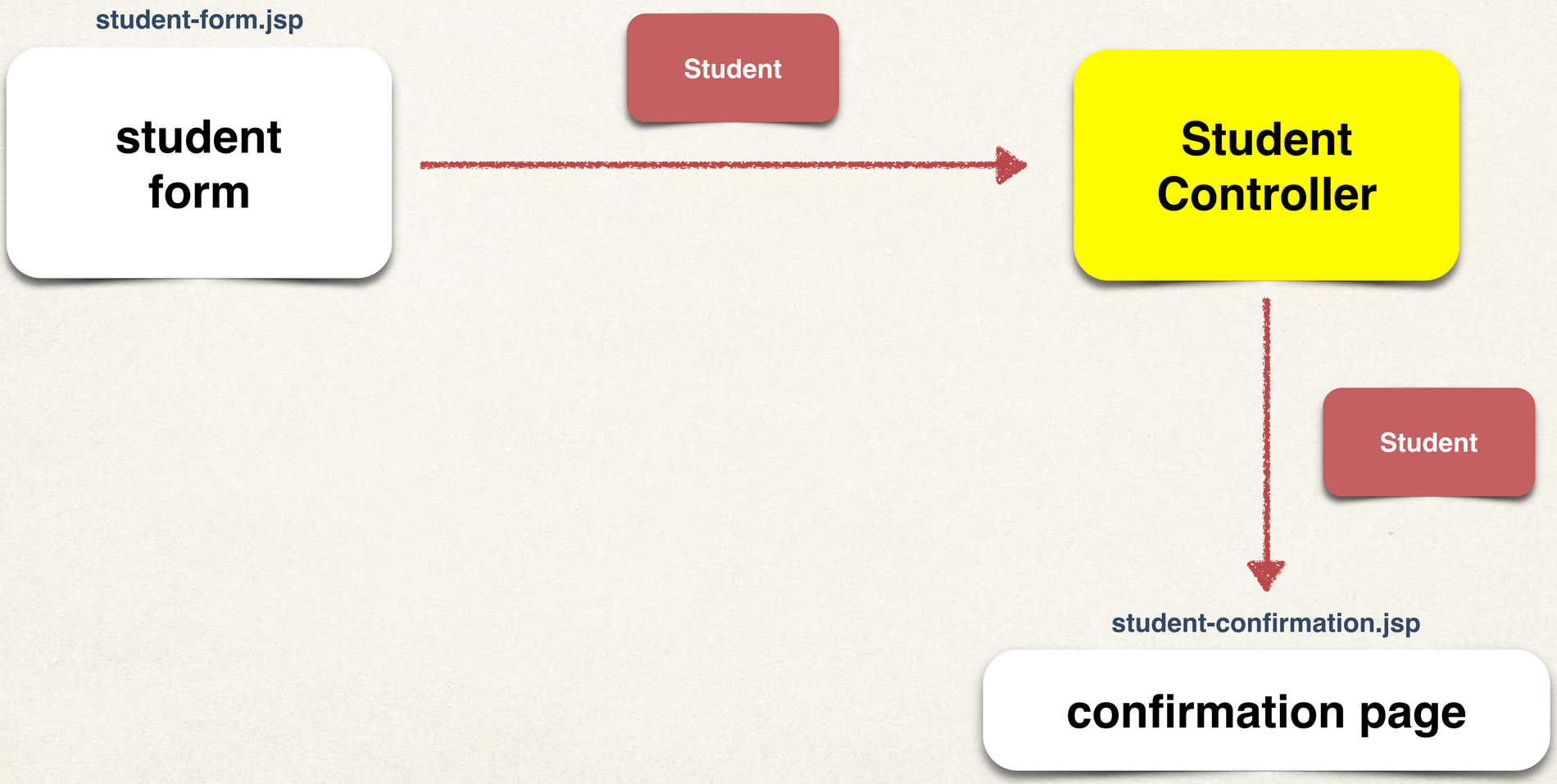
<form:select>

Code Snippet

```
<form:select path="country">

  <form:option value="Brazil" label="Brazil" />
  <form:option value="France" label="France" />
  <form:option value="Germany" label="Germany" />
  <form:option value="India" label="India" />

</form:select>
```



Pulling It All Together



Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

Spring MVC Form Tag - Radio Buttons

Radio Button Demo

First name:

Last name:

Favorite Programming Languages:

Java C# PHP Ruby

Submit

Spring MVC Tag

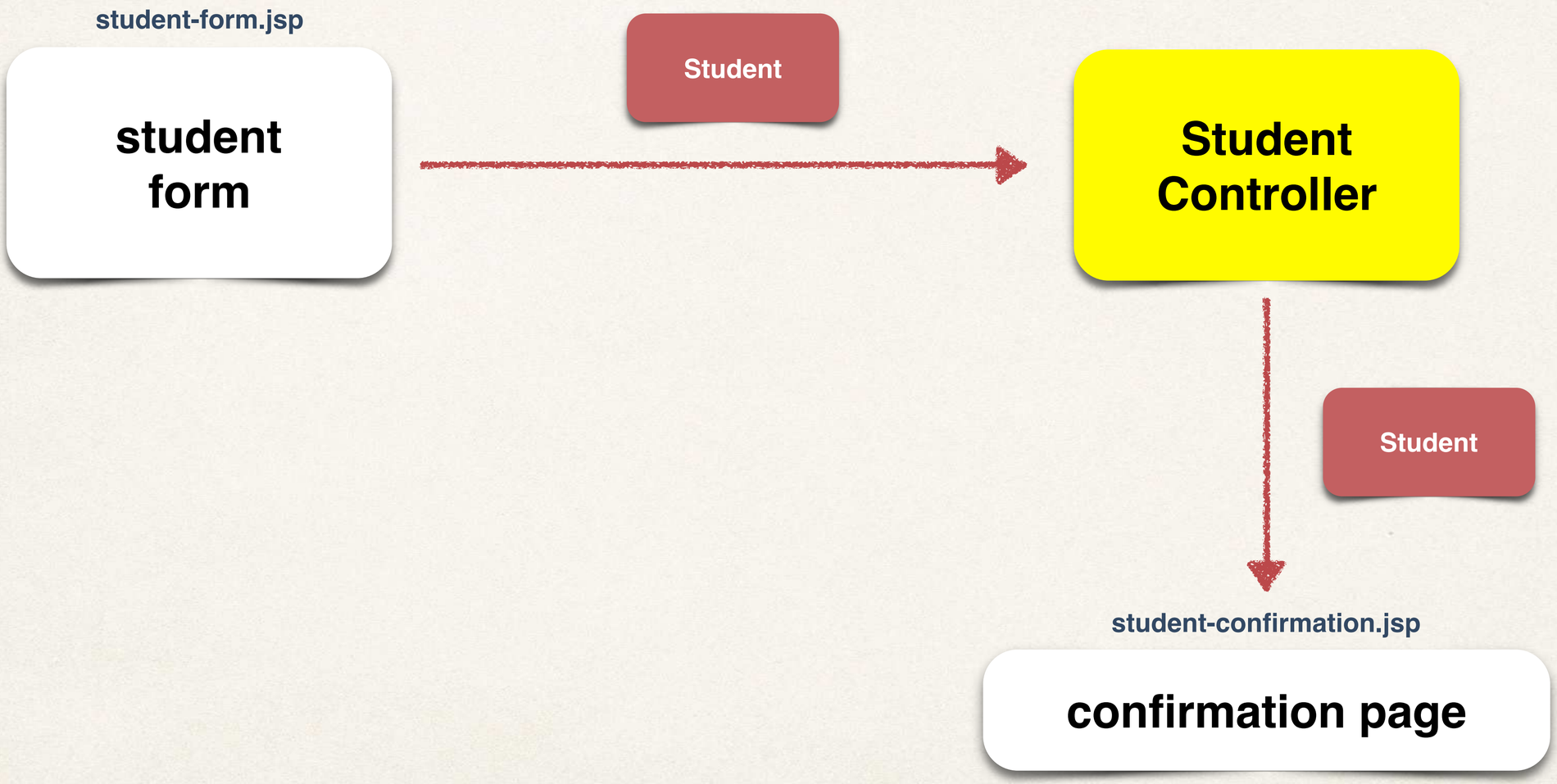
A Radio Button is represented by the tag

<form:radiobutton>

Code Example

```
Java <form:radiobutton path="favoriteLanguage" value="Java" />
C# <form:radiobutton path="favoriteLanguage" value="C#" />
PHP <form:radiobutton path="favoriteLanguage" value="PHP" />
Ruby <form:radiobutton path="favoriteLanguage" value="Ruby" />
```

Pulling It All Together



Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

Spring MVC Form Tag - Check Box

Check Box - Pick Your Favorite OS

Operating Systems: Linux ☐ Mac OS ☐ MS Windows ☐

Submit

Spring MVC Tag

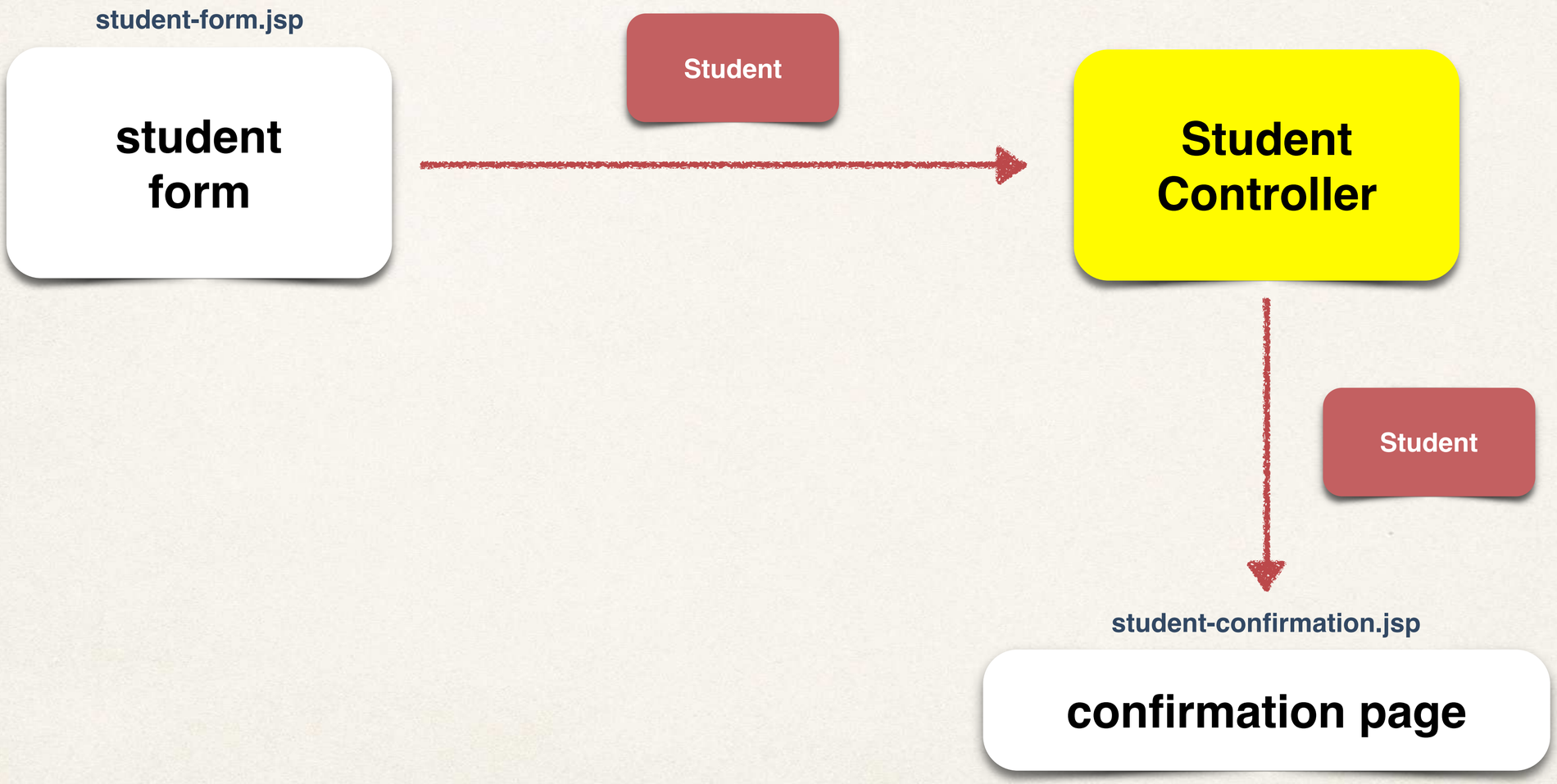
A Check Box is represented by the tag

<form:checkbox>

Code Example

```
Linux <form:checkbox path="operatingSystems" value="Linux" />
Mac OS <form:checkbox path="operatingSystems" value="Mac OS" />
MS Windows <form:checkbox path="operatingSystems"
                          value="MS Windows" />
```

Pulling It All Together



Development Process

Step-By-Step

1. Update HTML form

2. Update Student class - add getter/setter for new property

3. Update confirmation page

luv2code

Spring MVC Form Validation

The Need for Validation

The Need for Validation

Check the user input form for

required fields

valid numbers in a range

valid format (postal code)

custom business rule

Java's Standard Bean Validation API

Java's Standard Bean Validation API

- Java has a standard Bean Validation API

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Not tied to either the web tier or the persistence tier

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Not tied to either the web tier or the persistence tier
- Available for server-side apps and also client-side JavaFX/Swing apps

Java's Standard Bean Validation API

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Not tied to either the web tier or the persistence tier
- Available for server-side apps and also client-side JavaFX/Swing apps

http://www.beanvalidation.org

Spring and Validation

Spring and Validation

- Spring version 4 and higher supports Bean Validation API

Spring and Validation

- Spring version 4 and higher supports Bean Validation API
- Preferred method for validation when building Spring apps

Spring and Validation

- Spring version 4 and higher supports Bean Validation API
- Preferred method for validation when building Spring apps
- Simply add Validation JARs to our project

Bean Validation Features

Validation Feature
required
validate length
validate numbers
validate with regular expressions
custom validation

Validation Annotations

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number >= value
@Max	Must be a number <= value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others …	

Our Road Map

Our Road Map

1. **set up our development environment**

Our Road Map

1. **set up our development environment**
2. **required field**

Our Road Map

1. **set up our development environment**
2. **required field**
3. **validate number range: min, max**

Our Road Map

1. set up our development environment
2. required field
3. validate number range: min, max
4. validate using regular expression (regexp)

Our Road Map

1. set up our development environment
2. required field
3. validate number range: min, max
4. validate using regular expression (regexp)
5. custom validation

Setting Up Development Environment

Java's Standard Bean Validation API

Bean Validation

Constrain once, validate everywhere

Java's Standard Bean Validation API

Bean Validation

Constrain once, validate everywhere

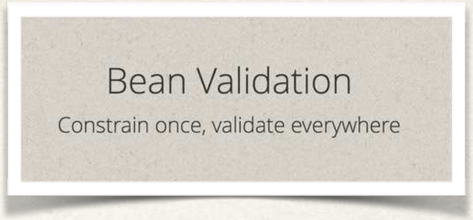
- Java's standard Bean Validation API (JSR-303)

Java's Standard Bean Validation API

- Java's standard Bean Validation API (JSR-303)

- Only a specification … vendor independent … portable

Java's Standard Bean Validation API



- Java's standard Bean Validation API (JSR-303)
- Only a specification … vendor independent … portable
- BUT we still need an implementation …

The Hibernate Team to the rescue!

The Hibernate Team to the rescue!

- Hibernate started as an ORM project

The Hibernate Team to the rescue!

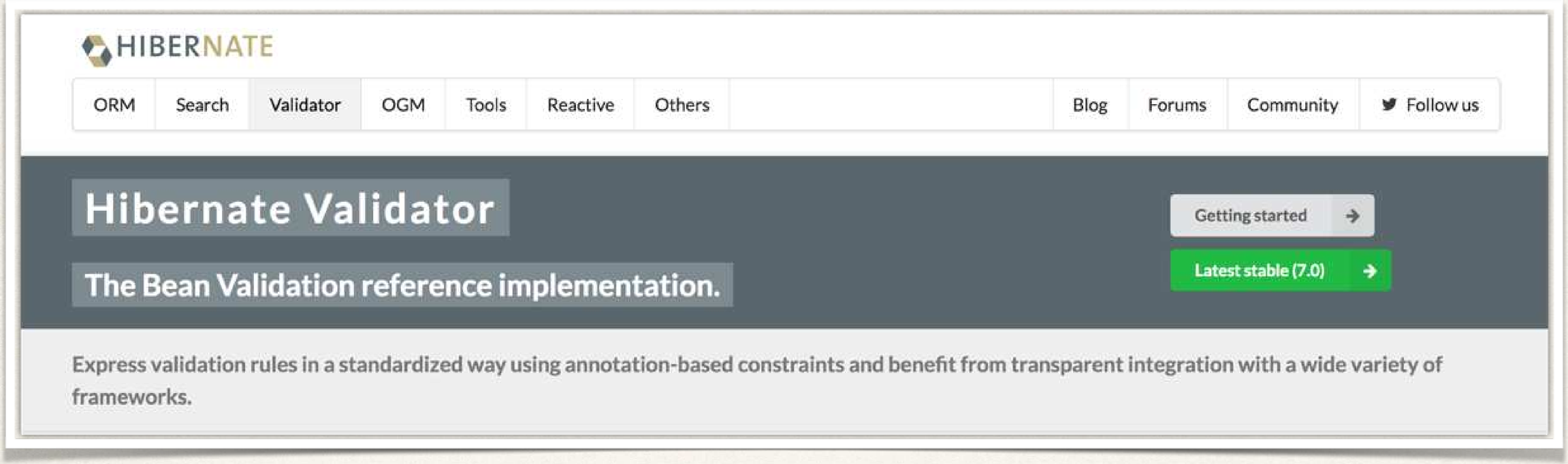
- Hibernate started as an ORM project
- But in recent years, they have expanded into other areas

The Hibernate Team to the rescue!

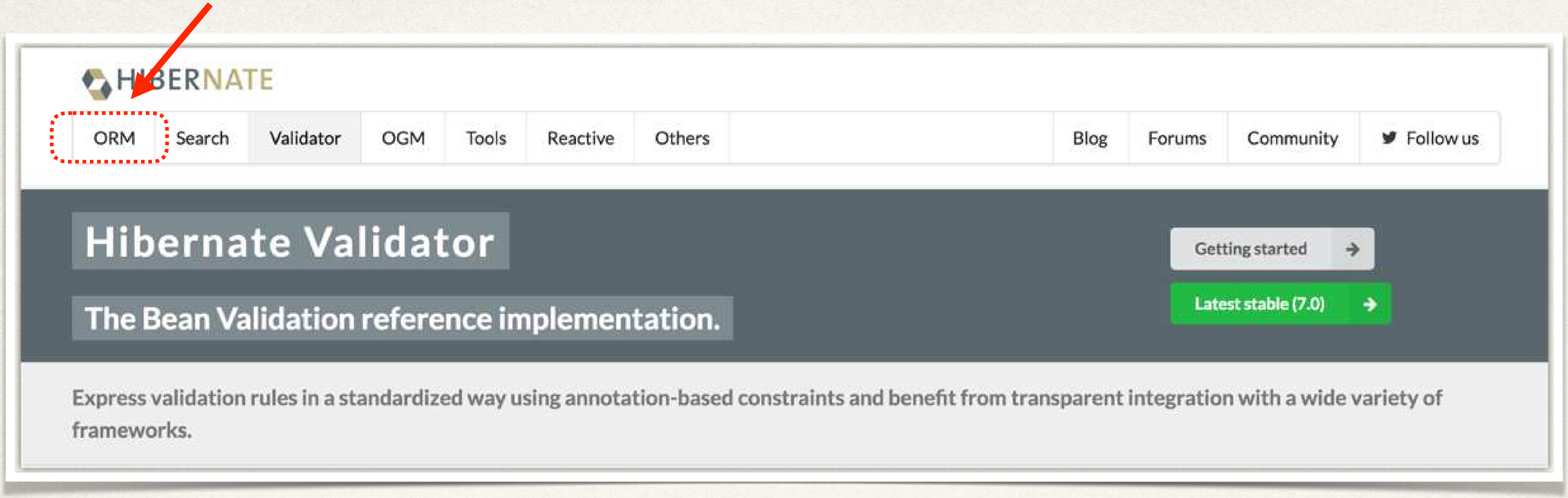
- Hibernate started as an ORM project
- But in recent years, they have expanded into other areas
- They have a fully compliant JSR-303 implementation

The Hibernate Team to the rescue!

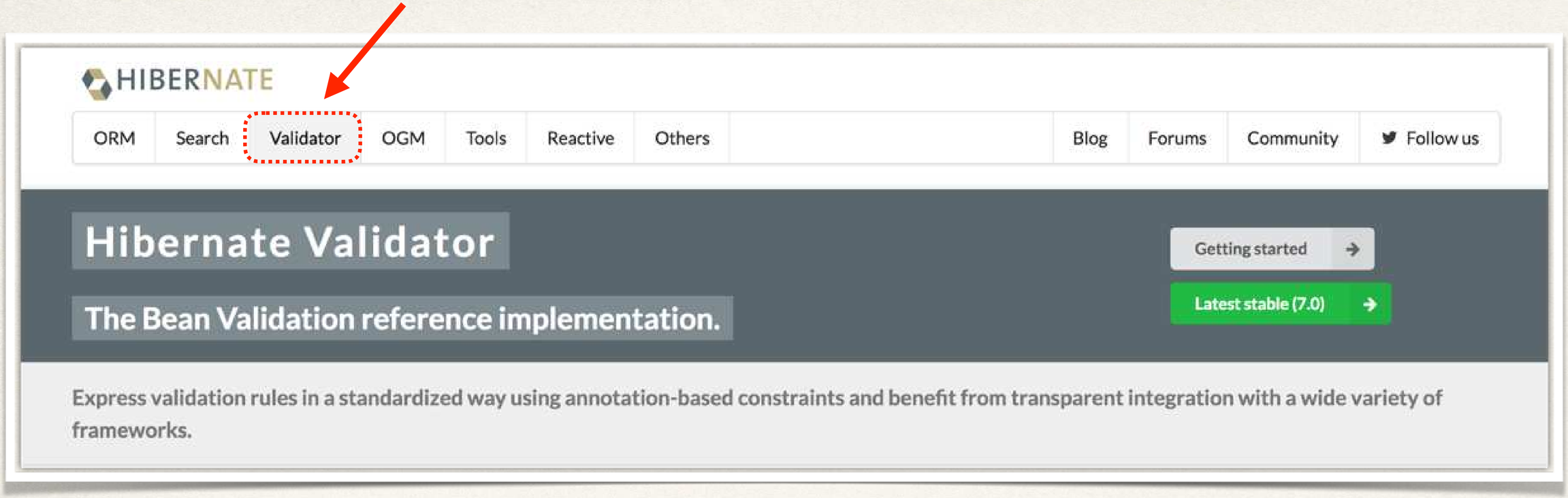
- Hibernate started as an ORM project
- But in recent years, they have expanded into other areas
- They have a fully compliant JSR-303 implementation
  - Not tied to ORM or database work … separate project



http://www.hibernate.org/validator



http://www.hibernate.org/validator



http://www.hibernate.org/validator

About the Versions ... tricky!

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

Servlets
JSP
JDBC
Enterprise Java Beans
Java Message Service
...

https://javaee.github.io/javaee-spec

- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

Servlets
JSP
JDBC
Enterprise Java Beans
Java Message Service
...

https://javaee.github.io/javaee-spec

- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)

- Allows innovation of **Jakarta EE** with community-driven approach

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

Servlets
JSP
JDBC
Enterprise Java Beans
Java Message Service
...

https://javaee.github.io/javaee-spec

- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)
- Allows innovation of **Jakarta EE** with community-driven approach
- **Jakarta EE** does not replace Java EE

About the Versions ... tricky!

- Hibernate Validator 7 is based on **Jakarta EE 9**

Servlets
JSP
JDBC
Enterprise Java Beans
Java Message Service
...

https://javaee.github.io/javaee-spec

- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)
- Allows innovation of **Jakarta EE** with community-driven approach
- **Jakarta EE** does not replace Java EE
  - Last version is Java EE 8 (August 2017)

About the Versions ... tricky!

Servlets
JSP
JDBC
Enterprise Java Beans
Java Message Service
...

https://javaee.github.io/javaee-spec

- Hibernate Validator 7 is based on **Jakarta EE 9**
- **Jakarta EE** is the community version of Java EE (rebranded, relicensed)
- Allows innovation of **Jakarta EE** with community-driven approach
- **Jakarta EE** does not replace Java EE
  - Last version is Java EE 8 (August 2017)
  - **Jakarta EE** is moving forward with **Jakarta EE 9** (December 2020) and future releases

Jakarta EE

Jakarta EE

- At the moment, main change with Jakarta EE .... package renaming

Jakarta EE

- At the moment, main change with Jakarta EE .... package renaming

- `javax.*` packages are renamed to `jakarta.*`

Jakarta EE

- At the moment, main change with Jakarta EE .... package renaming
- `javax.*` packages are renamed to `jakarta.*`
- For example

Jakarta EE

- At the moment, main change with Jakarta EE .... package renaming
- `javax.*` packages are renamed to `jakarta.*`
- For example
  - `javax.servlet.http.HttpServlet` is now **`jakarta`**`.servlet.http.HttpServlet`

Jakarta EE

- At the moment, main change with Jakarta EE .... package renaming
- `javax.*` packages are renamed to `jakarta.*`
- For example
  - `javax.servlet.http.HttpServlet` is now **`jakarta`**`.servlet.http.HttpServlet`

https://jakarta.ee/about

Okay ... what impact on Hibernate Validator??

Okay ... what impact on Hibernate Validator??

- Hibernate Validator 7 is based on **Jakarta EE 9**

Okay ... what impact on Hibernate Validator??

- Hibernate Validator 7 is based on **Jakarta EE 9**

- Spring 5 is still based on some components of Java EE (javax.*)

Okay ... what impact on Hibernate Validator??

- Hibernate Validator 7 is based on **Jakarta EE 9**

- Spring 5 is still based on some components of Java EE (javax.*)
- Spring may use Jakarta EE components in the future ... but no news yet

Okay ... what impact on Hibernate Validator??

- Hibernate Validator 7 is based on **Jakarta EE 9**

- Spring 5 is still based on some components of Java EE (javax.*)
- Spring may use Jakarta EE components in the future ... but no news yet

- As a result, Spring 5 is not compatible with Hibernate Validator 7

But wait ... I need to use latest version!!!

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator
  - Hibernate Validator 7 is for Jakarta EE 9 projects

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator
  - Hibernate Validator 7 is for Jakarta EE 9 projects
  - Hibernate **Validator 6.2** is compatible with **Spring 5**

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator
  - Hibernate Validator 7 is for Jakarta EE 9 projects
  - Hibernate **Validator 6.2** is compatible with **Spring 5**

- Hibernate Validator 6.2 has the **SAME** features as Hibernate Validator 7

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator
  - Hibernate Validator 7 is for Jakarta EE 9 projects
  - Hibernate **Validator 6.2** is compatible with **Spring 5**

- Hibernate Validator 6.2 has the **SAME** features as Hibernate Validator 7
  - So you are not being left behind by using Hibernate Validator 6.2

But wait ... I need to use latest version!!!

- Two releases of Hibernate Validator
  - Hibernate Validator 7 is for Jakarta EE 9 projects
  - Hibernate **Validator 6.2** is compatible with **Spring 5**

- Hibernate Validator 6.2 has the **SAME** features as Hibernate Validator 7
  - So you are not being left behind by using Hibernate Validator 6.2
  - Just tricky things with version numbers

In Summary

In Summary

- If you are using Spring 5 .... then use Hibernate Validator 6.x

In Summary

- If you are using Spring 5 .... then use Hibernate Validator 6.x

- If you are using Jakarta EE 9 ... then use Hibernate Validator 7.x

In Summary

- If you are using Spring 5 .... then use Hibernate Validator 6.x
- If you are using Jakarta EE 9 ... then use Hibernate Validator 7.x
- Remember

In Summary

- If you are using Spring 5 .... then use Hibernate Validator 6.x
- If you are using Jakarta EE 9 ... then use Hibernate Validator 7.x
- Remember
  - Hibernate Validator 6.2 has the **SAME** features as Hibernate Validator 7

More details on Hibernate Validator release

More details on Hibernate Validator release

https://in.relation.to/hibernate-validator

Development Process

Step-By-Step

1. Download Validation JAR files from Hibernate Validator website

2. Add JAR files to project

Spring MVC Form Validation
Required Fields

Required Fields

Fill out the form. Asterisk (*) means required.

First name:

Last name (*):

Submit

Fill out the form. Asterisk (*) means required.

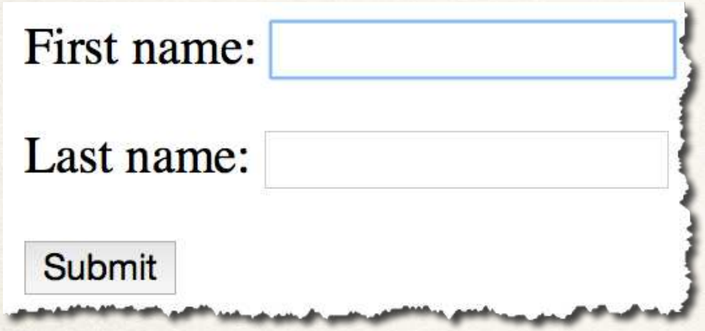
First name:

Last name (*): is required

Submit

Pulling It All Together

customer-form.jsp



Customer

Customer Controller

Customer

customer-confirmation.jsp

Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Perform validation in the Controller class

4. Update confirmation page

Step 1: Add validation rule to Customer class

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Customer {

   private String firstName;

   @NotNull(message="is required")
   @Size(min=1, message="is required")
   private String lastName;

   // getter/setter methods

}
```

Step 2: Display error message on HTML form

customer-form.jsp

```
<form:form action="processForm" modelAttribute="customer">

   First name: <form:input path="firstName" />

   <br><br>

   Last name (*): <form:input path="lastName" />
   <form:errors path="lastName" cssClass="error" />

   <br><br>

   <input type="submit" value="Submit" />

</form:form>
```

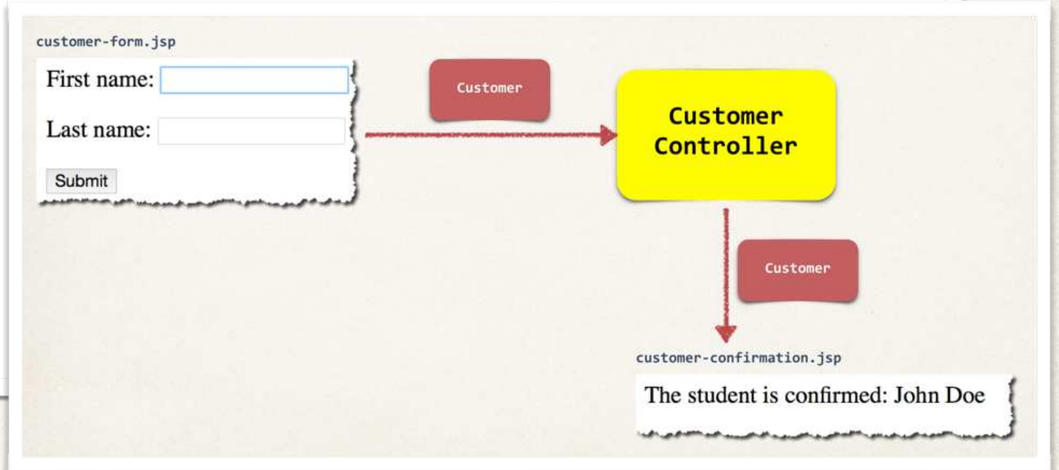
First name:

Last name (*): is required

Submit

Step 3: Perform validation in Controller class

```
@RequestMapping("/processForm")
public String processForm(
                        @Valid @ModelAttribute("customer") Customer theCustomer,
                        BindingResult theBindingResult) {

   if (theBindingResult.hasErrors()) {
      return "customer-form";
   }
   else {
      return "customer-confirmation";
   }
}
```



Step 4: Update confirmation page

```
<html>

<body>

The customer is confirmed: ${customer.firstName} ${customer.lastName}

</body>

</html>
```

Spring MVC Validation @InitBinder

White Space

- Our previous example had a problem with white space
  - **Last name** field with all whitespace **passed** … YIKES!
  - Should have **failed**!
- We need to trim whitespace from input fields

@InitBinder

Advanced

- **@InitBinder** annotation works as a pre-processor
- It will pre-process each web request to our controller
- Method annotated with **@InitBinder** is executed

@InitBinder

- We will use this to trim Strings
  - Remove leading and trailing white space
- If String only has white spaces … trim it to **null**
- Will resolve our validation problem … whew :-)

Register Custom Editor in Controller

CustomerController.java

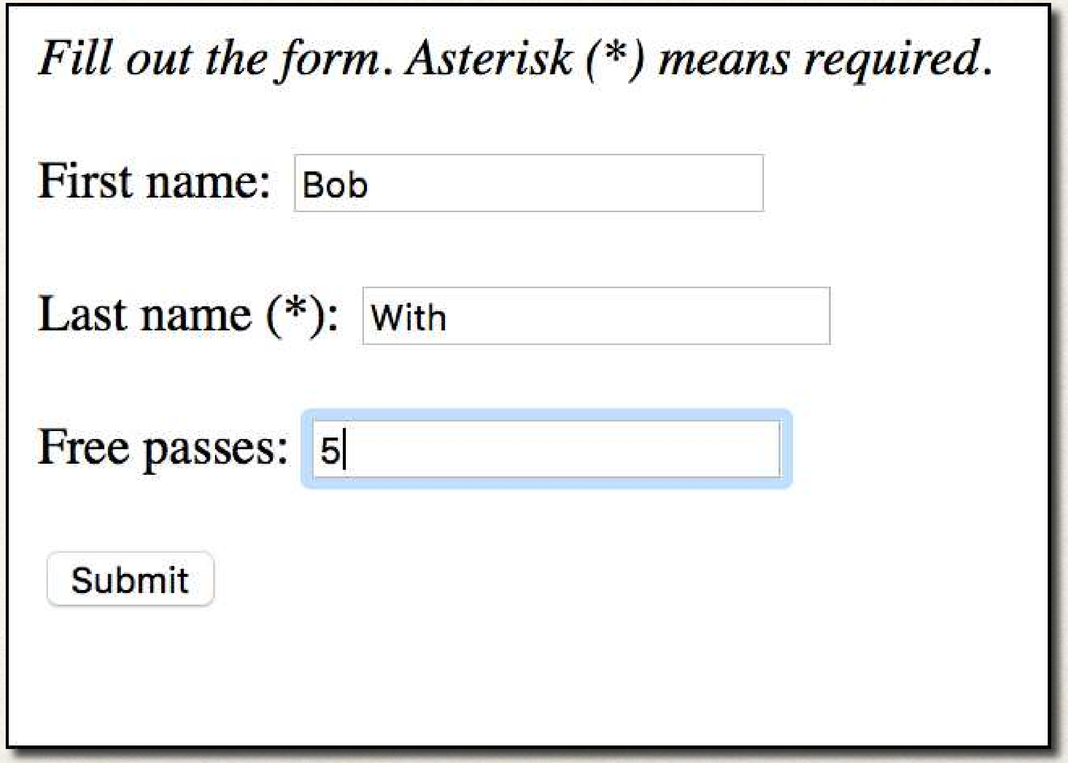
```
...

@InitBinder
public void initBinder(WebDataBinder dataBinder) {

    StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

    dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
}

...
```

Spring MVC Validation
Number Range: @Min and @Max

Validate a Number Range

- Add a new input field on our form for: **Free Passes**

- User can only enter a range: 0 to 10



Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Perform validation in the Controller class

4. Update confirmation page

Step 1: Add validation rule to Customer class

```
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;

public class Customer {

   @Min(value=0, message="must be greater than or equal to zero")
   @Max(value=10, message="must be less than or equal to 10")
   private int freePasses;

   // getter/setter methods

}
```

Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Perform validation in the Controller class

4. Update confirmation page

Spring MVC Validation
Regular Expressions

Regular Expressions

Advanced

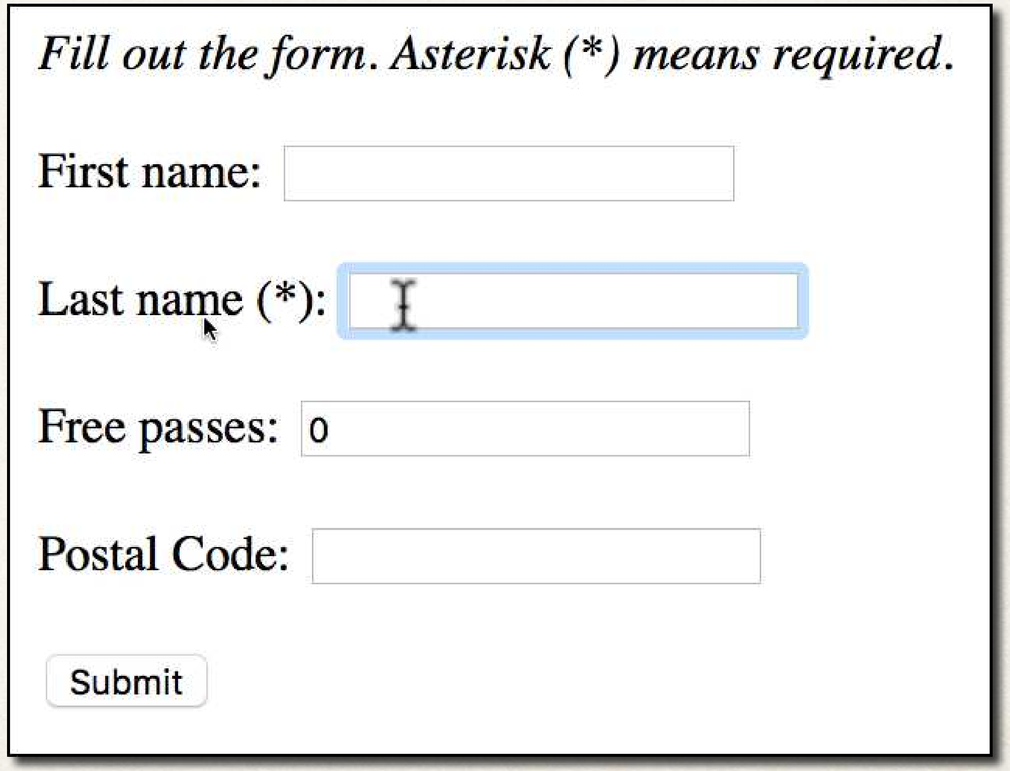
- A sequence of characters that define a search pattern
  - This pattern is used to find or match strings

- Regular Expressions is like its own language (advanced topic)
  - I will assume you already know about regular expressions

- If not, then plenty of free tutorials available
  - https://docs.oracle.com/javase/tutorial/essential/regex/

Validate a Postal Code

- Add a new input field on our form for: **Postal Code**
- User can only enter 5 chars / digits
- Apply ***Regular Expression***



Development Process

Step-By-Step

1. Add validation rule to Customer class

2. Display error messages on HTML form

3. Update confirmation page

Step 1: Add validation rule to Customer class

Advanced

```
import javax.validation.constraints.Pattern;

public class Customer {

   @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")
   private String postalCode;

   // getter/setter methods

}
```

Spring MVC Validation
Custom Validation

Custom Validation Demo

First name:

Last name:

Course Code:

Submit

Custom Validation

- Perform custom validation based on your business rules
  - Our example: Course Code must start with “LUV”
- Spring MVC calls our custom validation
- Custom validation returns boolean value for pass/fail (true/false)

Create a custom Java Annotation … from scratch

Advanced

- So far, we've used predefined validation rules: @Min, @Max, …
- For custom validation … we will create a **Custom Java Annotation**
  - **@CourseCode**

```
@CourseCode(value="LUV", message="must start with LUV")
 private String courseCode;
```

Development Process

Step-By-Step

1. Create custom validation rule

2. Add validation rule to Customer class

3. Display error messages on HTML form

4. Update confirmation page

Development Process - Drill Down

Step-By-Step

1. Create custom validation rule
   a. Create **@CourseCode** annotation
   b. Create **CourseCodeConstraintValidator**

Step 1a: Create @CourseCode annotation

Advanced

Usage Example

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Step 1a: Create @CourseCode annotation

Advanced

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {



  ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Step 1a: Create @CourseCode annotation

Advanced

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {

   // define default course code
   public String value() default "LUV";



   ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Step 1a: Create @CourseCode annotation

Advanced

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {

   // define default course code
   public String value() default "LUV";

   // define default error message
   public String message() default "must start with LUV";

   ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Step 1b: Create CourseCodeConstraintValidator

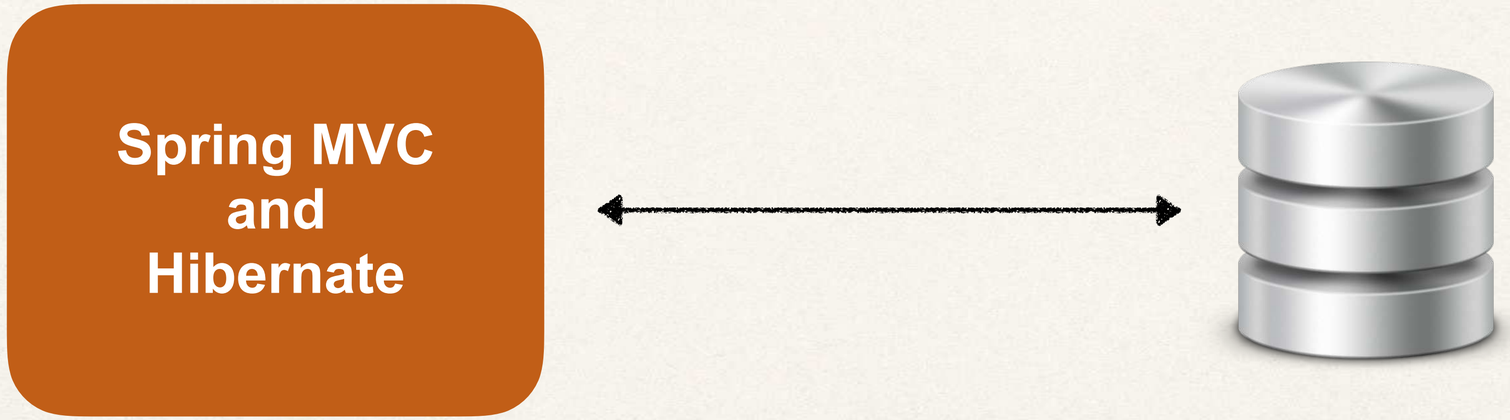
Advanced

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CourseCodeConstraintValidator
    implements ConstraintValidator<CourseCode, String> {

    private String coursePrefix;

    @Override
    public void initialize(CourseCode theCourseCode) {
        coursePrefix = theCourseCode.value();
    }

    @Override
    public boolean isValid(String theCode,
                           ConstraintValidatorContext theConstraintValidatorContext) {

        boolean result;

        if (theCode != null) {
            result = theCode.startsWith(coursePrefix);
        }
        else {
            result = true;
        }

        return result;
    }
}
```

Create a Database Application

Class Project

Full working *Spring MVC and Hibernate* application that connects to a database



Customer Relationship Management - CRM

Customer Relationship Management - CRM

Step-By-Step

- Set up Database Dev Environment
- List Customers
- Add a new Customer
- Update a Customer
- Delete a Customer

Setup Database Table

Two Database Scripts

1. Folder: **sql-scripts**
   - **01-create-user.sql**
   - **02-customer-tracker.sql**

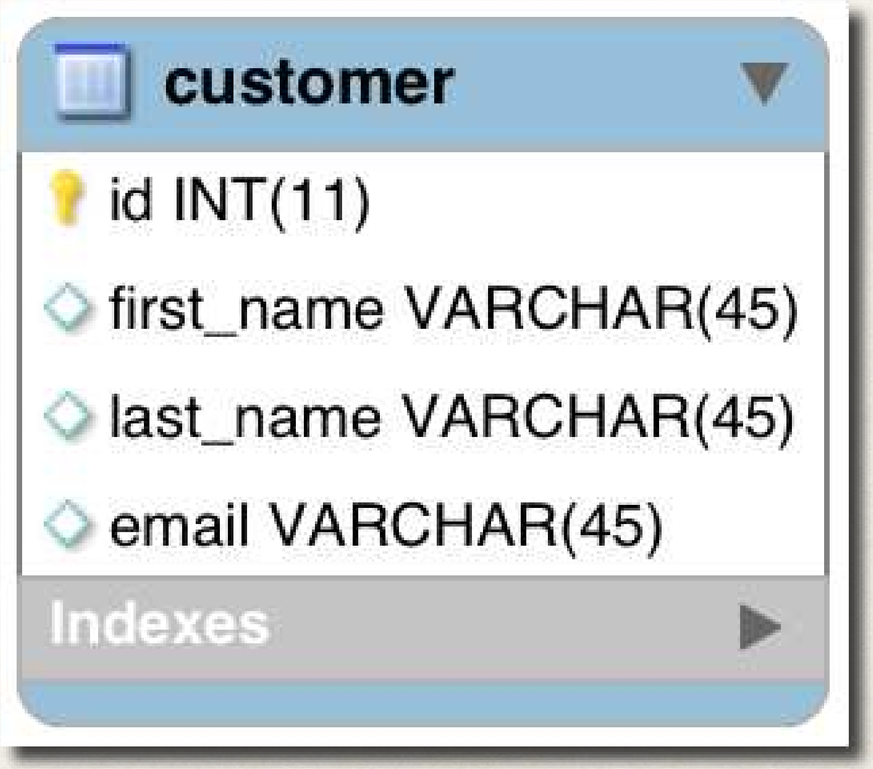
About: 01-create-user.sql

1. Create a new MySQL user for our application
   - user id: **springstudent**
   - password: **springstudent**

About: 02-customer-tracker.sql

1. Create a new database table: **customer**

2. Load table with sample data



Setup Project - Test DB Connection

Test DB Connection

1. Set up our Eclipse project

2. Add JDBC Driver for MySQL

3. Sanity test … make sure we can connect :-)

Setup Dev Environment - Part 2

Configuration for Spring + Hibernate

Step-By-Step

1. Define database dataSource / connection pool
2. Setup Hibernate session factory
3. Setup Hibernate transaction manager
4. Enable configuration of transactional annotations

Placement of Configurations

Add the following configurations in your Spring MVC configuration file

For our example
spring-mvc-crud-demo-servlet.xml

Step 1: Define database dataSource / connection pool

```
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">

 <property name="driverClass" value="com.mysql.jdbc.Driver" />
 <property name="jdbcUrl"
            value="jdbc:mysql://localhost:3306/web_customer_tracker?useSSL=false" />

 <property name="user" value="springstudent" />
 <property name="password" value="springstudent" />

 <!-- these are connection pool properties for C3P0 -->
 <property name="minPoolSize" value="5" />
 <property name="maxPoolSize" value="20" />
 <property name="maxIdleTime" value="30000" />
</bean>
```

Step 2: Setup Hibernate session factory

```
<bean id="sessionFactory"
   class="“org.springframework.orm.hibernate5.LocalSessionFactoryBean">

   <property name="dataSource" ref="myDataSource" />
   <property name="packagesToScan" value="com.luv2code.springdemo.entity" />

   <property name="hibernateProperties">
     <props>
       <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
       <prop key="hibernate.show_sql">true</prop>
     </props>
   </property>

</bean>
```

Step 3: Setup Hibernate transaction manager

```
<bean id="myTransactionManager"
      class="“org.springframework.orm.hibernate5.HibernateTransactionManager">

   <property name="sessionFactory" ref="sessionFactory"/>

</bean>
```

Step 4: Enable configuration of transactional annotations

```
<tx:annotation-driven transaction-manager="myTransactionManager" />
```

Configuration for Spring + Hibernate

Step-By-Step

1. Define database dataSource / connection pool
2. Setup Hibernate session factory
3. Setup Hibernate transaction manager
4. Enable configuration of transactional annotations

Setup Dev Environment

Setup Dev Environment

Step-By-Step

1. Copy starter config files
    1. web.xml and spring config

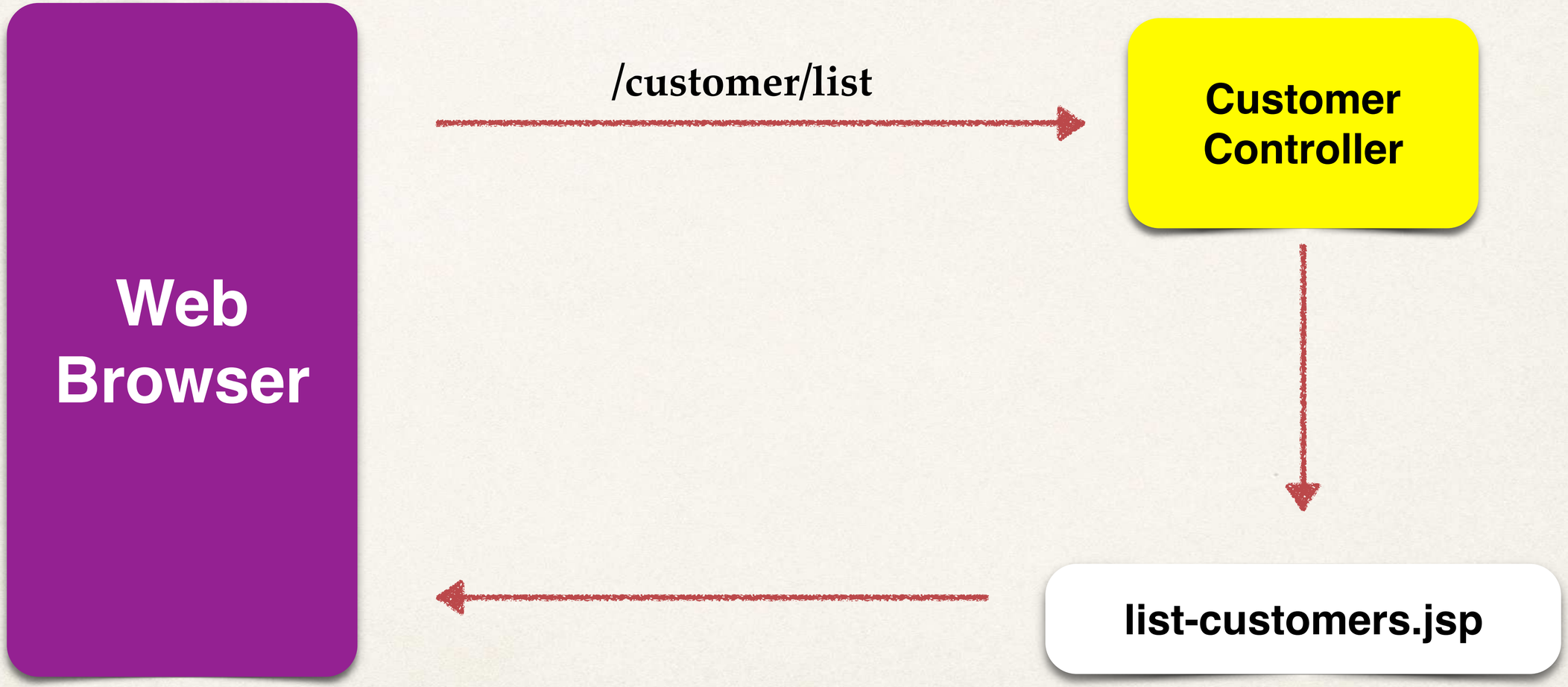
2. Copy over basic libs
    1. Jakarta Commons and JSTL

3. Copy latest Spring JAR files

4. Copy latest Hibernate JAR files

Test Basic Spring MVC Controller

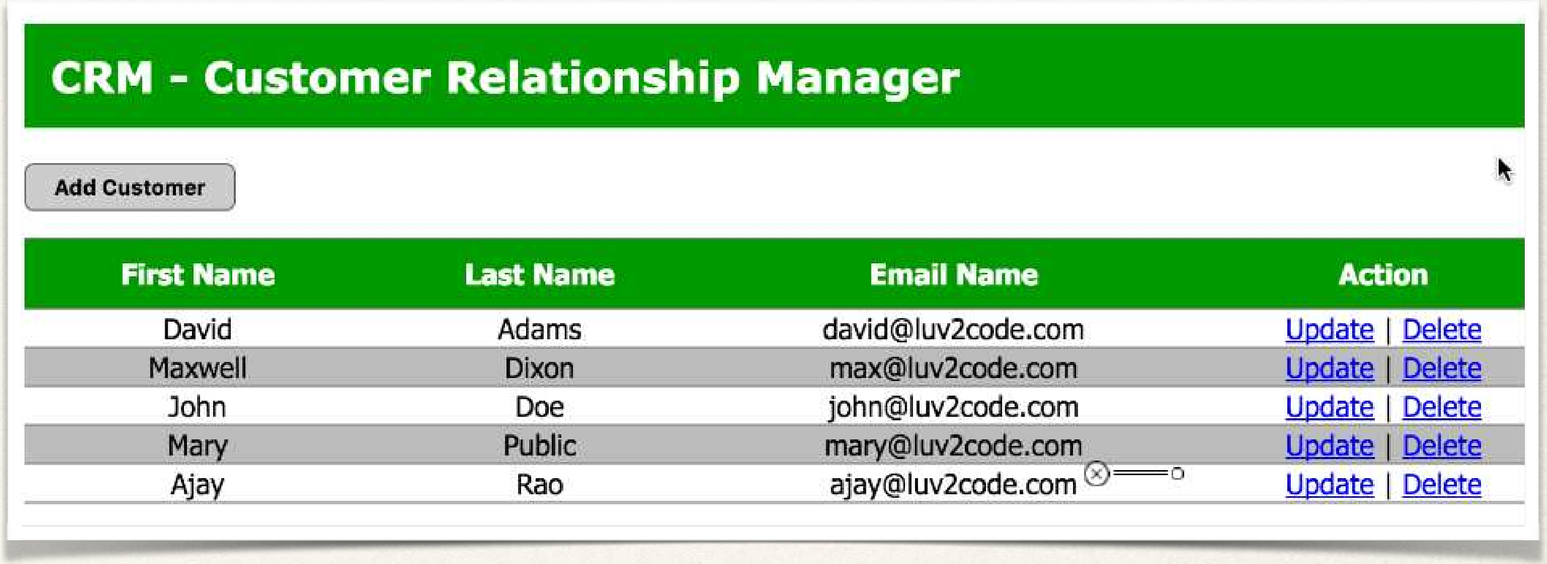
Customer Controller



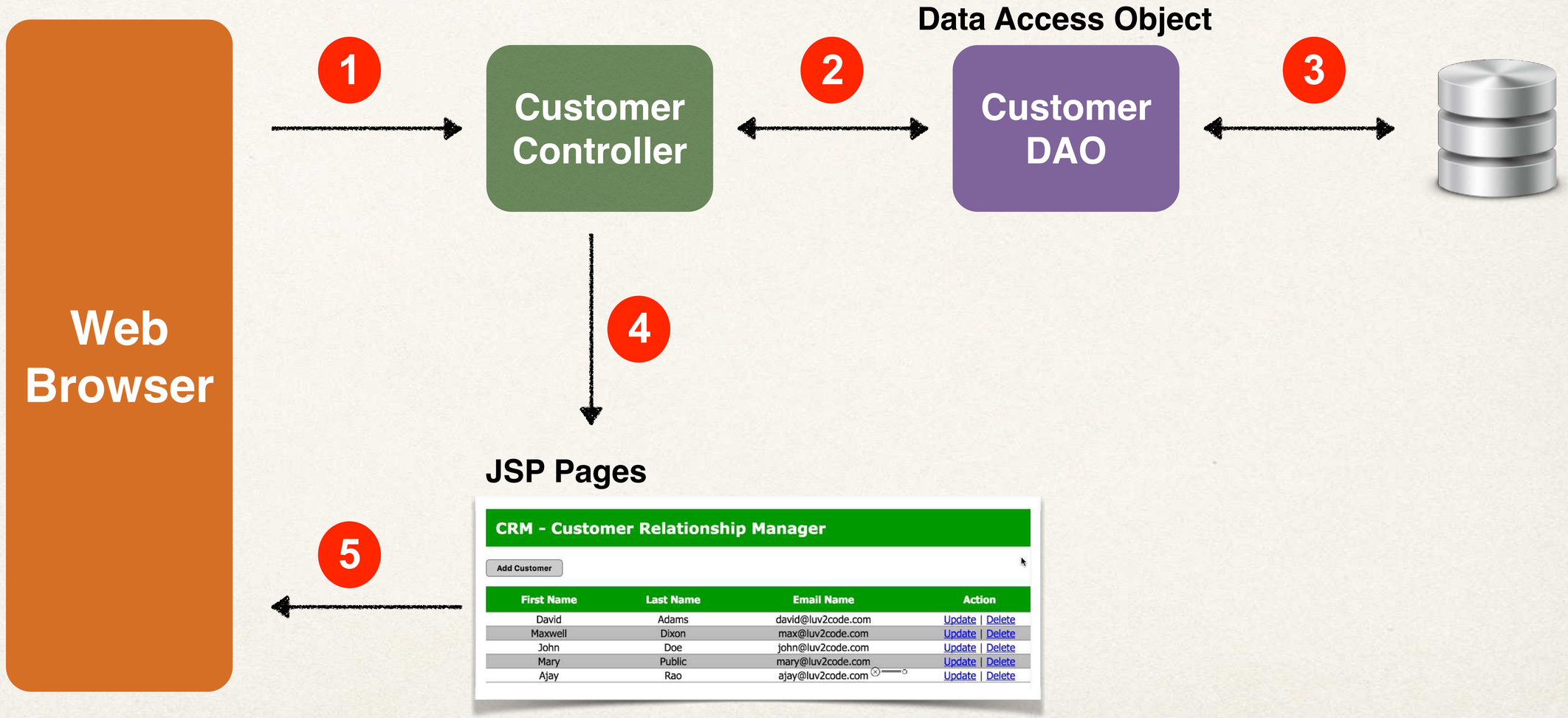
Sample App Architecture

Sample App Features - Refresher

- List Customers
- Add a new Customer
- Update a Customer
- Delete a Customer

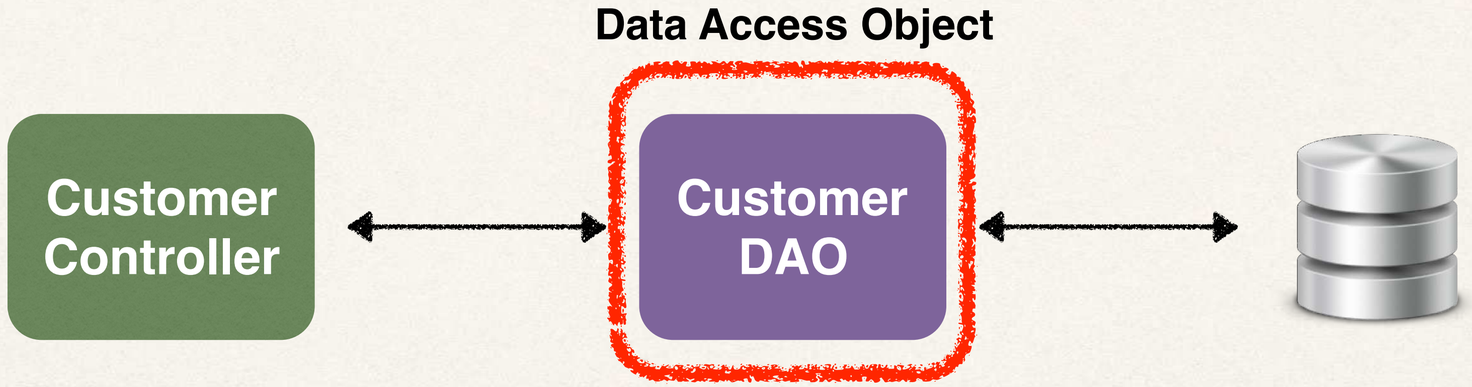


Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

Customer Data Access Object

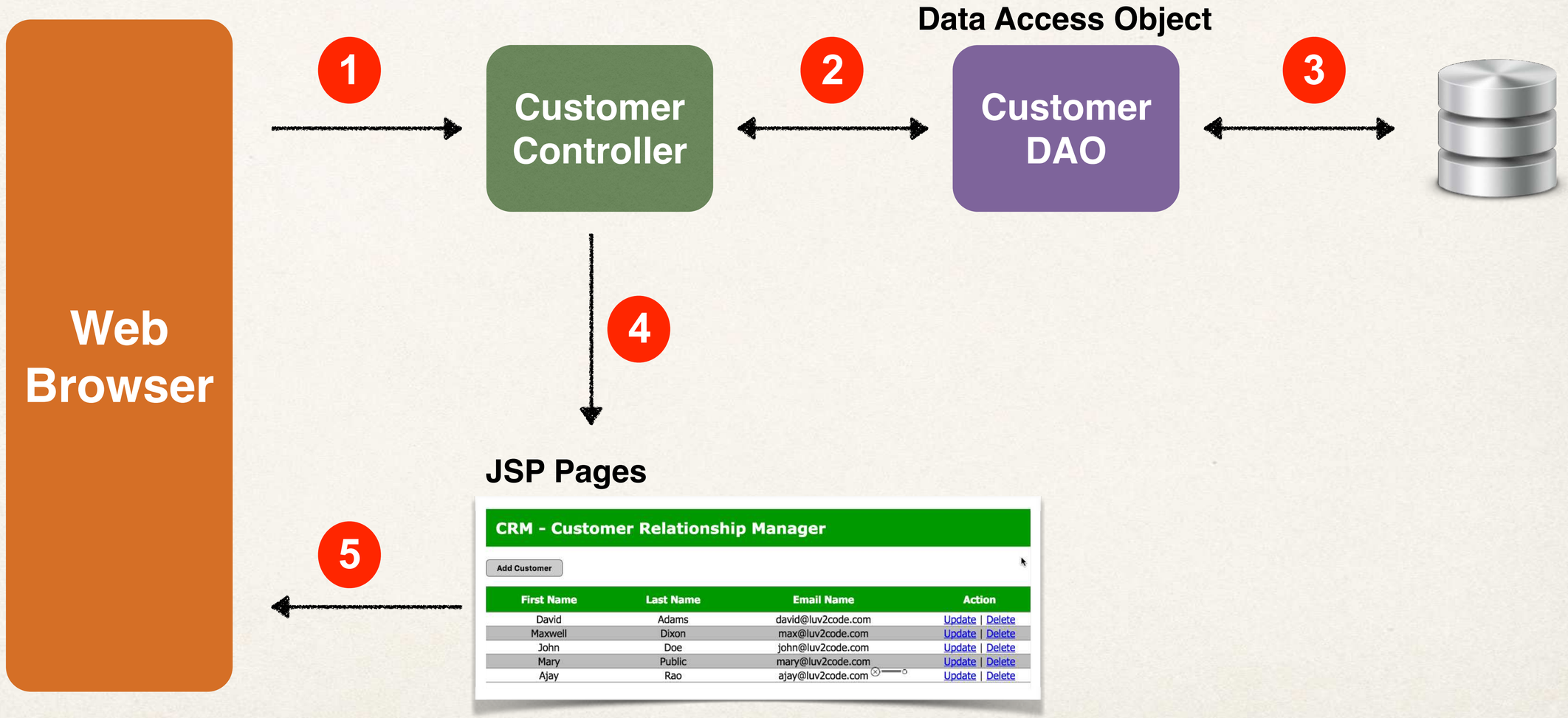
- Responsible for interfacing with the database
- This is a common design pattern: **Data Access Object (DAO)**



Customer Data Access Object

Methods
saveCustomer(…)
getCustomer(…)
getCustomers()
updateCustomer(…)
deleteCustomer(…)

Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

List Customers - Dev Process

List Customers

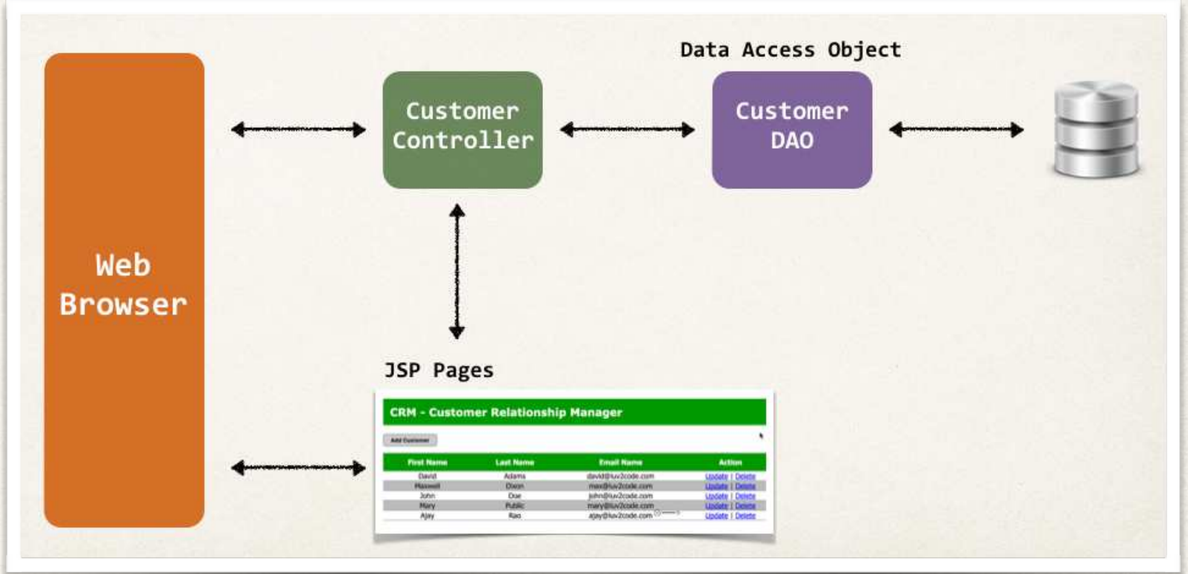
Step-By-Step

1. Create **Customer.java**

2. Create **CustomerDAO.java**
    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page: **list-customers.jsp**



List Customers

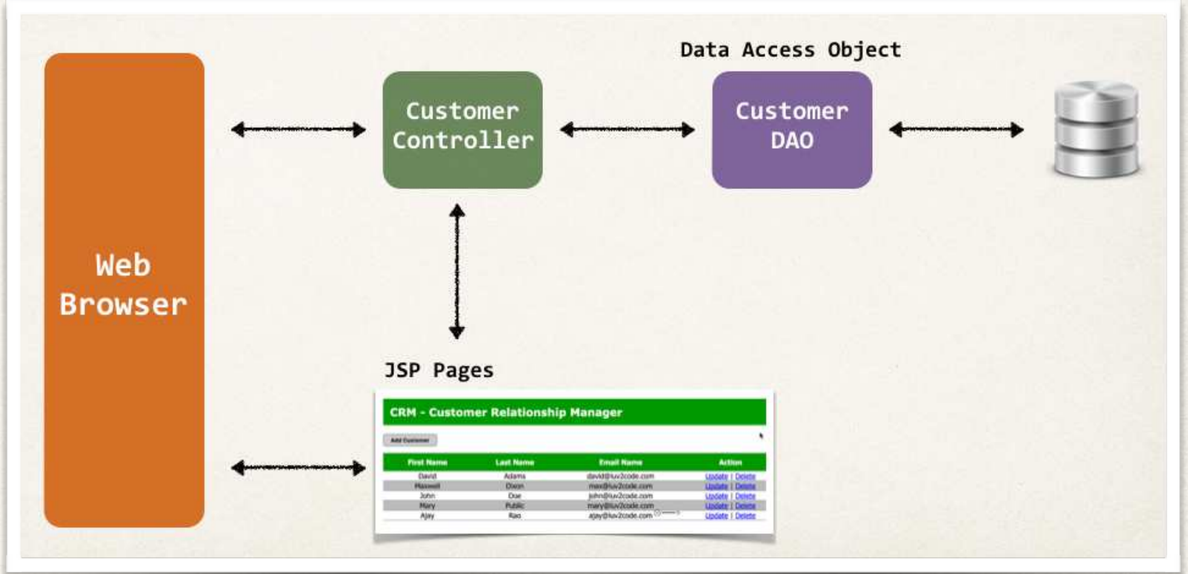
Step-By-Step

1. Create **Customer.java**

2. Create **CustomerDAO.java**
    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page: **list-customers.jsp**



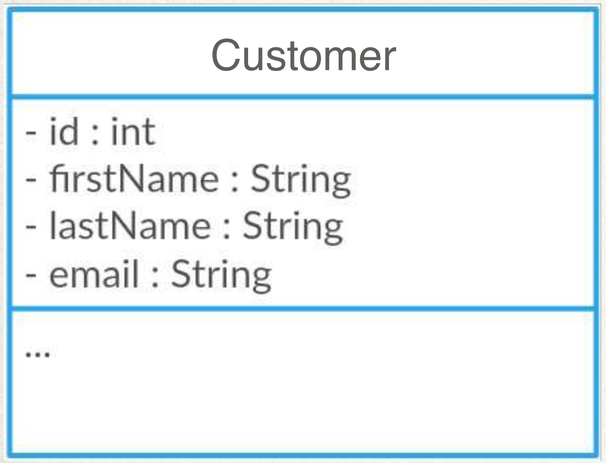
Hibernate Terminology - Refresh

Entity Class

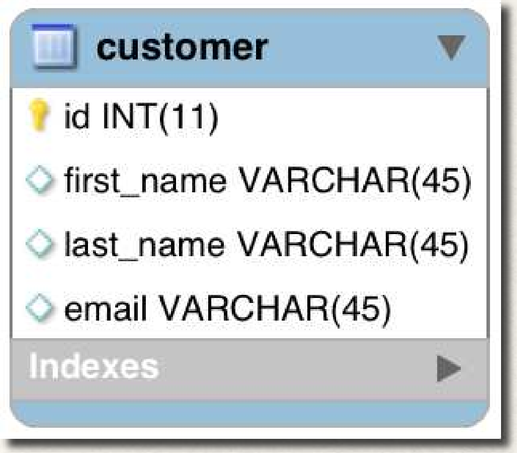
Java class that is mapped to a database table

Object-to-Relational Mapping (ORM)

Java Class



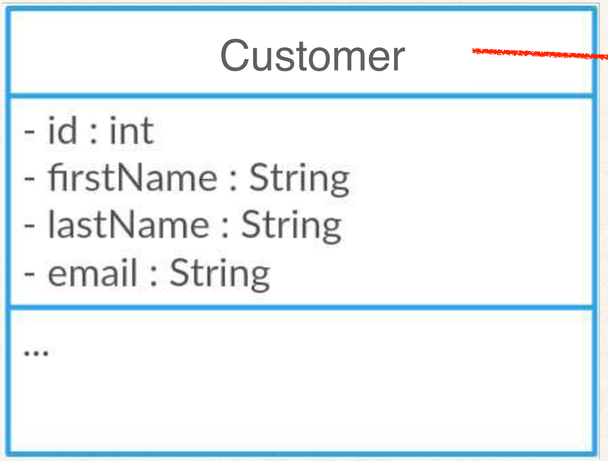
Database Table



Step 1: Map class to database table

```
@Entity
@Table(name="customer")
public class Customer {

  ...

}
```

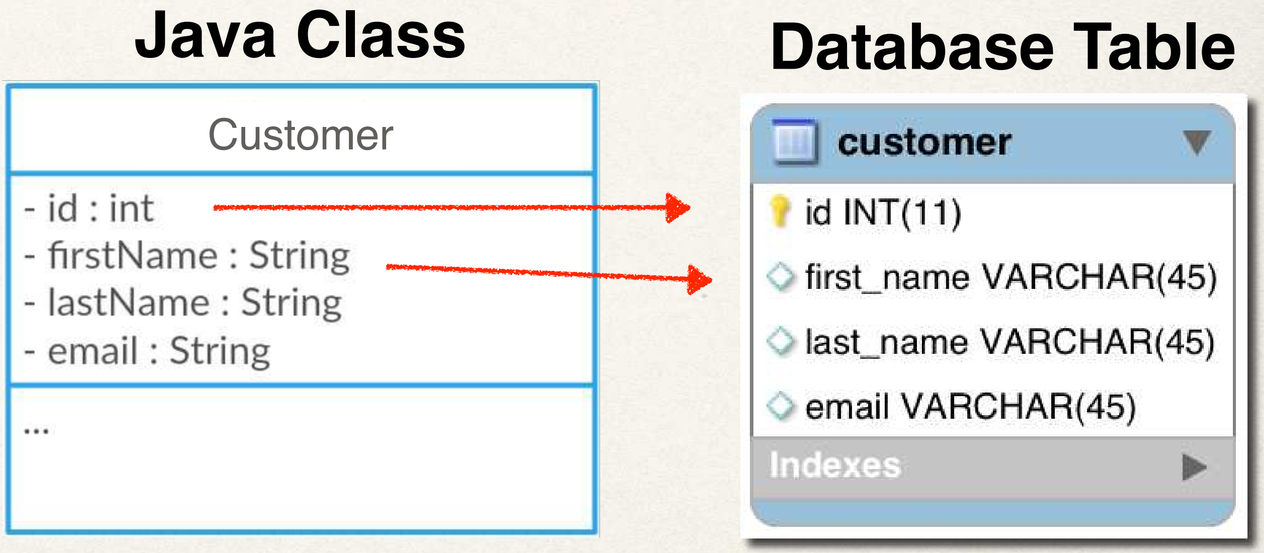
Java Class



Database Table

Step 2: Map fields to database columns

```
@Entity
@Table(name="customer")
public class Customer {

   @Id
   @Column(name="id")
   private int id;

   @Column(name="first_name")
   private String firstName;
  ...
}
```



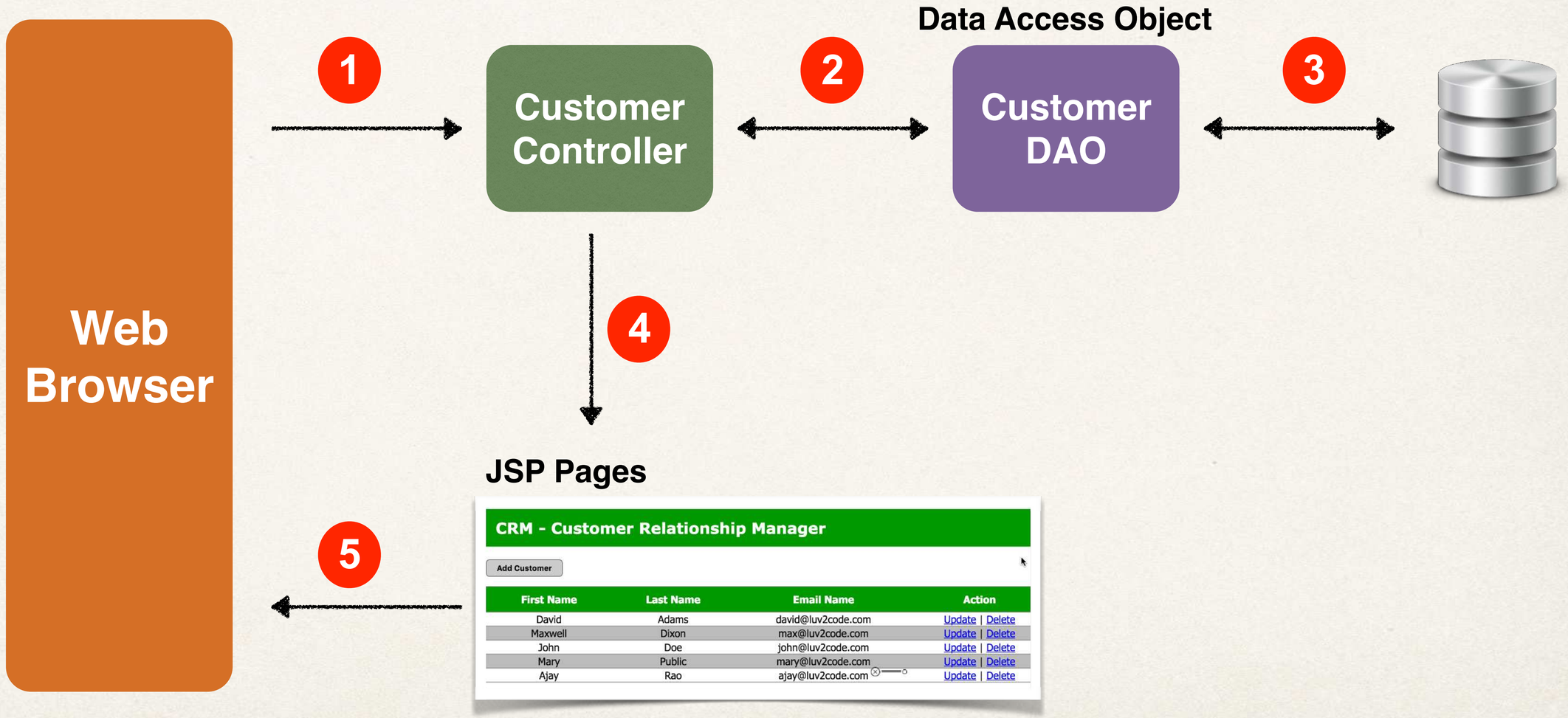
Entity Scanning

- Remember our Spring MVC config file?

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

   <property name="dataSource" ref="myDataSource" />

   <property name="packagesToScan" value="com.luv2code.springdemo.entity" />

   ...

</bean>
```

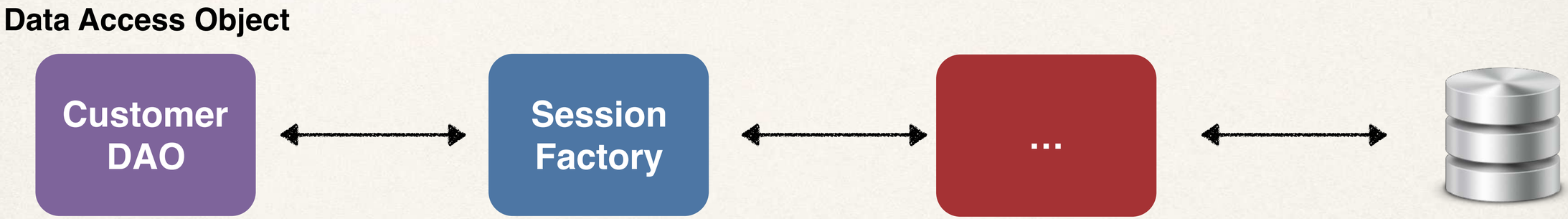
Define Data Access Object

Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

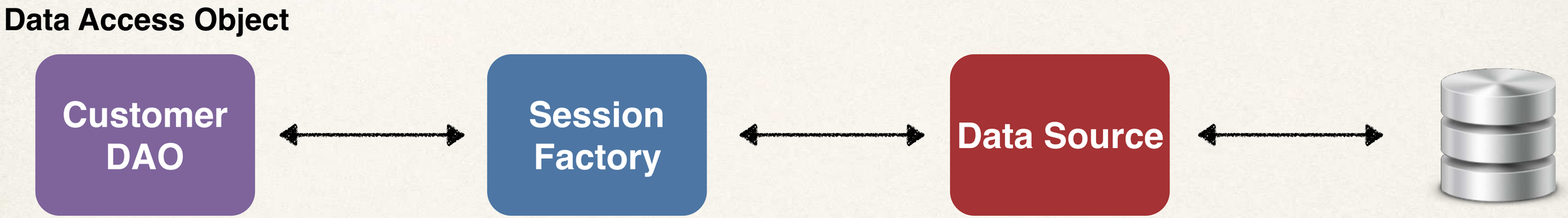
Customer Data Access Object

- For Hibernate, our DAO needs a Hibernate SessionFactory



Hibernate Session Factory

- Our Hibernate Session Factory needs a Data Source
  - The data source defines database connection info



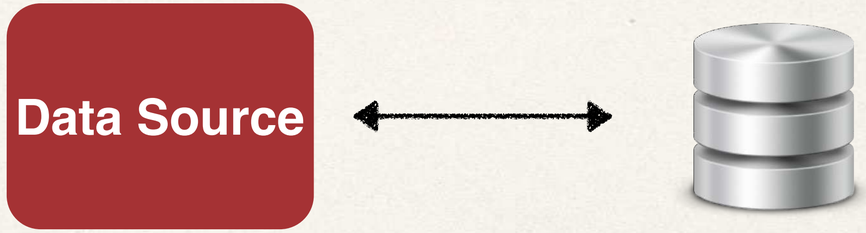
Dependencies

These are all dependencies!

We will wire them together with Dependency Injection (DI)

Data Source

```
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="“close">

 <property name="driverClass" value="com.mysql.jdbc.Driver" />
 <property name="jdbcUrl"
                value="jdbc:mysql://localhost:3306/web_customer_tracker?useSSL=false" />

     … user id, password etc …
</bean>
```



Session Factory

```
<bean id="sessionFactory"
  class="“org.springframework.orm.hibernate5.LocalSessionFactoryBean">

  <property name="dataSource" ref="myDataSource" />

  ...

</bean>
```

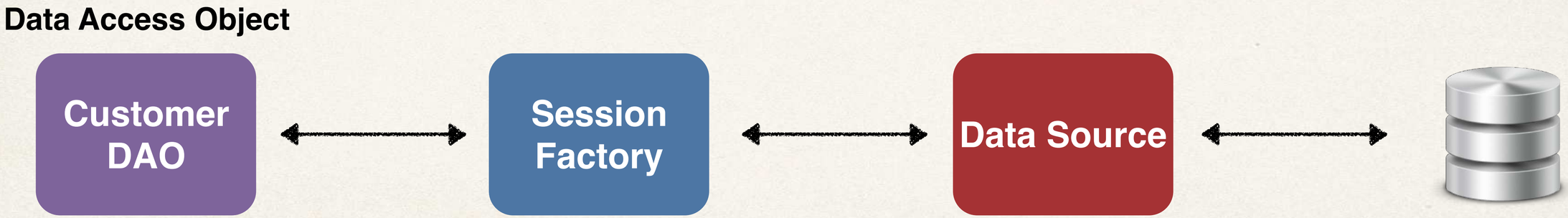
Customer DAO

Step-By-Step

1. Define DAO interface

2. Define DAO implementation

- Inject the session factory



Step 1: Define DAO interface

```
public interface CustomerDAO {

   public List<Customer> getCustomers();

}
```

Step 2: Define DAO implementation

```
public class CustomerDAOImpl implements CustomerDAO {

  @Autowired
  private SessionFactory sessionFactory;

  public List<Customer> getCustomers() {
    …
  }

}
```

Spring @Transactional

- Spring provides an **@Transactional** annotation

- **Automagically** begin and end a transaction for your Hibernate code
  - No need for you to explicitly do this in your code

- This Spring **magic** happens behind the scenes

Flash back - Standalone Hibernate code

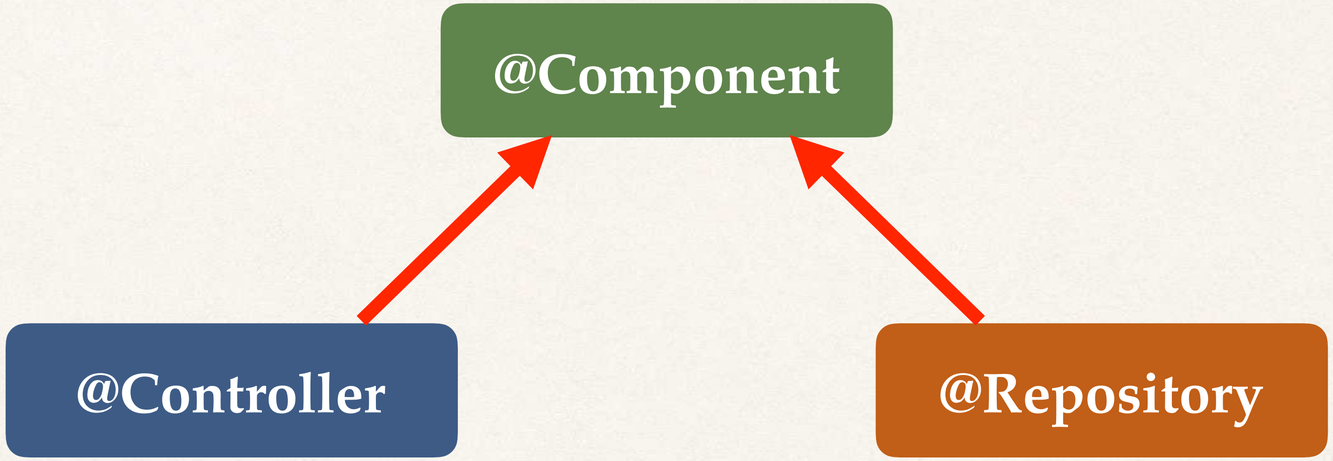
```
// start a transaction
session.beginTransaction();


// DO YOUR HIBERNATE STUFF HERE
// ...


// commit transaction
session.getTransaction().commit();
```

Spring @Transactional Magic

```
@Transactional
public List<Customer> getCustomers() {

    // get the current hibernate session
    Session currentSession = sessionFactory.getCurrentSession();

    // create a query
    Query<Customer> theQuery =
        currentSession.createQuery("from Customer", Customer.class);

    // get the result list
    List<Customer> customers = theQuery.getResultList();

    return customers;
}
```

Specialized Annotation for DAOs

- Spring provides the **@Repository** annotation



Specialized Annotation for DAOs

- Applied to DAO implementations
- Spring will automatically register the DAO implementation
  - thanks to component-scanning
- Spring also provides translation of any JDBC related exceptions

Updates for the DAO implementation

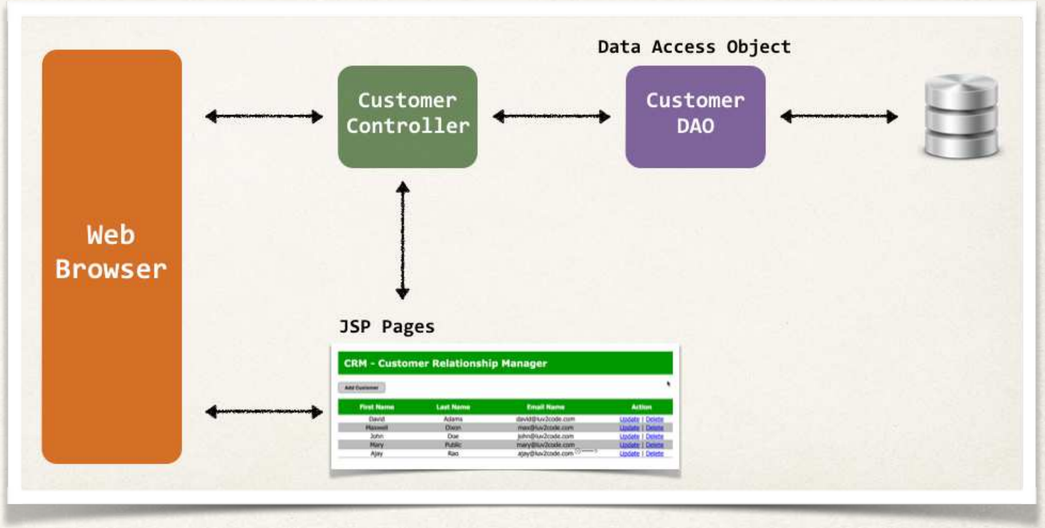
```
@Repository
public class CustomerDAOImpl implements CustomerDAO {

  @Autowired
  private SessionFactory sessionFactory;

 @Transactional
  public List<Customer> getCustomers() {
    ...
  }

}
```

Inject DAO into Controller

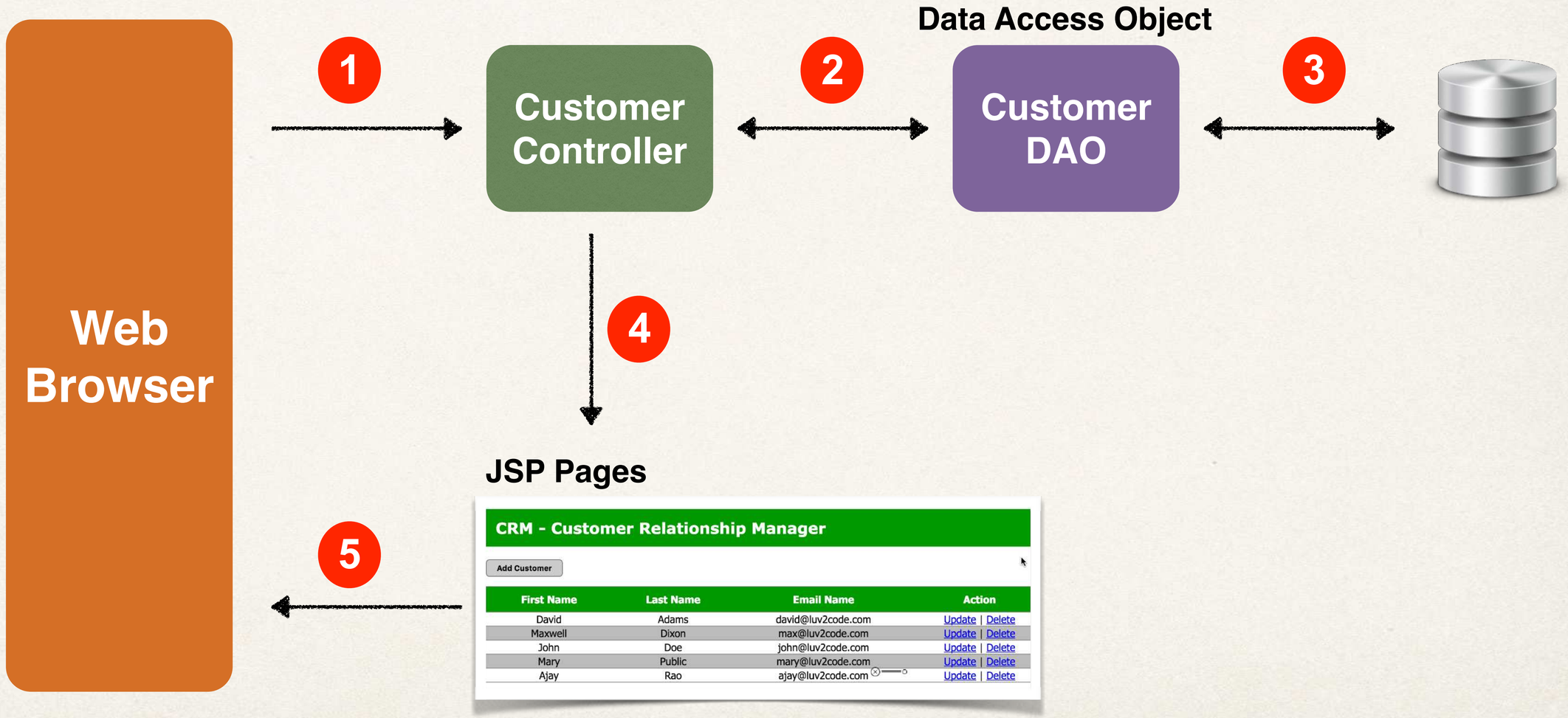
List Customers

Step-By-Step

1. Create **Customer.java**
2. Create **CustomerDAO.java**
   1. and **CustomerDAOImpl.java**
3. Create **CustomerController.java**
4. Create JSP page: **list-customers.jsp**



Big Picture

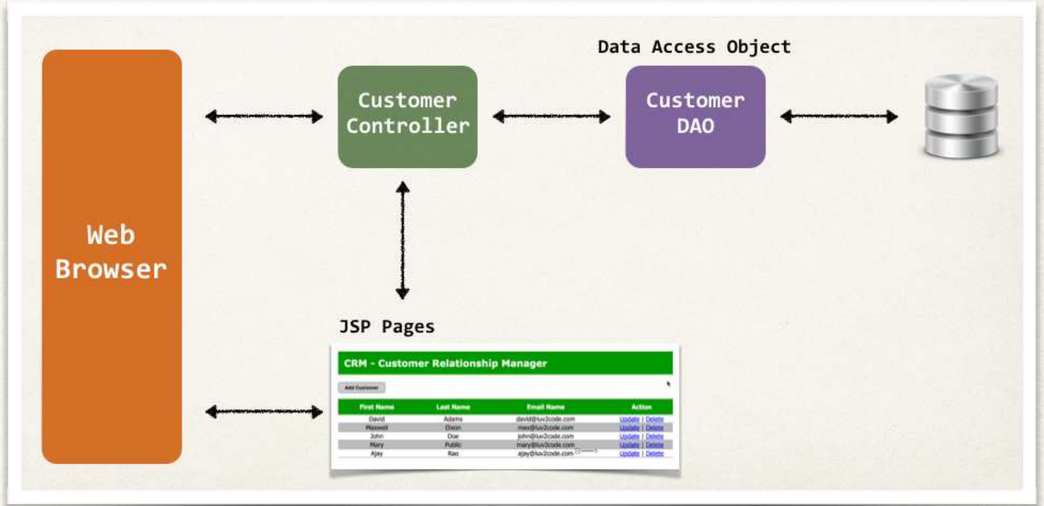

Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

Create JSP View Page

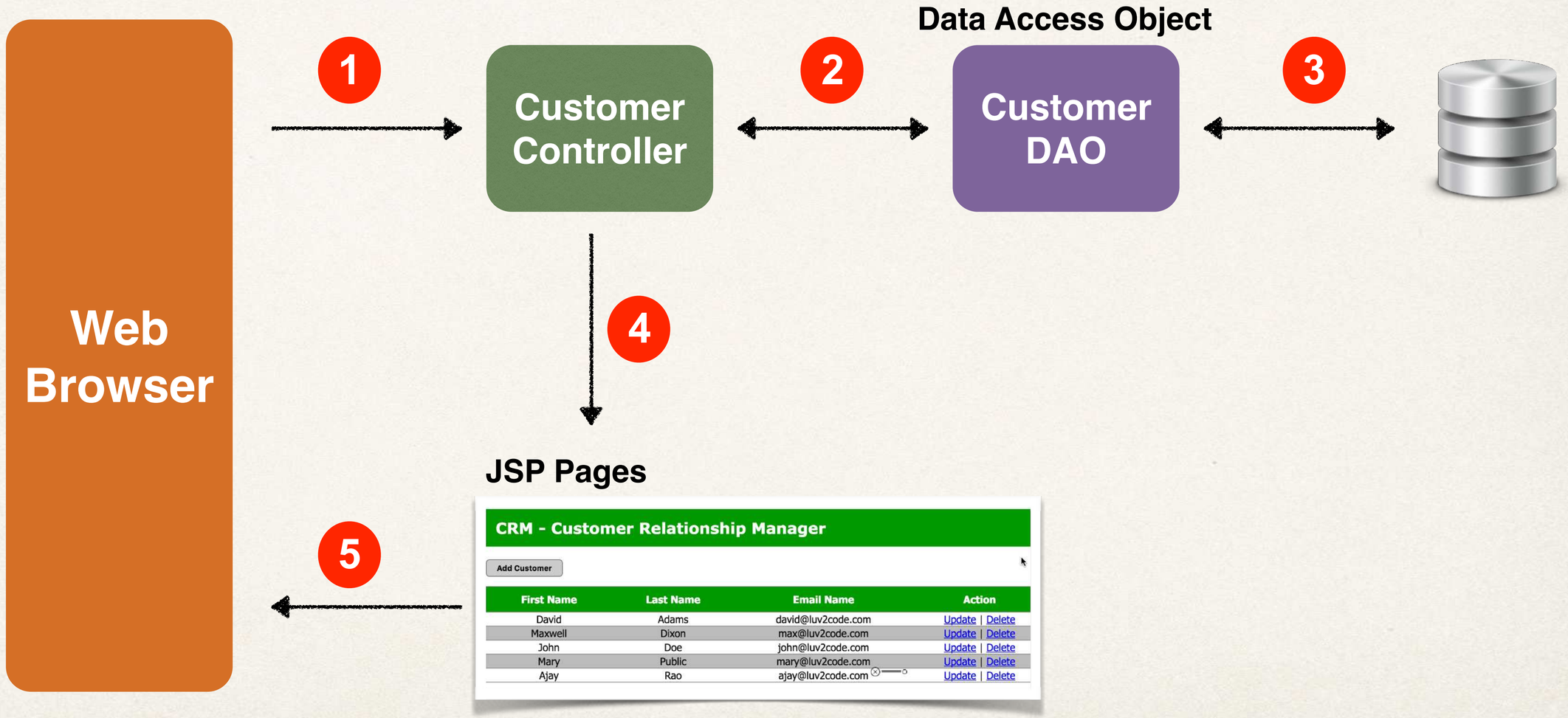
List Customers

Step-By-Step

1. Create **Customer.java**
2. Create **CustomerDAO.java**
   1. and **CustomerDAOImpl.java**
3. Create **CustomerController.java**
4. Create JSP page: **list-customers.jsp**



Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

First Version - Plain

CRM - Customer Relationship Manager

First Name	Last Name	Email
David	Adams	david@luv2code.com
John	Doe	john@luv2code.com
Ajay	Rao	ajay@luv2code.com
Mary	Public	mary@luv2code.com
Maxwell	Dixon	max@luv2code.com

Adding CSS to Spring MVC App

First Version - Plain

CRM - Customer Relationship Manager

First Name	Last Name	Email
David	Adams	david@luv2code.com
John	Doe	john@luv2code.com
Ajay	Rao	ajay@luv2code.com
Mary	Public	mary@luv2code.com
Maxwell	Dixon	max@luv2code.com

Apply CSS and make it Pretty

CRM - Customer Relationship Manager

Add Customer

First Name	Last Name	Email Name
David	Adams	david@luv2code.com
Maxwell	Dixon	max@luv2code.com
John	Doe	john@luv2code.com
Mary	Public	mary@luv2code.com
Ajay	Rao	ajay@luv2code.com

Development Process

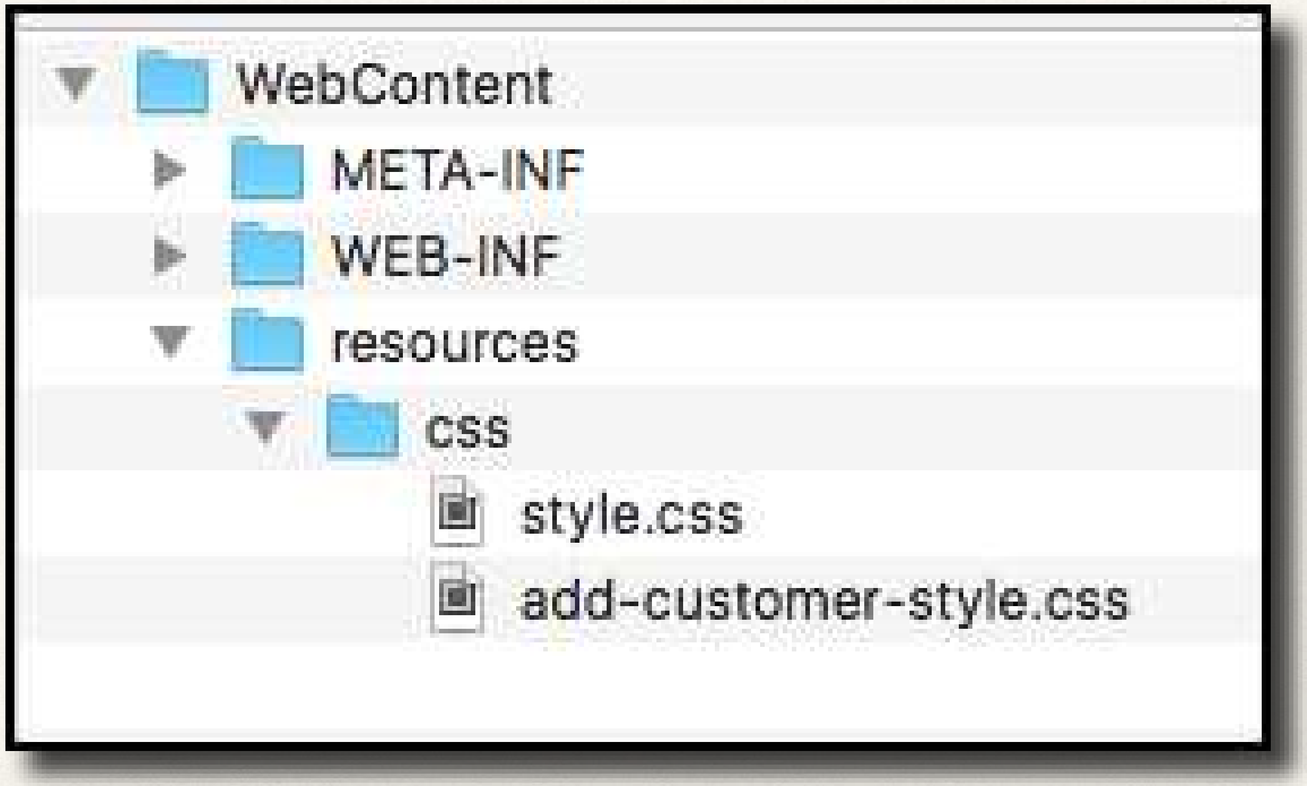
Step-By-Step

1. Place CSS in a ‘resources’ directory

2. Configure Spring to serve up ‘resources’ directory

3. Reference CSS in your JSP

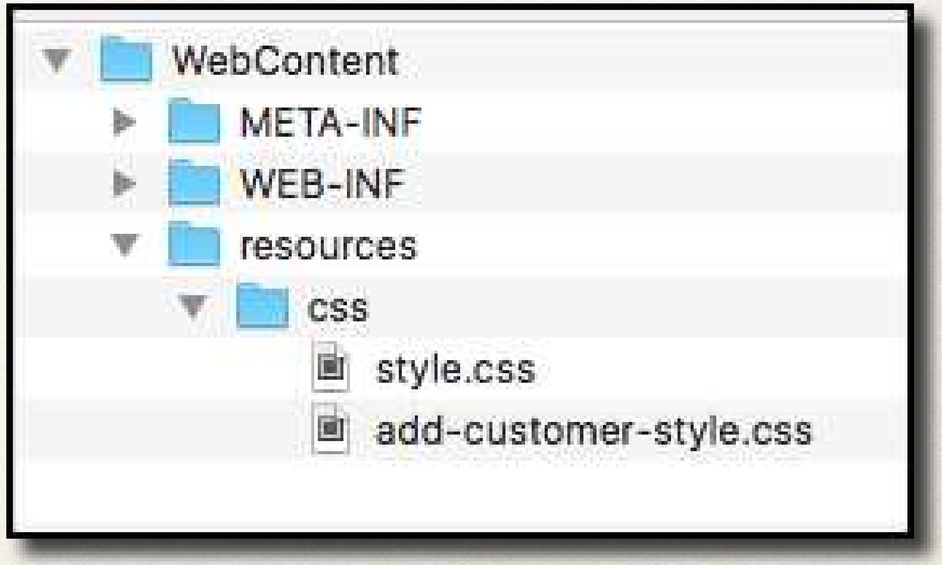
Step 1: Place CSS in 'resources' directory



Step 2: Configure Spring to serve up ‘resources’ directory

File: spring-mvc-crud-demo-servlet.xml

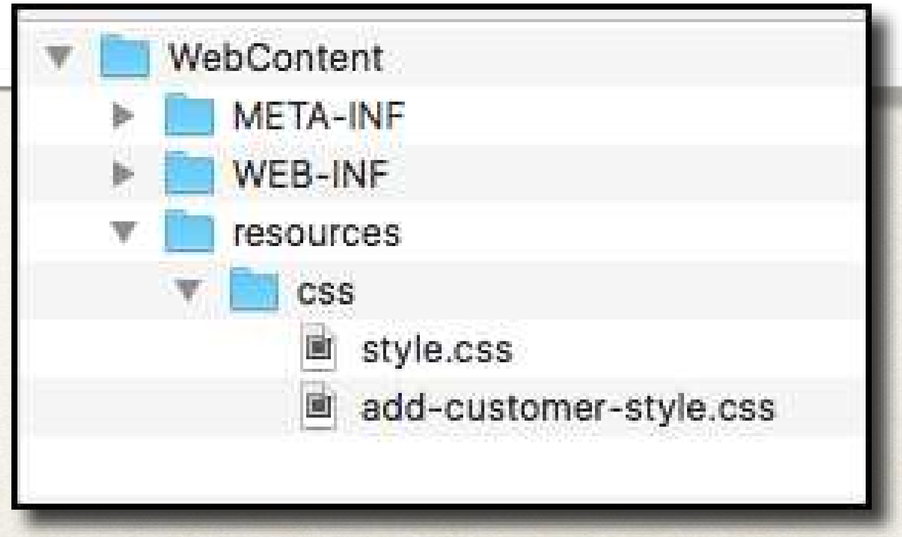
```
<mvc:resources location="“/resources/" mapping="/resources/**" />
```



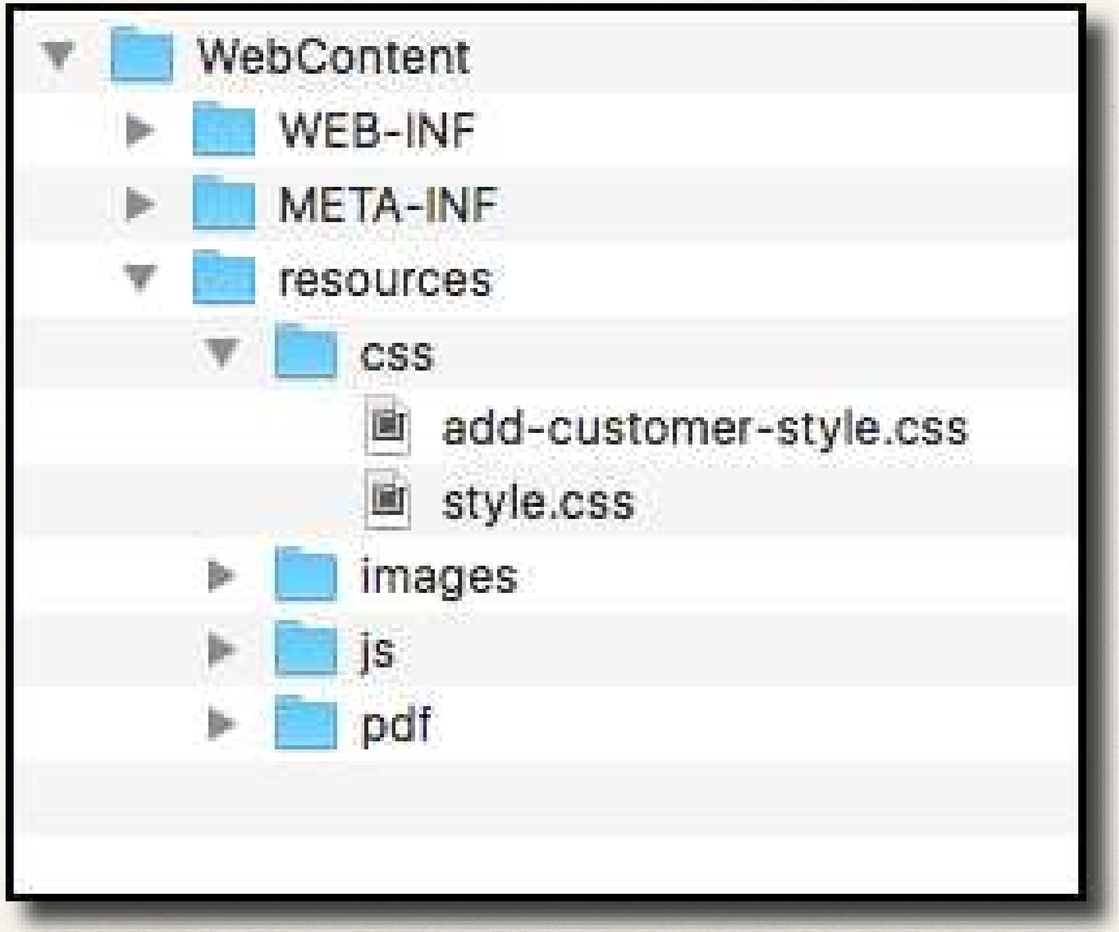
Step 3: Reference CSS in your JSP

File: list-customers.jsp

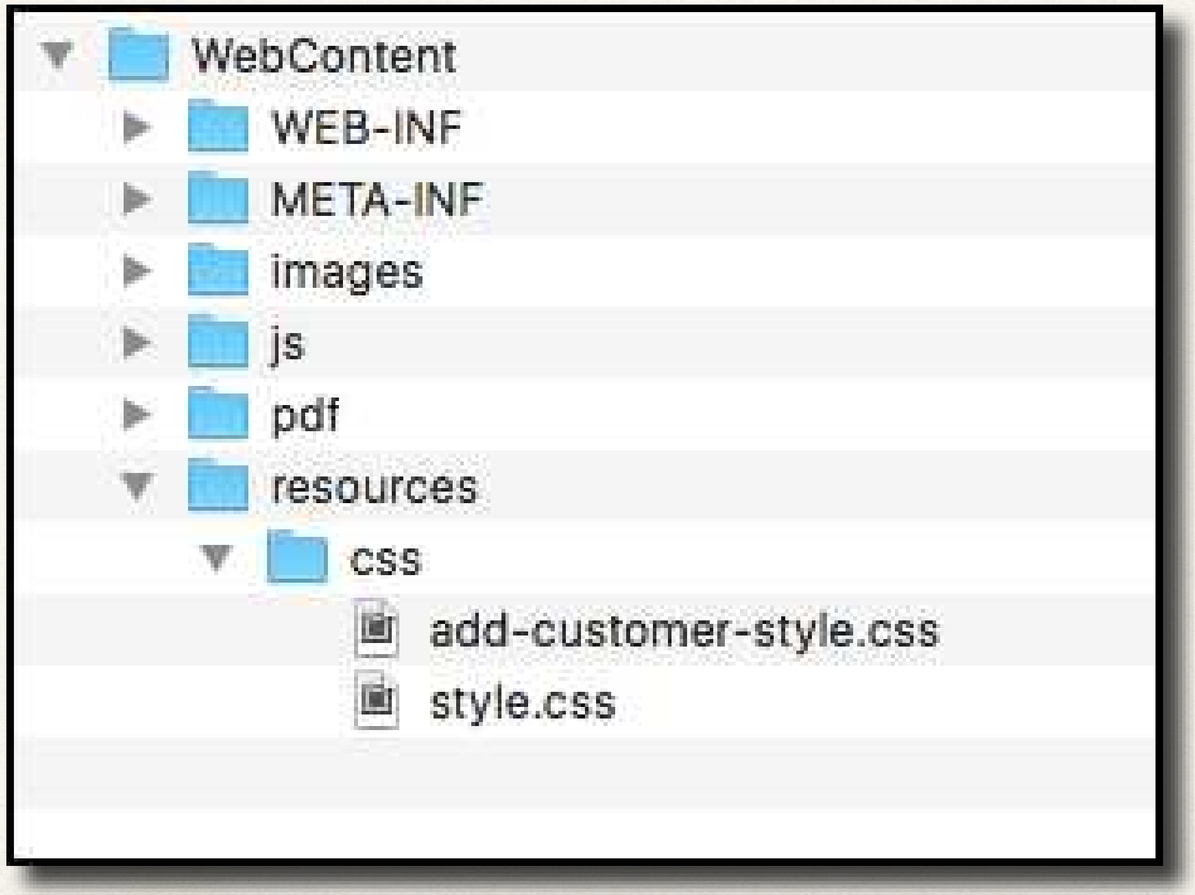
```
<head>
  <title>List Customers</title>

  <link type="text/css"
      rel="stylesheet"
      href="${pageContext.request.contextPath}/resources/css/style.css">

</head>
```



Applies for JavaScript, images, pdfs etc…



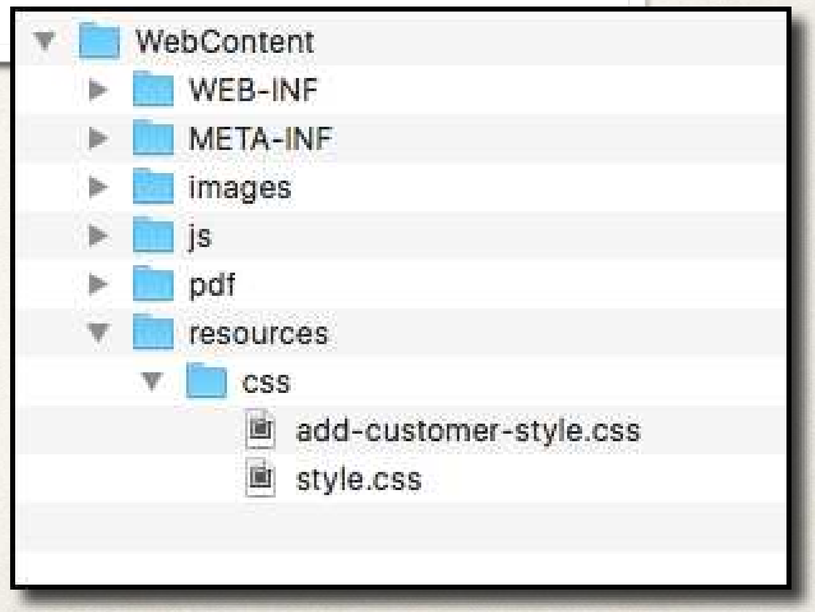
Alternate Directory Structure



Alternate Directory Structure

File: spring-mvc-crud-demo-servlet.xml

```
<mvc:resources location="/resources/" mapping="/resources/**" />
<mvc:resources location="/images/" mapping="/images/**" />
<mvc:resources location="/js/" mapping="/js/**" />
<mvc:resources location="/pdf/" mapping="/pdf/**" />
```



Adding a Welcome File

This is our “welcome page” ???

Welcome File

http://localhost:8080/web-customer-tracker

1. Server will look for a welcome file

2. If it doesn't find one, then you'll get 404 :-(

3. Welcome files are configured in **web.xml**

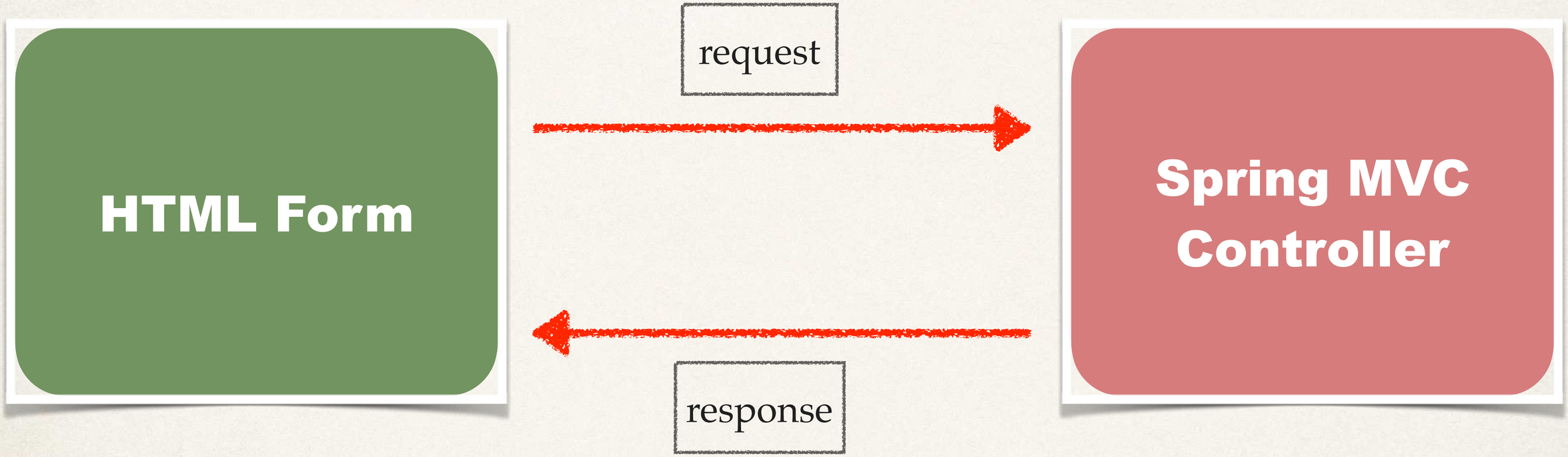
@GetMapping and @PostMapping

New Annotations

- @GetMapping
- @PostMapping

Added in Spring 4.3

HTTP Request / Response



Most Commonly Used HTTP Methods

Method	Description
GET	Requests data from given resource
POST	Submits data to given resource
others	…

Sending Data with GET method

```
<form action="processForm" method="GET" ...>
  ...
</form>
```

- Form data is added to end of URL as name/value pairs
  - **theUrl?field1=value1&field2=value2…**

Handling Form Submission

```
@RequestMapping("/processForm")
public String processForm(...) {
  ...
}
```

- This mapping handles **ALL HTTP** methods
  - GET, POST, etc …

Constrain the Request Mapping - GET

```
@RequestMapping(path="/processForm", method=RequestMethod.GET)
public String processForm(...) {
   ...
}
```

- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

New Annotation Short-Cut

```
@GetMapping("/processForm")
public String processForm(...) {
  ...
}
```

Added in Spring 4.3

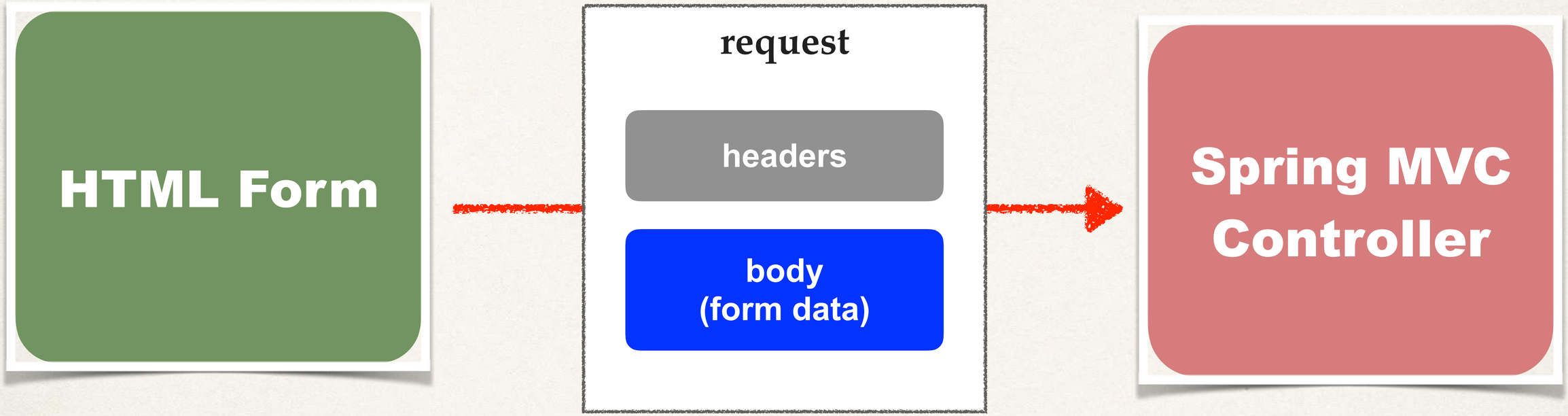
- New annotation: **@GetMapping**
- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

Sending Data with POST method

```
<form action="processForm" method="POST" ...>
  ...
</form>
```

- Form data is passed in the body of HTTP request message

Sending Data with POST method



Constrain the Request Mapping - POST

```
@RequestMapping(path="/processForm", method=RequestMethod.POST)
public String processForm(...) {
   ...
}
```

- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

New Annotation Short-Cut

```
@PostMapping("/processForm")
public String processForm(...) {
  ...
}
```

Added in Spring 4.3

- New annotation: **@PostMapping**
- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

Well …. which one???

GET

- Good for debugging
- Bookmark or email URL
- Limitations on data length

POST

- Can't bookmark or email URL
- No limitations on data length
- Can also send binary data

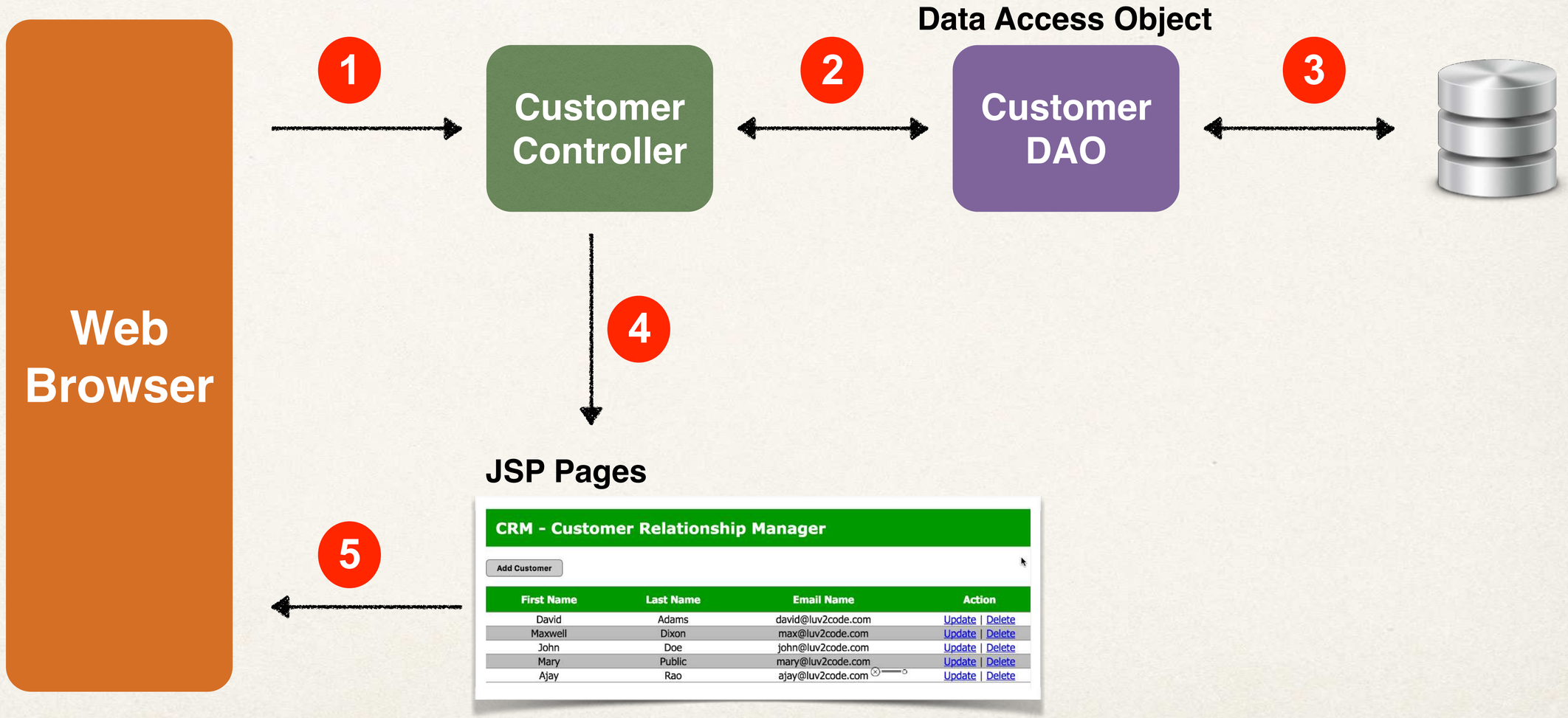
Recap: New Annotations

- @GetMapping
- @PostMapping

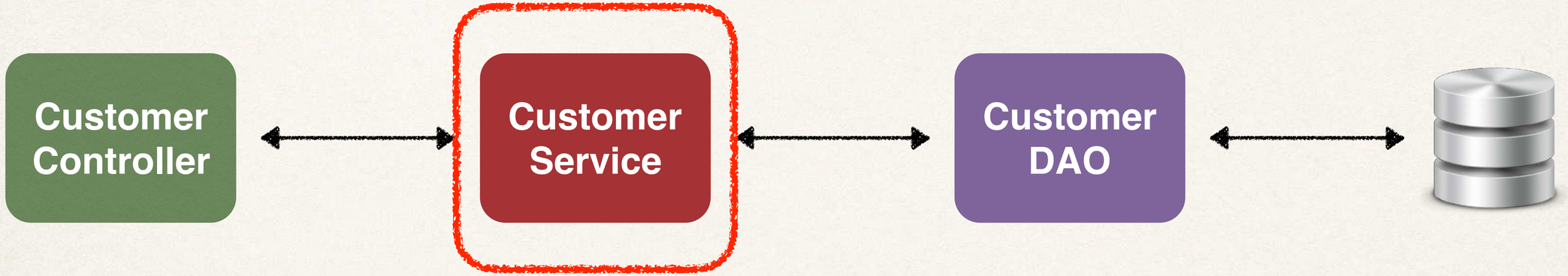
Added in Spring 4.3

Define Services with @Service

Big Picture

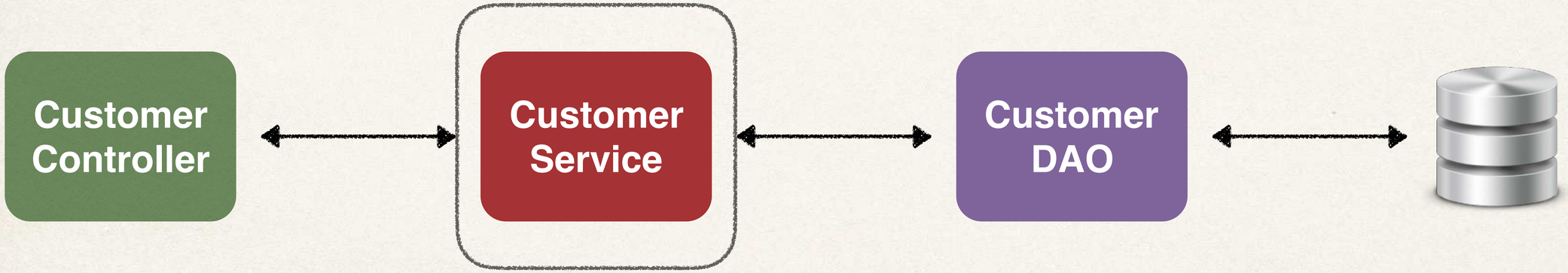

Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David Adams david@luv2code.com Update | Delete
Maxwell Dixon max@luv2code.com Update | Delete
John Doe john@luv2code.com Update | Delete
Mary Public mary@luv2code.com Update | Delete
Ajay Rao ajay@luv2code.com Update | Delete

Refactor: Add a Service Layer

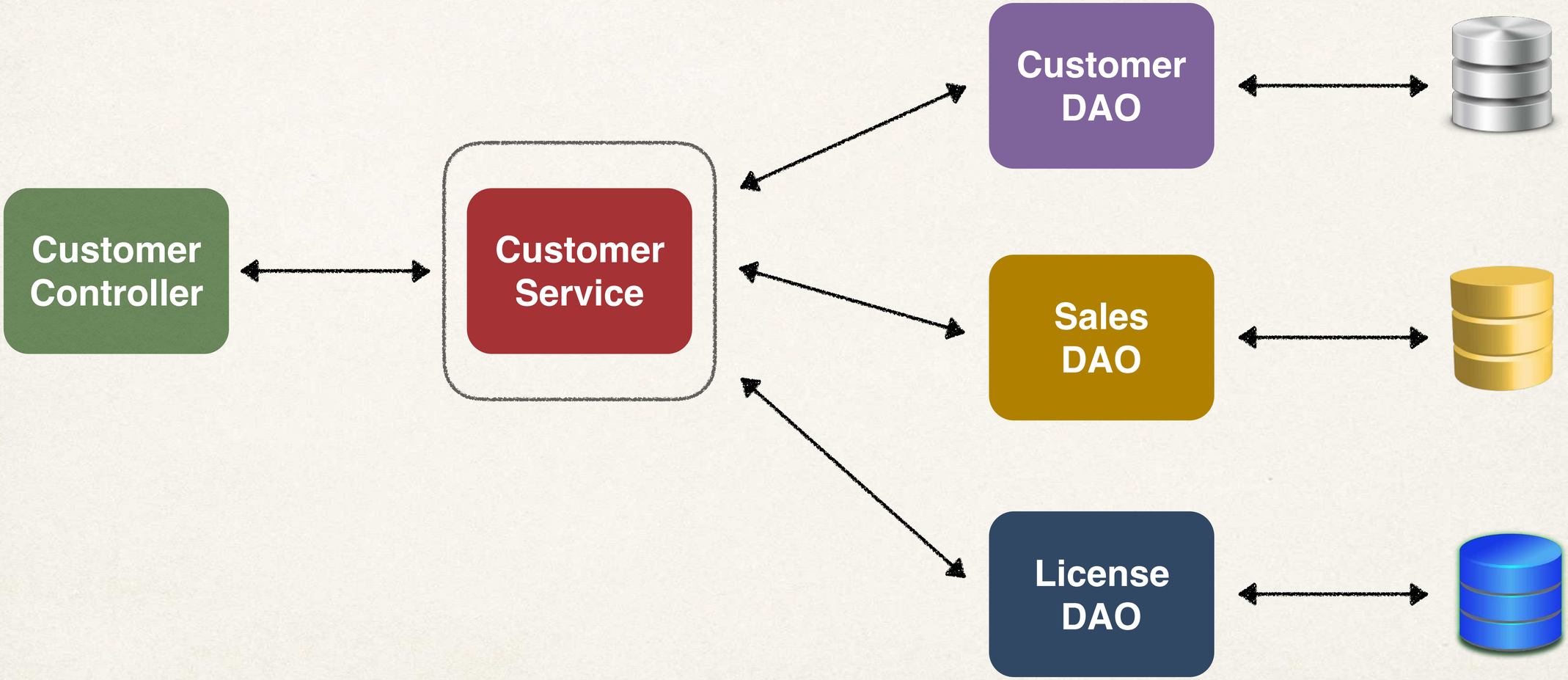


Purpose of Service Layer

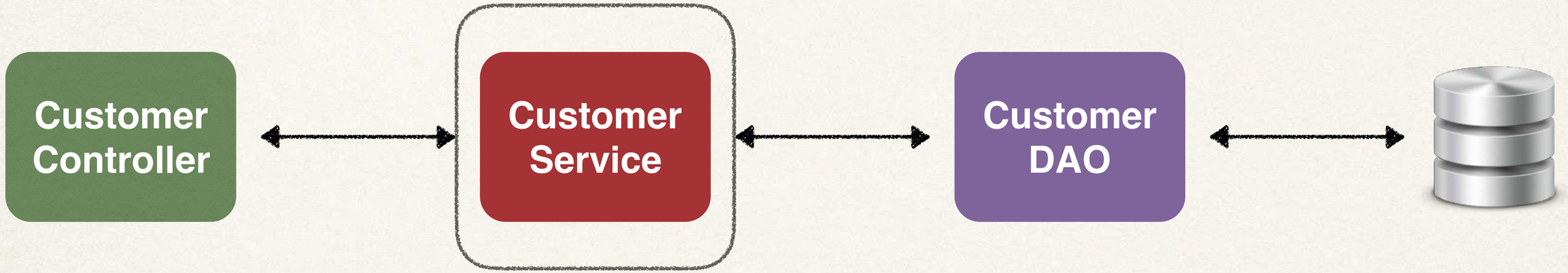
- **Service Facade** design pattern
- Intermediate layer for custom business logic
- Integrate data from multiple sources (DAO/repositories)



Integrate Multiple Data Sources

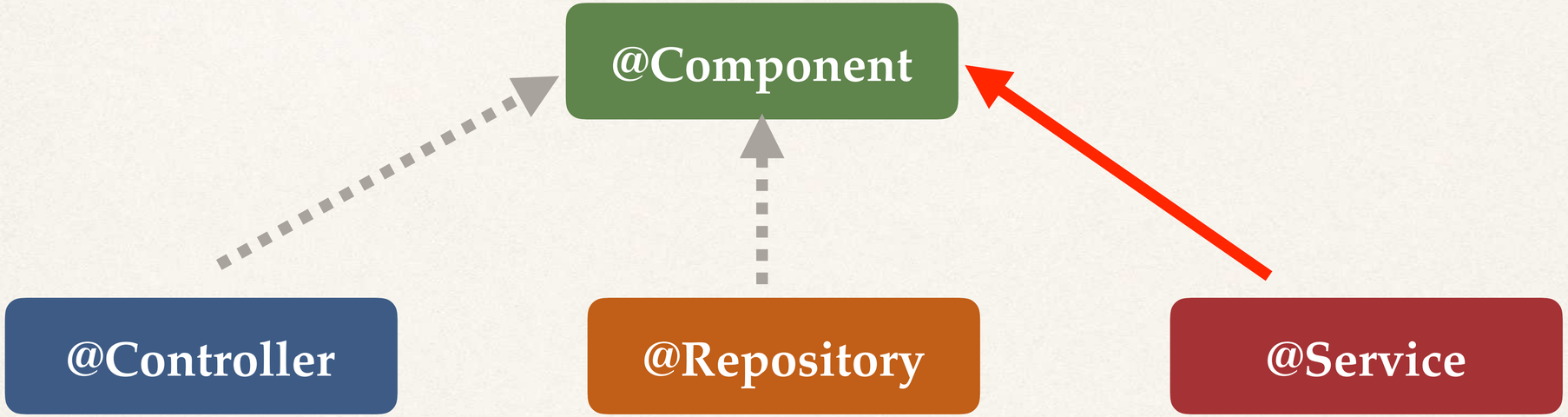


Most Times - Delegate Calls



Specialized Annotation for Services

- Spring provides the **@Service** annotation



Specialized Annotation for Services

- **@Service** applied to Service implementations

- Spring will automatically register the Service implementation
  - thanks to component-scanning

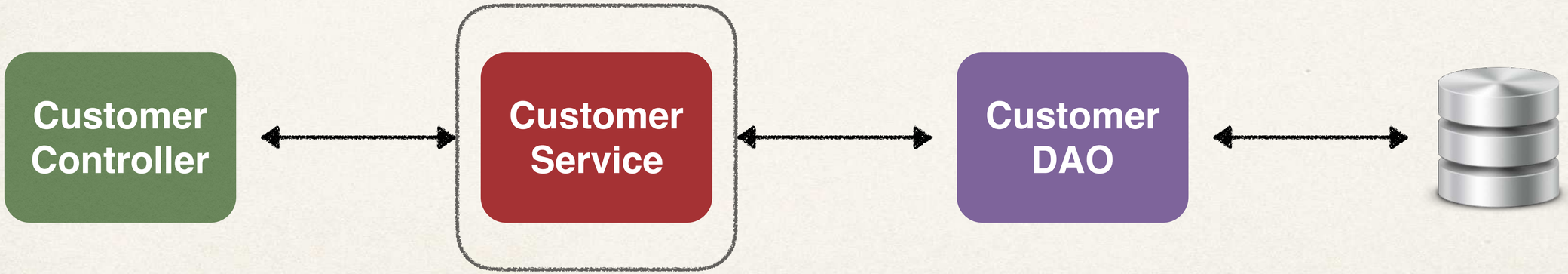
Customer Service

Step-By-Step

1. Define Service interface

2. Define Service implementation

   - Inject the CustomerDAO



Step 1: Define Service interface

```
public interface CustomerService {

   public List<Customer> getCustomers();

}
```

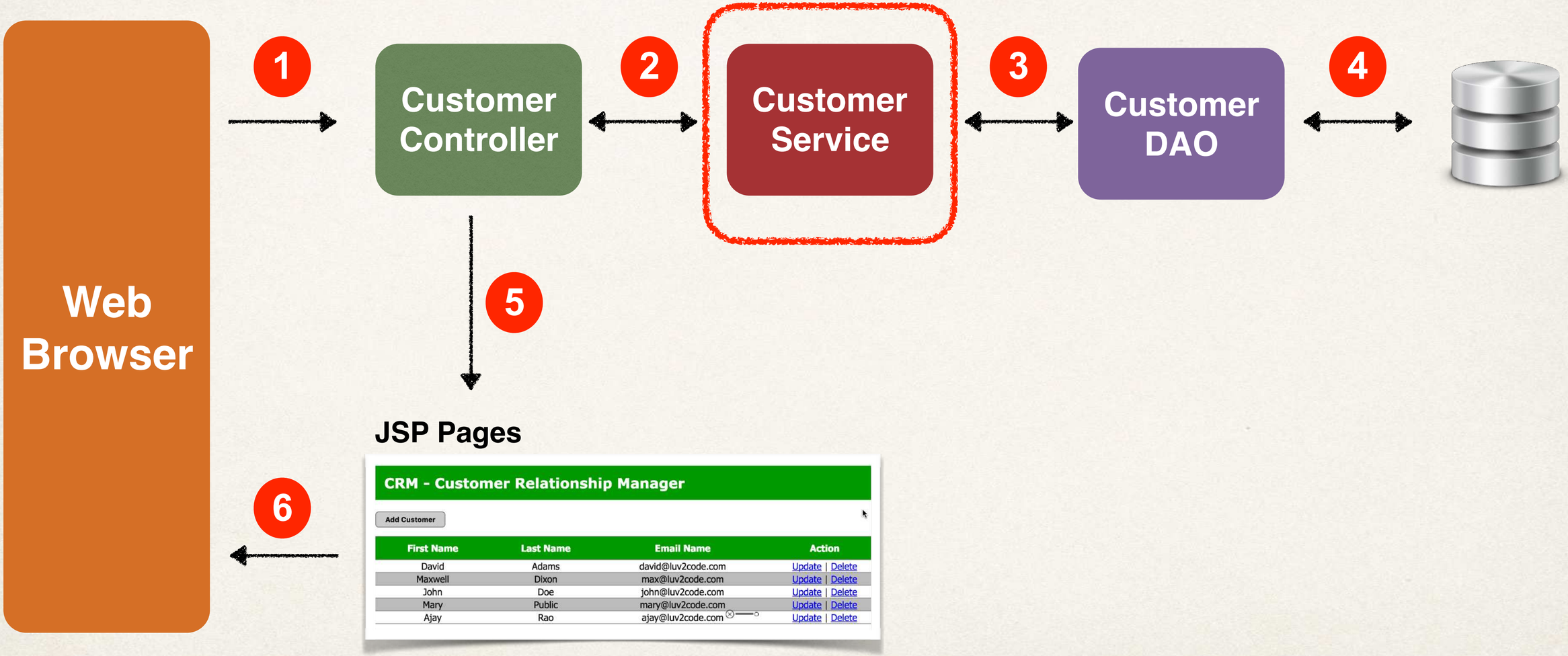
Step 2: Define Service implementation

```
@Service
public class CustomerServiceImpl implements CustomerService {

  @Autowired
  private CustomerDAO customerDAO;

  @Transactional
  public List<Customer> getCustomers() {
    ...
  }
}
```

Updates for the DAO implementation

```
@Repository
public class CustomerDAOImpl implements CustomerDAO {

  @Autowired
  private SessionFactory sessionFactory;

  public List<Customer> getCustomers() {
    ...
  }
}
```

Revised Big Picture

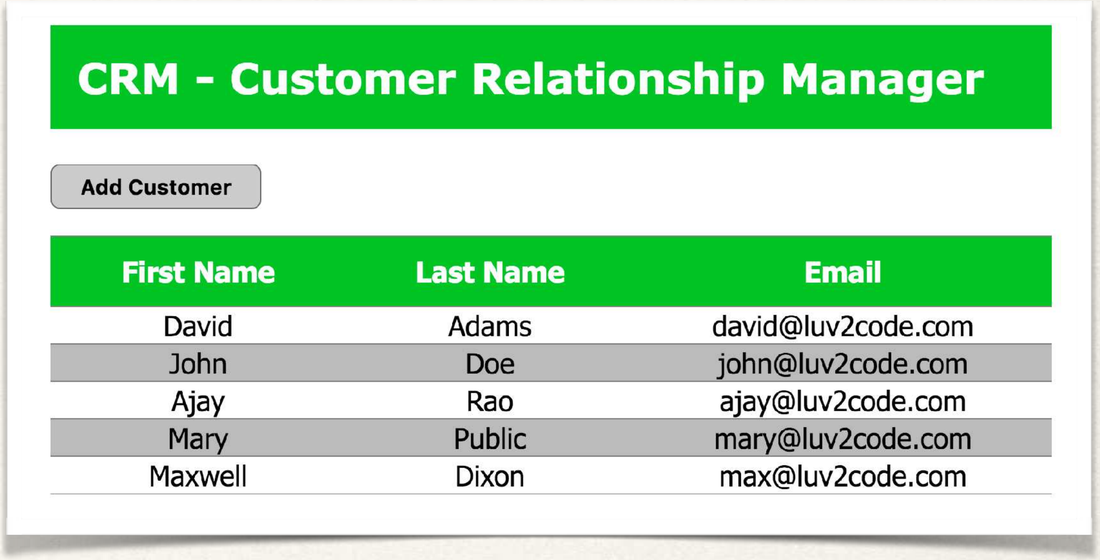


Add Customer

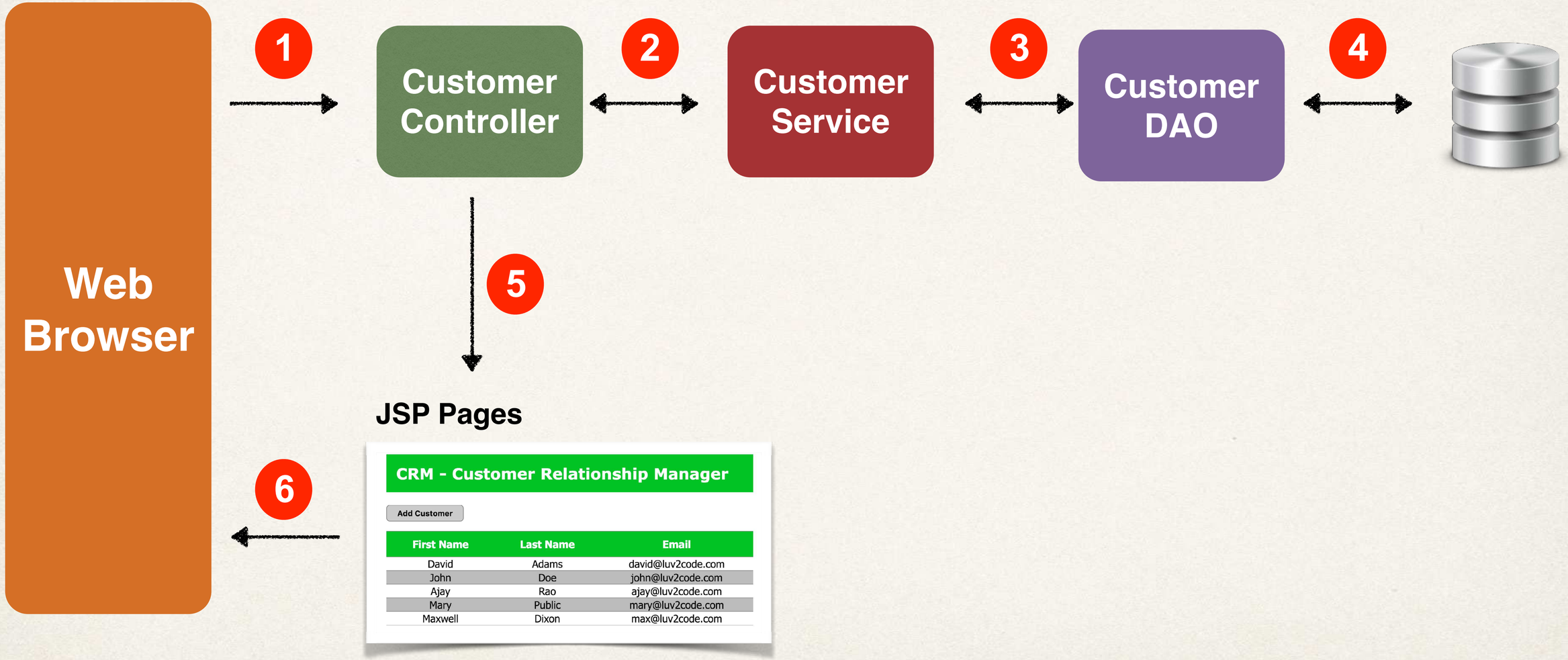
Add Customer

Step-By-Step

1. **Update list-customer.jsp**
    1. New “Add Customer” button
2. **Create HTML form for new customer**
3. **Process Form Data**
    1. Controller -> Service -> DAO



Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email
David
Adams
david@luv2code.com
John
Doe
john@luv2code.com
Ajay
Rao
ajay@luv2code.com
Mary
Public
mary@luv2code.com
Maxwell
Dixon
max@luv2code.com

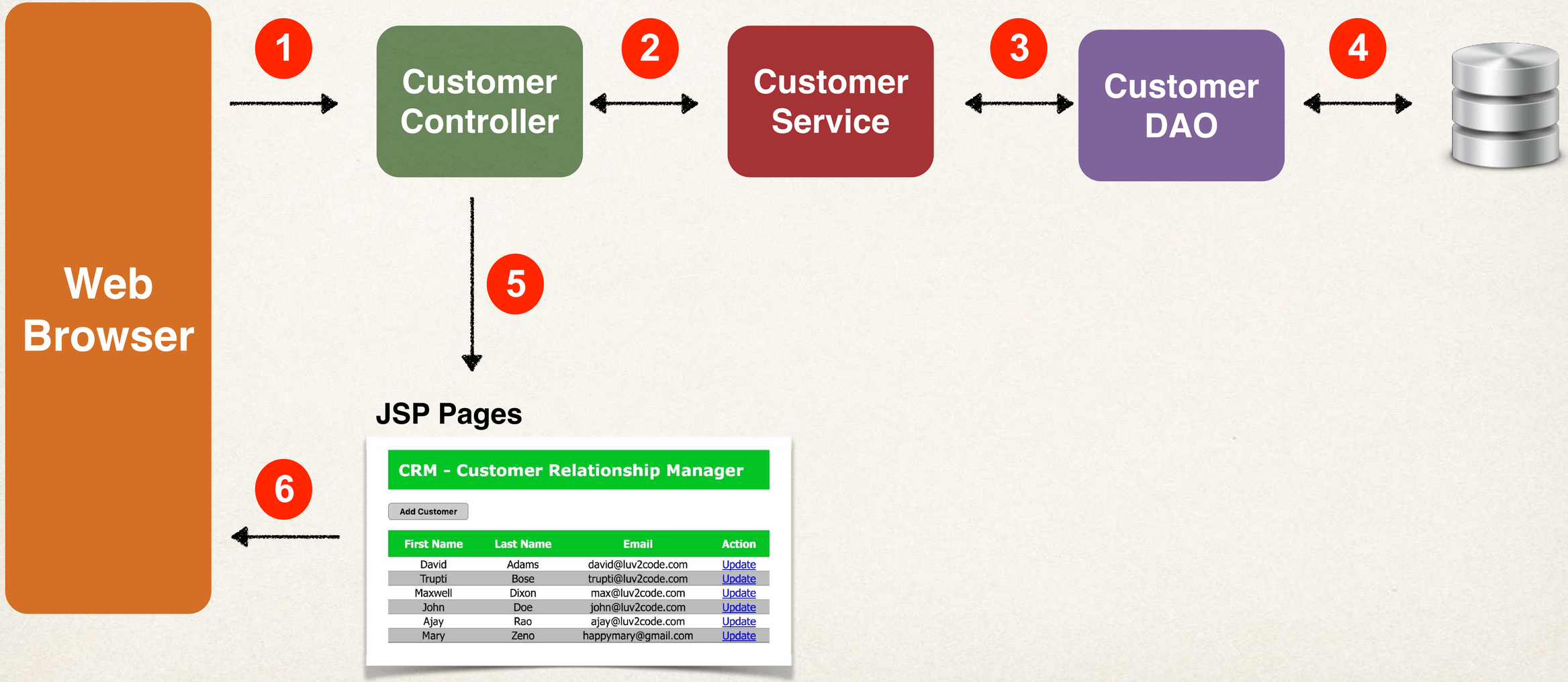
Add Customer

Step-By-Step

1. **Update list-customer.jsp** ✅
    1. New “Add Customer” button
2. **Create HTML form for new customer**
3. **Process Form Data**
    1. Controller -> Service -> DAO

Update Customer

Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name | Last Name | Email | Action
David | Adams | david@luv2code.com | Update
Trupti | Bose | trupti@luv2code.com | Update
Maxwell | Dixon | max@luv2code.com | Update
John | Doe | john@luv2code.com | Update
Ajay | Rao | ajay@luv2code.com | Update
Mary | Zeno | happymary@gmail.com | Update

Action Link - Update

CRM - Customer Relationship Manager

Add Customer

First Name	Last Name	Email	Action
David	Adams	david@luv2code.com	Update
Trupti	Bose	trupti@luv2code.com	Update
Maxwell	Dixon	max@luv2code.com	Update
John	Doe	john@luv2code.com	Update
Ajay	Rao	ajay@luv2code.com	Update
Mary	Zeno	happymary@gmail.com	Update

Each row has an **Update** link

- current customer id embedded in link

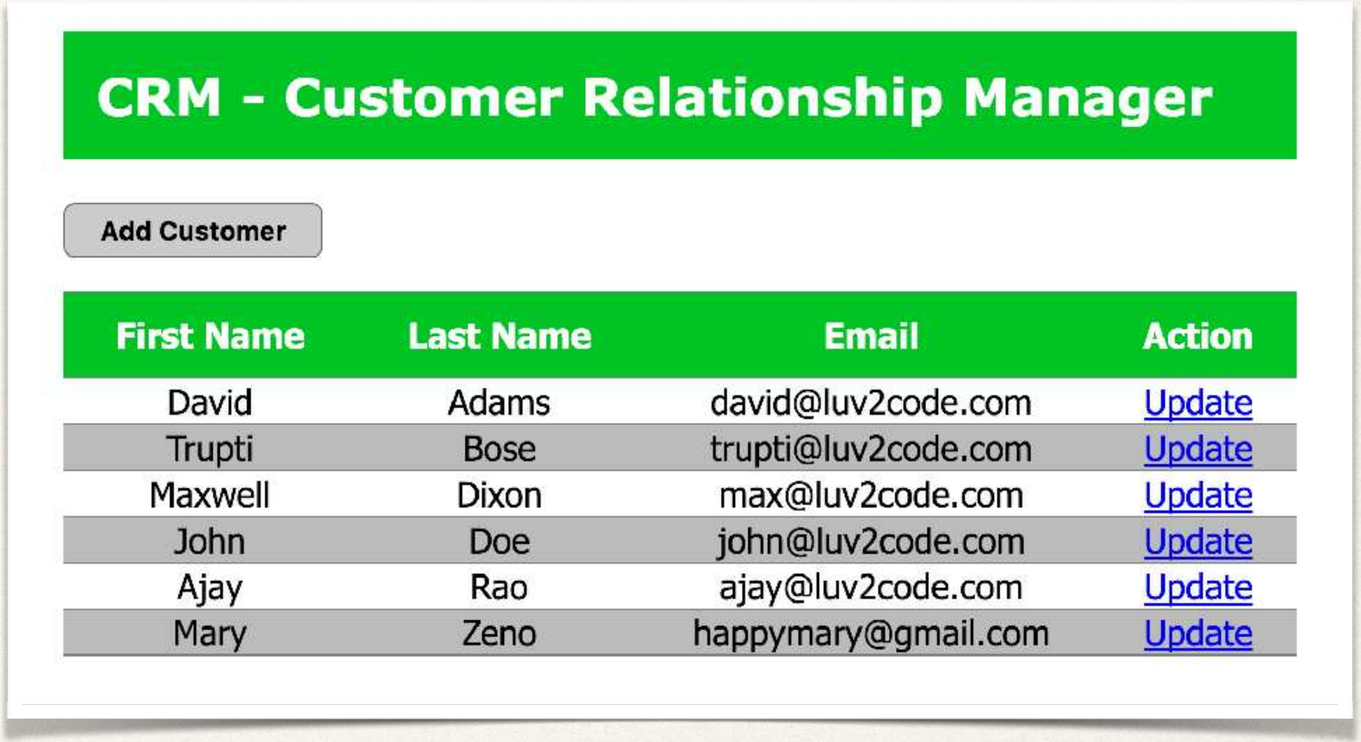
When **clicked**

- will load the customer from database
- prepopulate the form

Update Customer

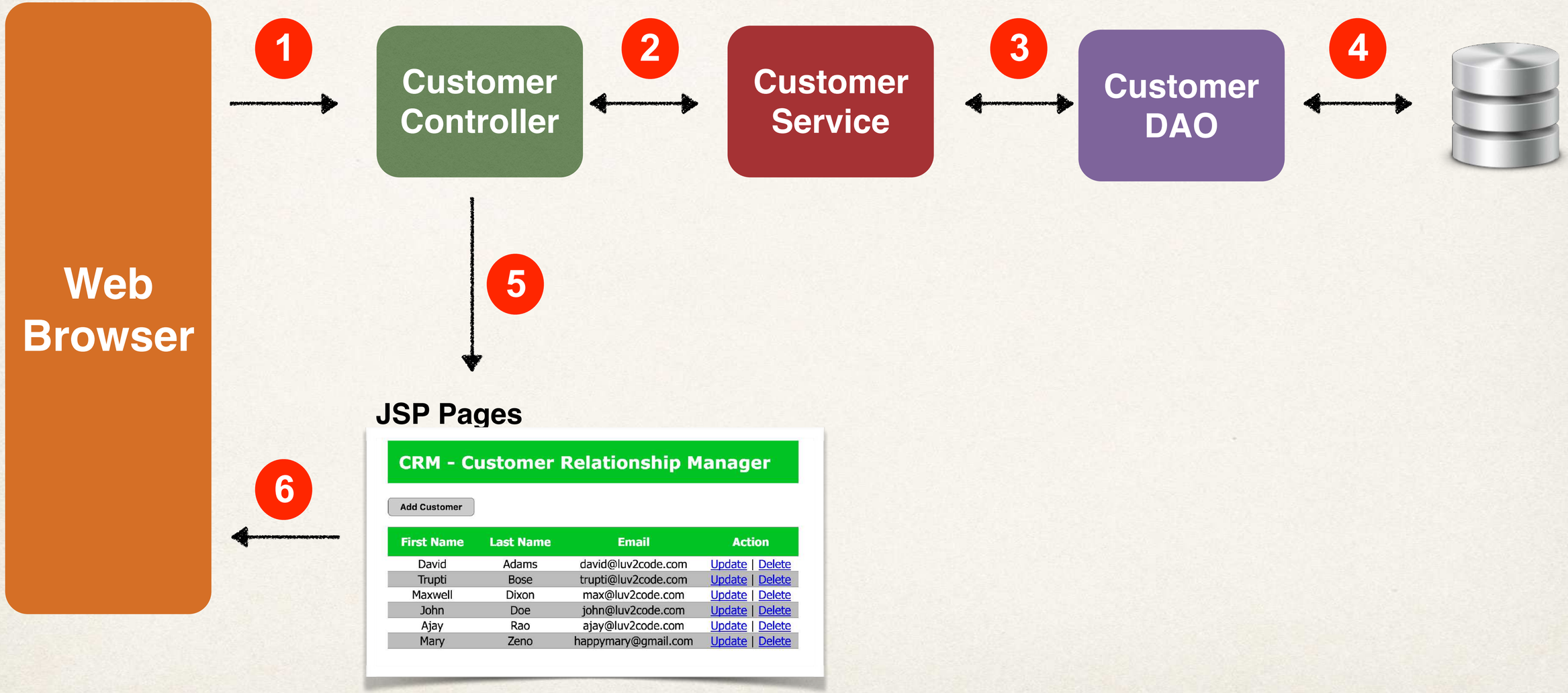
Step-By-Step

1. **Update list-customers.jsp**
    1. New "Update" link
2. **Create customer-form.jsp**
    1. Prepopulate the form
3. **Process form data**
    1. **Controller > Service > DAO**



Delete Customer

Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name | Last Name | Email | Action
David | Adams | david@luv2code.com | Update | Delete
Trupti | Bose | trupti@luv2code.com | Update | Delete
Maxwell | Dixon | max@luv2code.com | Update | Delete
John | Doe | john@luv2code.com | Update | Delete
Ajay | Rao | ajay@luv2code.com | Update | Delete
Mary | Zeno | happymary@gmail.com | Update | Delete

Action Link - Delete

CRM - Customer Relationship Manager

Add Customer

First Name	Last Name	Email	Action
David	Adams	david@luv2code.com	Update | Delete
Trupti	Bose	trupti@luv2code.com	Update | Delete
Maxwell	Dixon	max@luv2code.com	Update | Delete
John	Doe	john@luv2code.com	Update | Delete
Ajay	Rao	ajay@luv2code.com	Update | Delete
Mary	Zeno	happymary@gmail.com	Update | Delete

Each row has a **Delete** link

- current customer id embedded in link

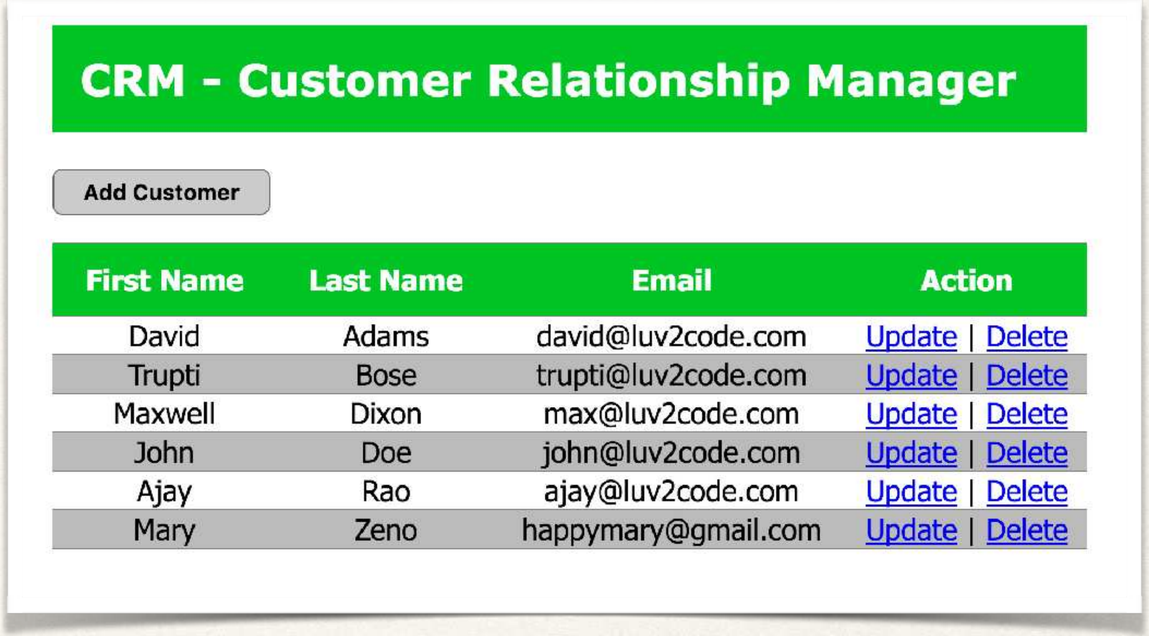
When **clicked**

- prompt user
- will delete the customer from database

Delete Customer

Step-By-Step

1. Add “Delete” link on JSP

2. Add code for “Delete”

   1. Controller > Service > DAO