

COMP S380F Lecture 6: MVC Model 1 & Model 2, Spring MVC Web Framework

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

- MVC design pattern
- MVC Model 1
- MVC Model 2 = MVC pattern
- Spring MVC web framework
 1. Front Controller: DispatcherServlet
 - Configuration in web.xml
 - [DispatcherServlet's name]-servlet.xml
 2. **Controller**: @Controller
 3. Handler Mapping: @RequestMapping
 4. View Resolver
 5. **View**: JSP View
 6. **Model**: ModelMap, ModelAndView

Java EE Design Patterns

- Java EE design patterns represent best practice design based on collective experience of working with Java EE projects.
- Design patterns aim to provide and document **a solution to a known, recurring problem** in building Java EE applications.

There are over 20 patterns (and growing). For example:

- **Model View Controller**: The J2EE BluePrints recommended architectural design pattern for interactive applications.
- **Front Controller (Command)**: Providing a central dispatch point to handle all incoming requests.
- **Data Access Object**: Typical pattern for data access layer (linking the data storage layer with the application)

Web Application Architecture: 3 logical layers

A web application architecture can be separated into 3 logical layers:

1. Presentation layer:

- What the user sees or interacts with.
- E.g., web pages, various visual objects, interactive objects, or reports.

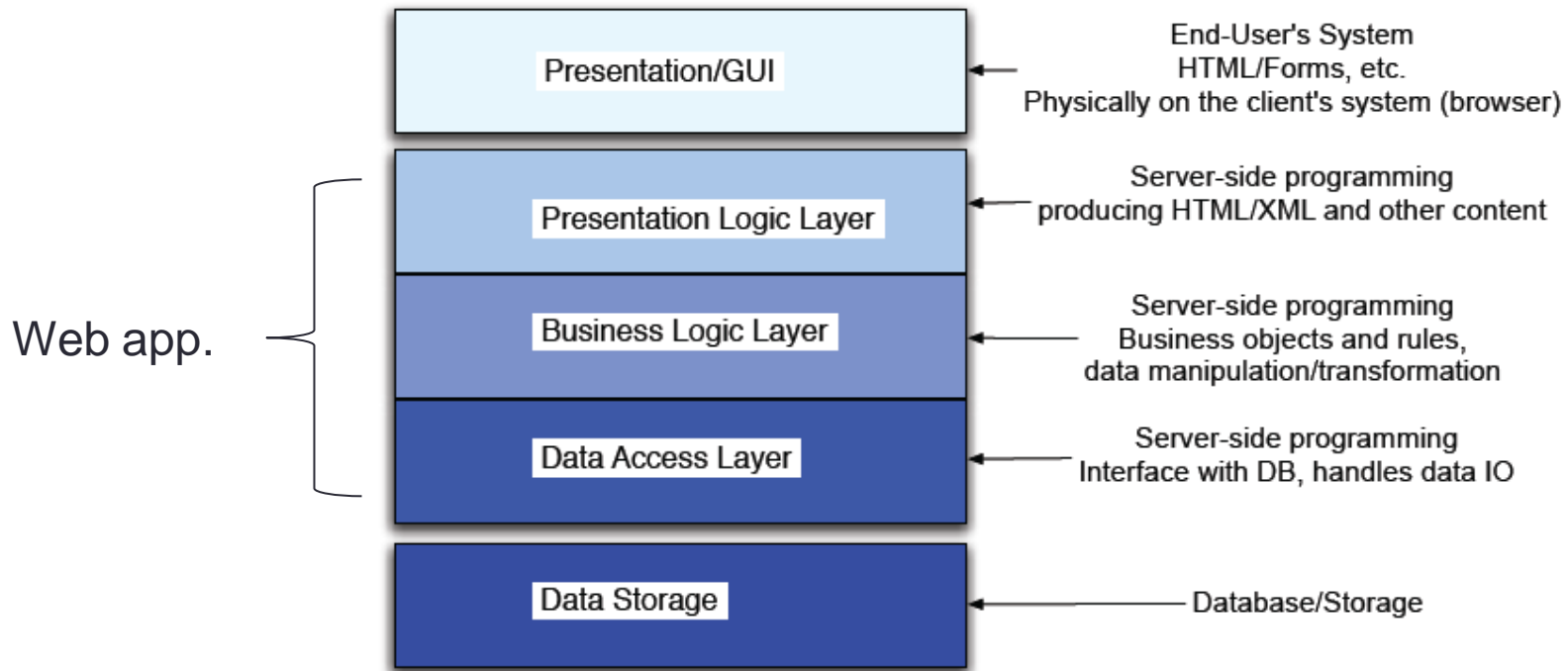
2. Business logic layer:

- The business rules enforced via programming logic (e.g., Java classes).

3. Data access layer:

- Definitions of database tables and columns and the computer logic that is needed to navigate the database.
- Enforce rules regarding the storage and access of information. E.g., dates must be valid dates.
- **MVC (Model View Controller) pattern** is designed to apply this logical separation of layers into system implementation.

Web application layers



- **Presentation Layer**
 - JSP, HTML, CSS
- **Business Logic**
 - Java classes
- **Data Access Layer**
 - Data Access Objects
- **Data Store**
 - RDBMS (Relational DBMS), NoSQL databases

The MVC design pattern

MVC stands for Model View Controller, which are the name of its components:

Model

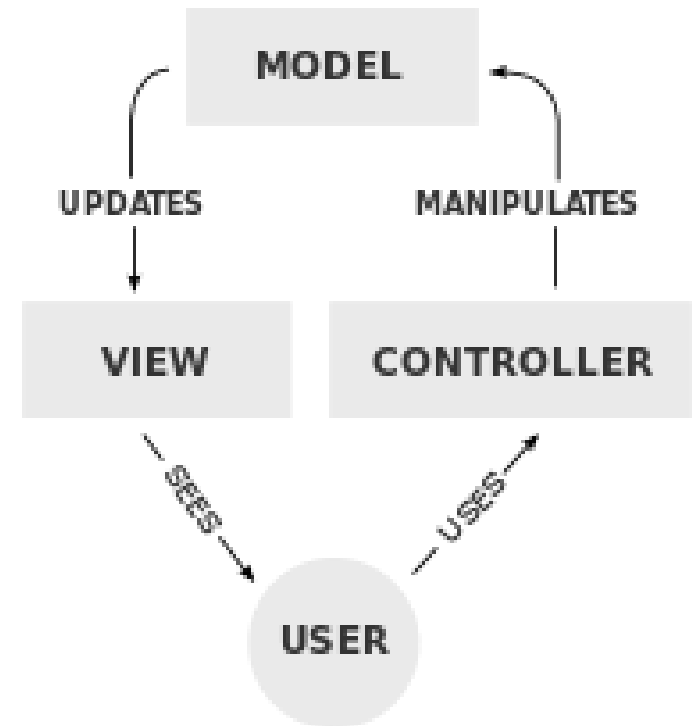
- Represents the data and rules that govern access and update of the data.
- ~ Data access layer

View

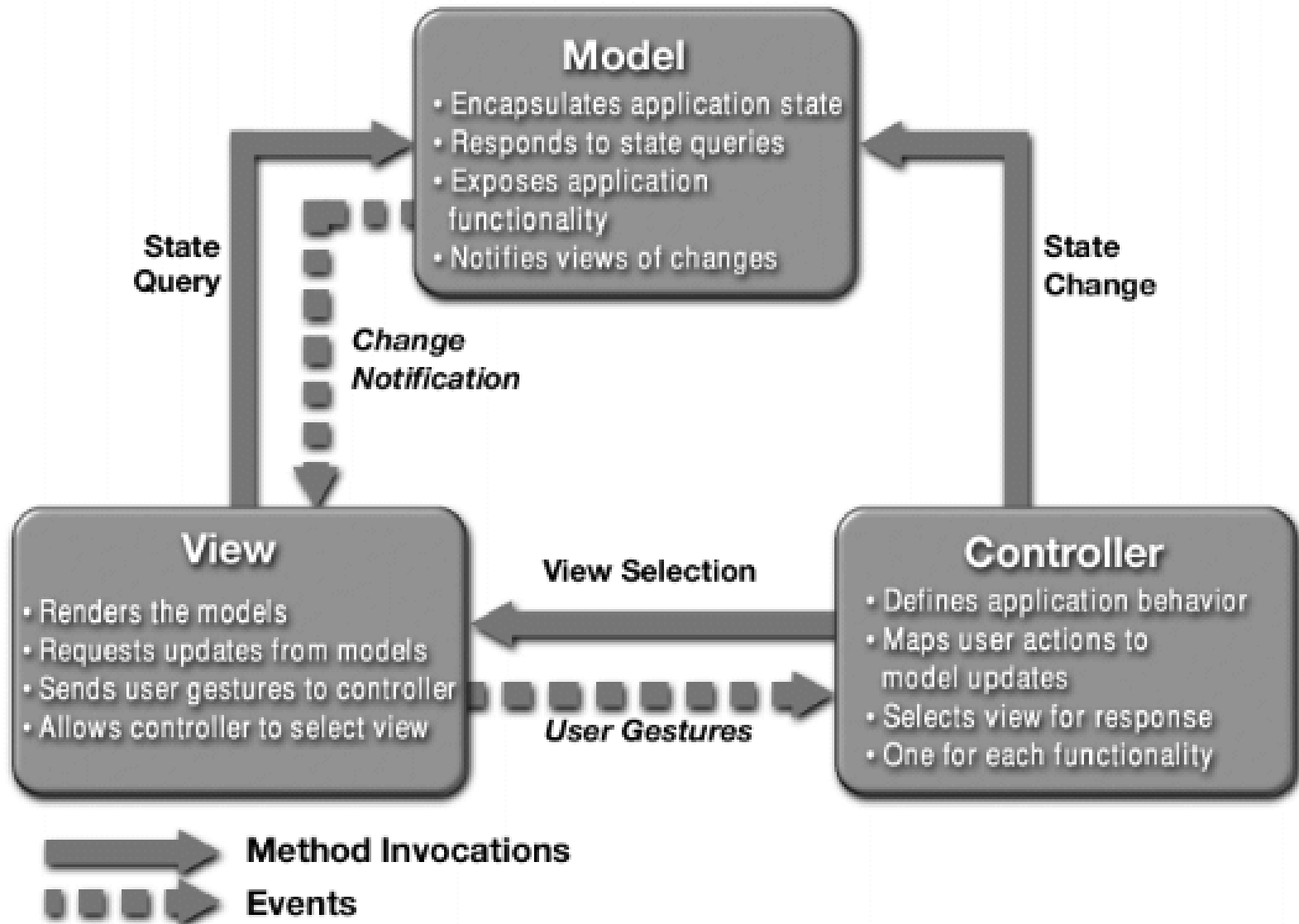
- Render the contents of a model data, and specifies how it is presented.
- ~ Presentation layer

Controller

- Translate the user's interactions with the view and model data into actions.
- ~ Business logic layer

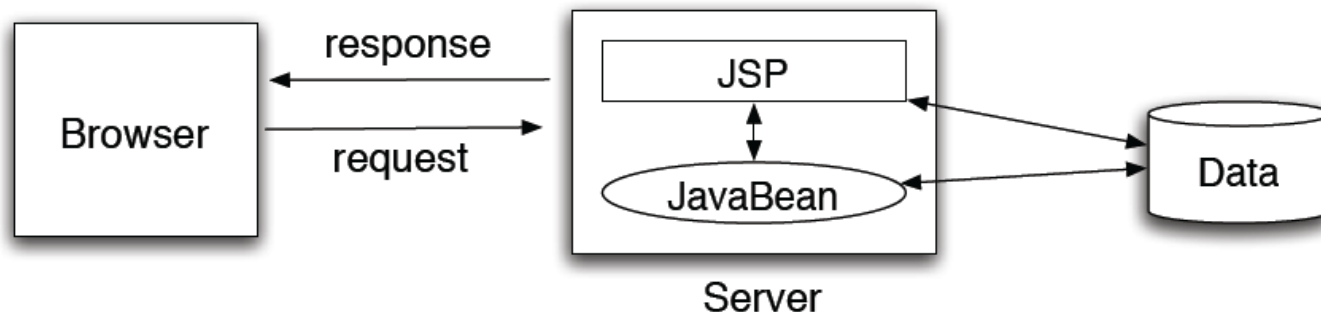


MVC component roles



MVC Model 1

- **MVC Model 1** was the first-generation approach that used JSP pages and the JavaBeans component architecture to implement the MVC architecture for the Web:
 - HTTP requests are sent to JSP pages.
 - The JSP pages implement the business logic and calls out to the model (JavaBeans) for data to update the view.
 - The next view to display (e.g., JSP page, servlet, HTML page) is determined either by hyperlinks selected in the source document or by request parameters.



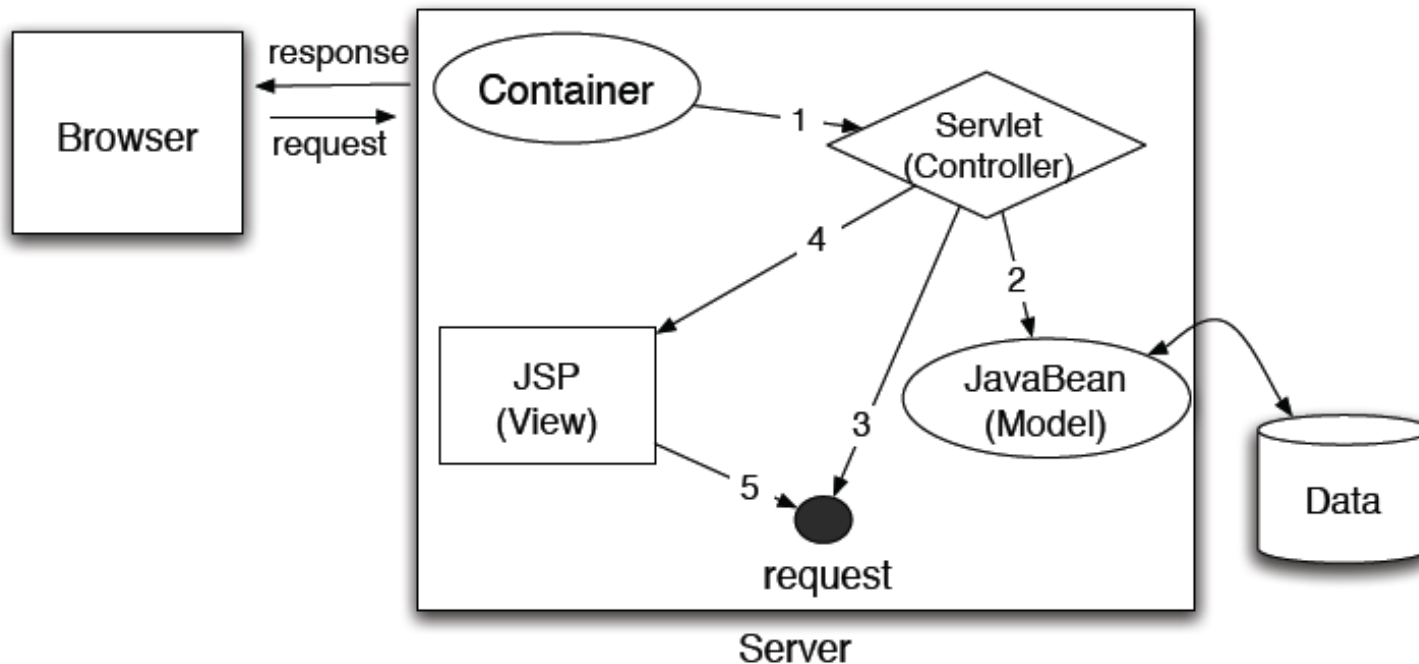
MVC Model 2

- **MVC Model 2** is a term invented by Sun Microsystems Inc.:
 - Introduce a **controller servlet** between the browser and the JSP / Servlet content being delivered.
 - The controller servlet centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state.
 - The controller also handles view selection, which decouples JSP pages and Servlets from one another.



MVC Model 1 and Model 2 simply refer to the absence or presence of a controller servlet, which dispatches request from the client tier and selects views.

Model 2 Architecture = MVC pattern

- The controller servlet (dispatcher) is the single entry point of the application.
- Presentation parts (JSP pages) are isolated from each other.
- MVC separates content generation and content presentation.



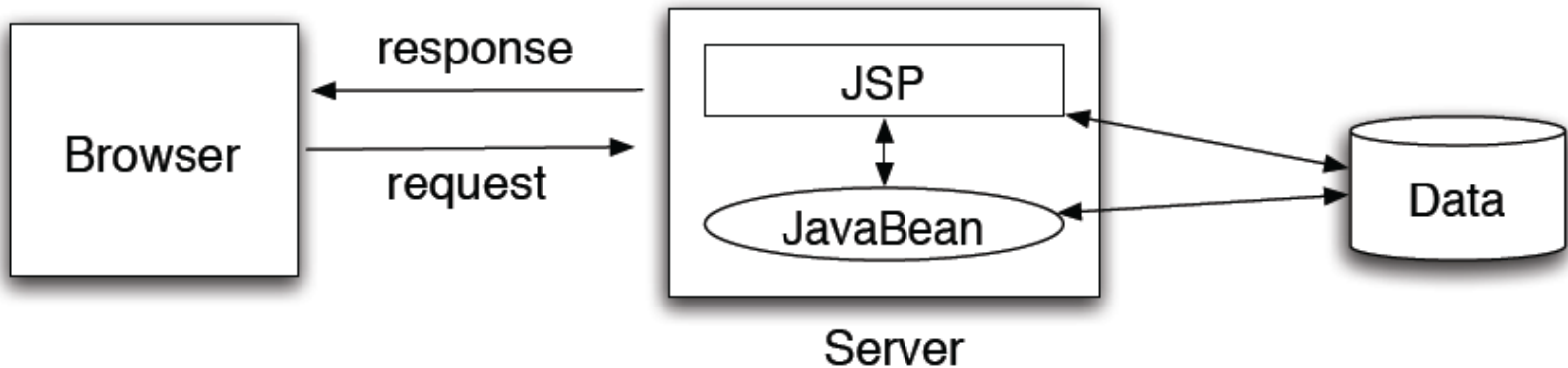
MVC Model 1 vs. MVC Model 2

	MVC Model 1	MVC Model 2
	<ul style="list-style-type: none">• Easy and quick for development.• Suitable for small projects that have simple page flow, little need for centralized security control or logging, and change little over time.	<ul style="list-style-type: none">• Views do not reference each other directly.• More flexible and easier to maintain and to extend.
	<ul style="list-style-type: none">• Controllers and views are mixed together.• Hard to achieve division of labor between page designer (presentation) and web developer (business logic).	<ul style="list-style-type: none">• Much more heavyweight on design and development than MVC Model 1.• Not suitable for small and static applications.

MVC Model 1 Example

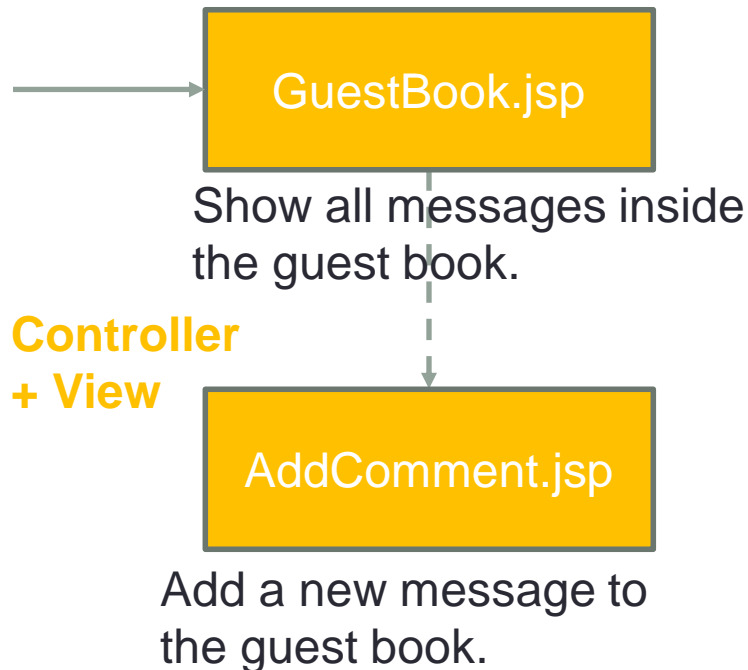
No distinct separation of presentation / business logic layers:

- The application consists of a series of JSP pages.
- The user is expected to proceed from the first page to the next.
- There may be JavaBeans (or POJOs) performing business operations.
- But each JSP page contains logic for processing its own output and maintaining application flow.



MVC Model 1 Example: Guest Book Application

Web app example: guestbook_mvc1



Guest Book

- President (2022-03-17 21:32:33):
Welcome to HKMU!
- Keith (2022-03-17 21:33:19):
Computer Science is fun.

[Add Comment](#)

Model

GuestBookBean.java

The guest book is an
`CopyOnWriteArrayList<GuestBookEntry>`

GuestBookEntry.java

A message contains the user name,
the message, and the date.

Add Guest Book Comment

Name:

Message:

Centralized Controller

Even with MVC pattern, maintaining Web application can be difficult:

- Maintaining links between different pages (views)
- Duplication of functionality (e.g., validation, authentication) at different places
- Maintaining access to Data Sources
- Updating layouts

Solution: Have a central (single) controller

- Centralize functions such as authentication, validation, etc.
- Maintain central database of page templates
- View selection is in one place
- Updating links requires updating in just one place.

Multiple Controllers

However, using a centralized controller has its disadvantages

- Centralized controllers can become heavy, with too much logic in a single class file.
- Multiple developers working on the Controller simultaneously.
- Testing becomes difficult.
- Certain sections of the web application may require different implementations of the same features (e.g., admin login vs. normal user login).

Solution:

- Some frameworks solve this by having a controller per logic unit (or feature group).
 - E.g., there may be a controller devoted to the admin functionality and covers all the views for an admin user.

Model 2 Example: Guest Book Application

1. Browser sends a request to controller.
2. Controller processes the request, updates some data.
3. Controller forwards the request and data to view.

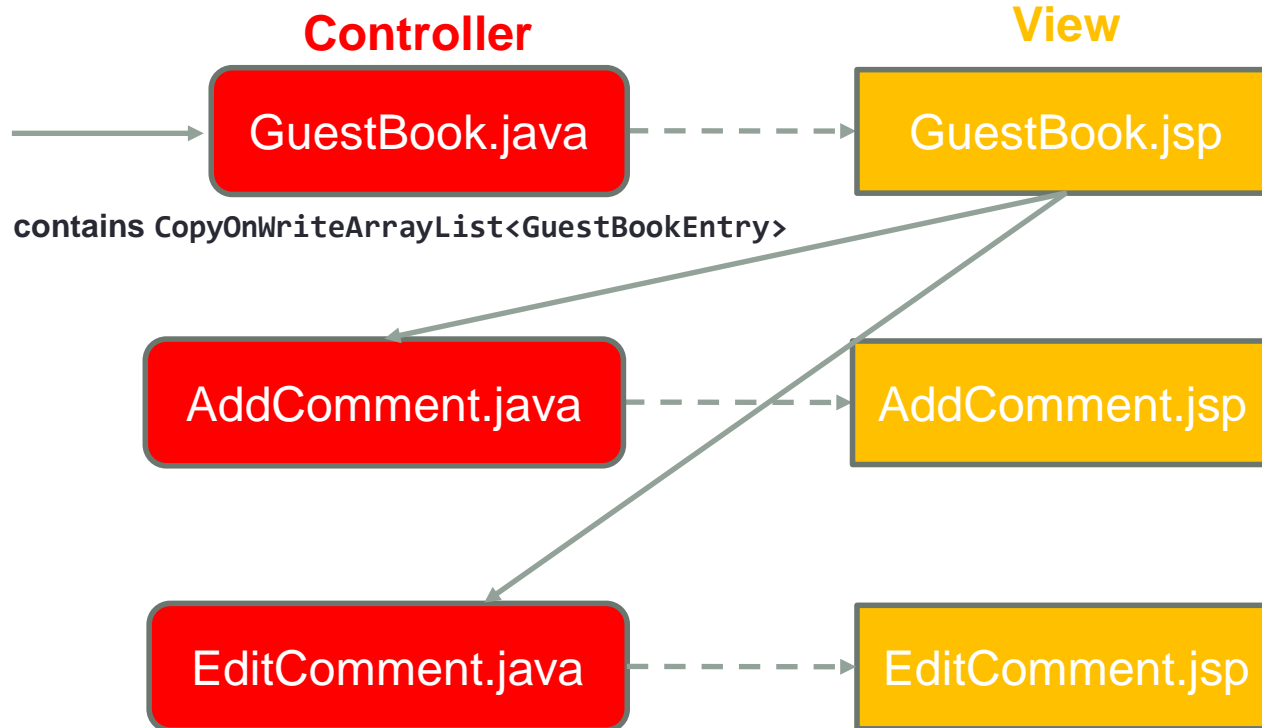


```
request.getRequestDispatcher("path_to_jsp")  
    .forward(request, response);
```

4. View generates the response that is sent back to the client.

MVC Model 2 Example: Guest Book Application

Web app example: guestbook_mvc2



Guest Book

- President (2022-03-17): [\[Edit\]](#)
Welcome to HKMU!

[Add Comment](#)

Edit Comment

Name:

Model

GuestBookEntry.java

- One operation, one controller
- Requests always go to controllers first
- JSP pages are **hidden** under /WEB-INF/

A message also contains ID now.

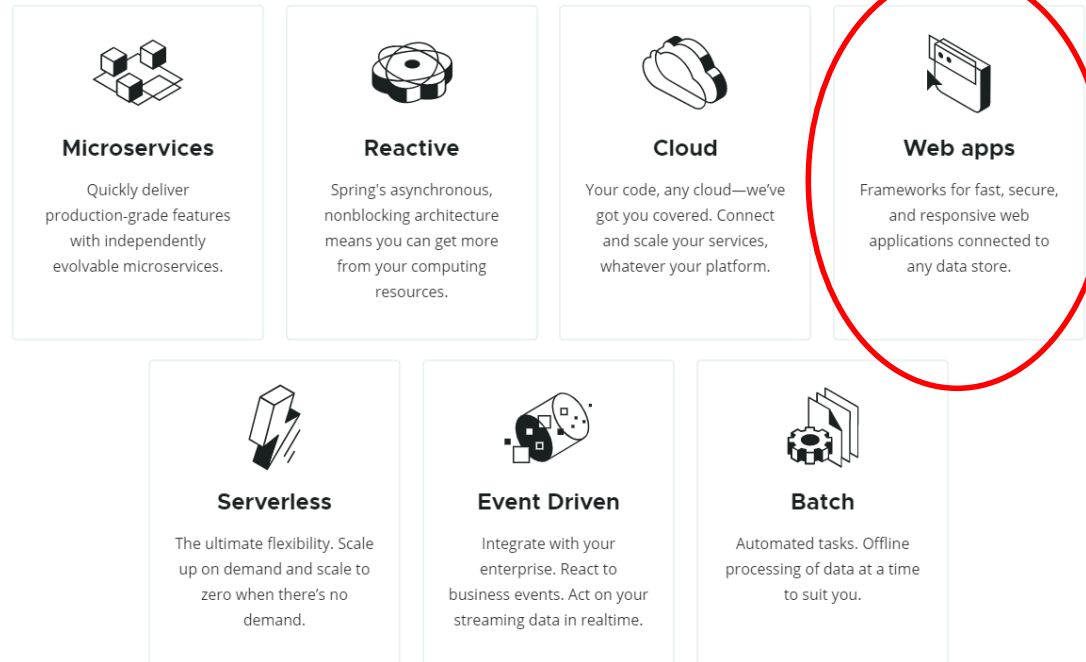
Web application development: The Servlet way

1. JSP or HTML Form
 2. Submit to servlet
 3. Servlet processes data & show information output
- Work well for small applications but can quickly grow out of control, because HTML, JSP scriptlets, JavaScript, JSP tag libraries, database access, and business logic make it difficult to organize
 - Lack of structure can cause a “soup” of different technologies

Web application development: A better way ?

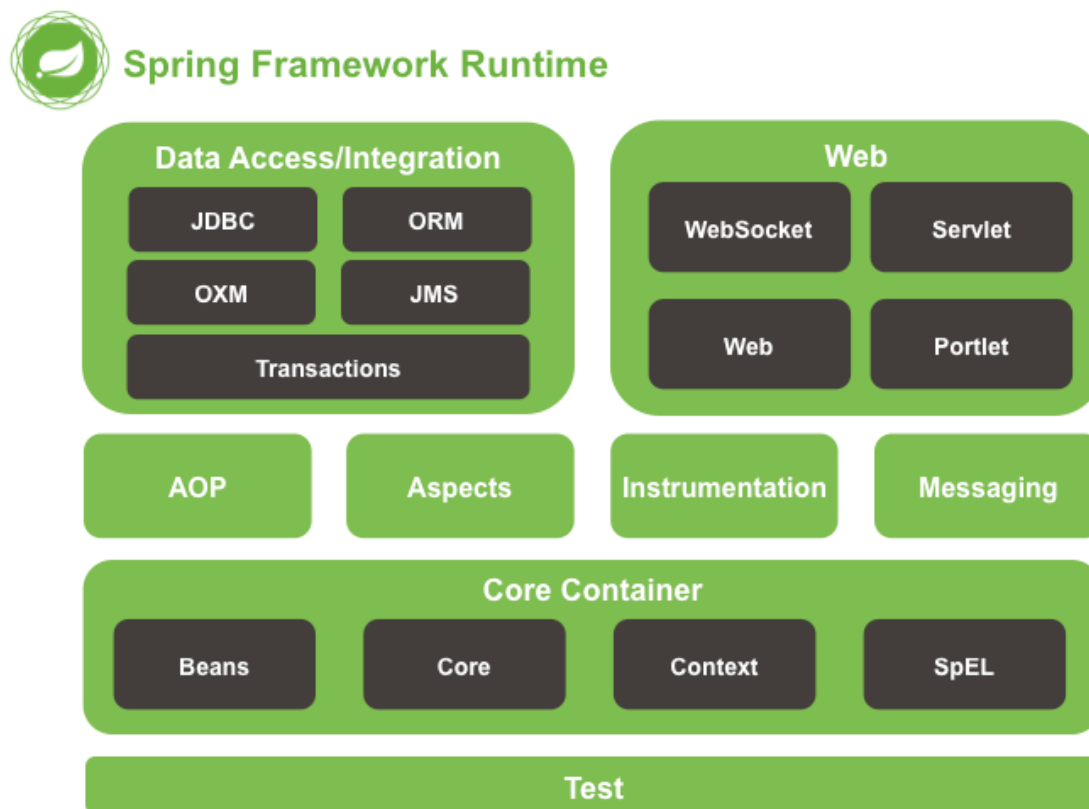
- Separate the data access, business logic and presentation using a MVC Framework
- **Spring**: “newer” of the frameworks
 - Integrates well with other frameworks
 - Current version:
Spring Framework 5

What Spring can do



Modules of the Spring Framework

The Spring Framework consists of features organized into about 20 modules



<https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html>

Modules of the Spring Framework (cont')

Core Container

- Spring bean factory
- Inversion of Control, Dependency Injection
- Spring application context

Data Access / Integration

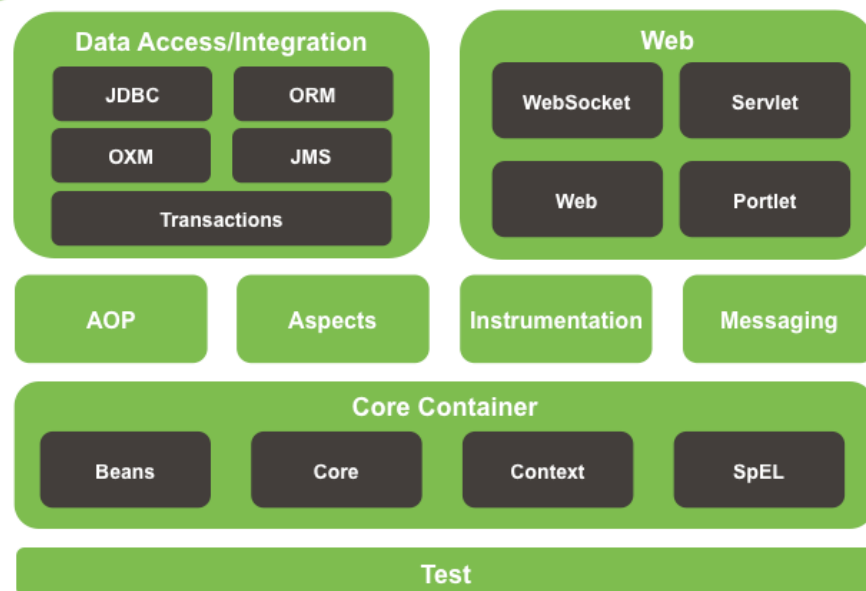
- JDBC, ORM, OXM, JMS and Transaction

Web

- **Spring MVC web framework** for developing web applications



Spring Framework Runtime

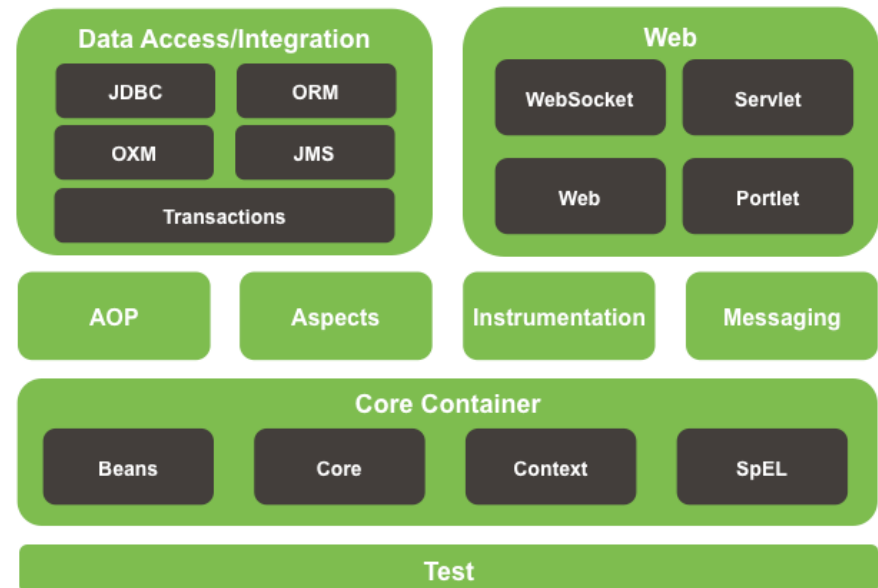


Modules of the Spring Framework (cont')

- AOP (Aspect-Oriented Programming)
- Remote Access
- Authentication and Authorization
- Remote Management
- Messaging Framework
- Web Services
- Email
- Testing



Spring Framework Runtime



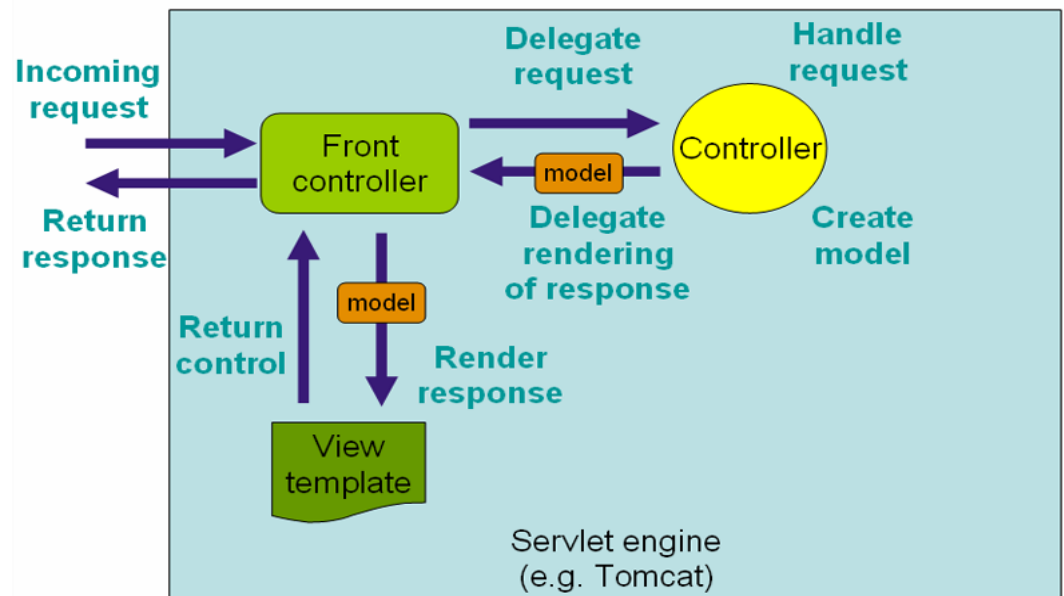
Spring is very loosely coupled; components are widely reusable and separately packaged.

What is Spring MVC?

- MVC Web Framework
- Developed by the Spring team in response to what they felt were deficiencies in frameworks like Struts
- Deeply integrated with Spring
- Allows most parts to be customized (e.g., you can use pretty much any view technology)
- RESTful Web Services
 - Based on JAX like method

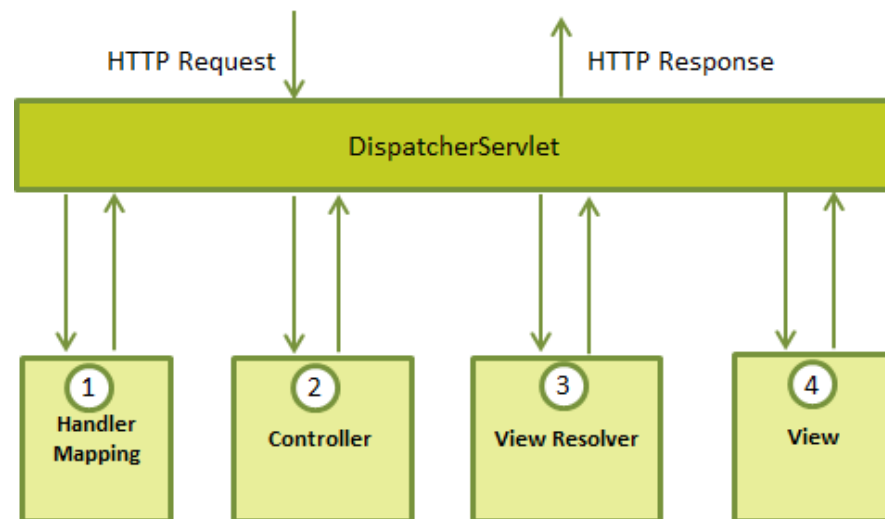
Spring MVC Features

- Clear separation of roles
- Simple, powerful **annotation-based configuration**
- Controllers are configured via Spring, which makes them easy to use with other Spring objects and makes them easy to test
- Customizable data binding
- Flexible view technology
- Customizable handler mapping and view resolution

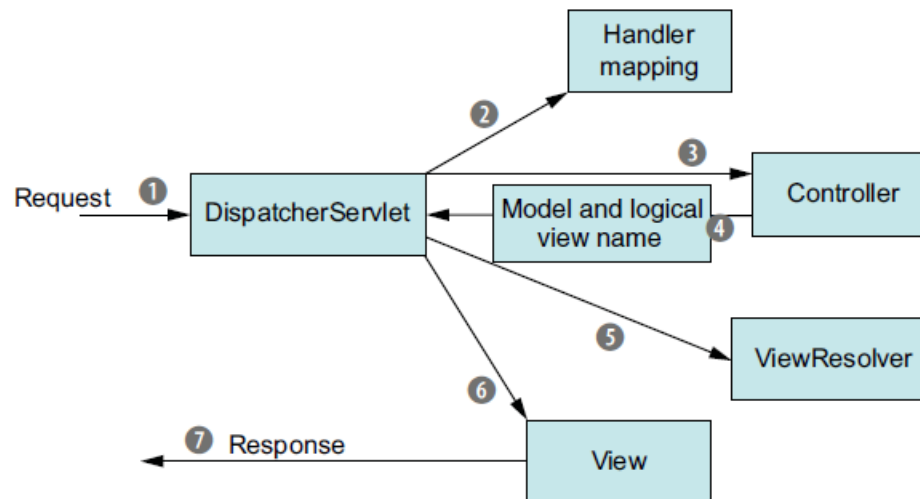


Front Controller: DispatcherServlet

- A **front controller** is a web app design pattern where a **single servlet** delegates responsibility for a request to other components of an application to perform actual processing.
- Spring uses a **DispatcherServlet** as the front controller:
 - Defined in the web.xml file to analyze a request URL pattern
 - Pass control to the correct **Controller** by using a URL mapping defined in a **Spring configuration XML file**
- Spring MVC is designed around **the DispatcherServlet that handles all the HTTP requests and responses.**

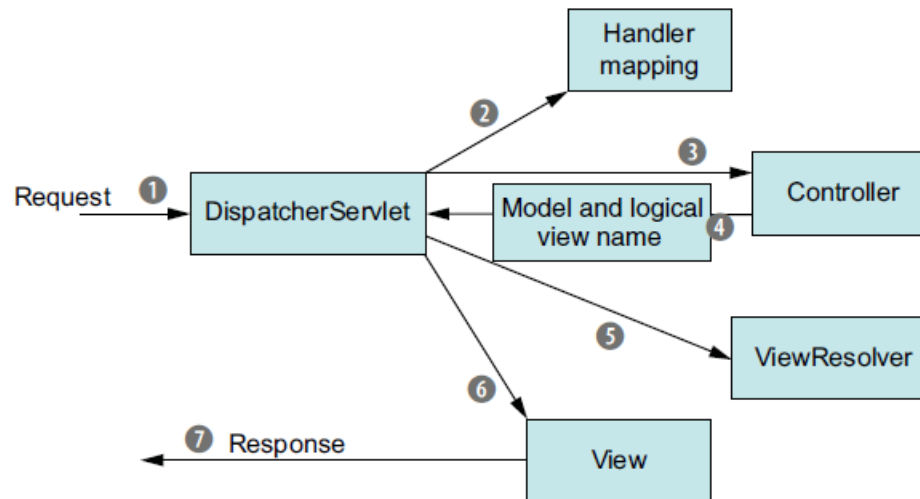


Life of a Request in Spring MVC



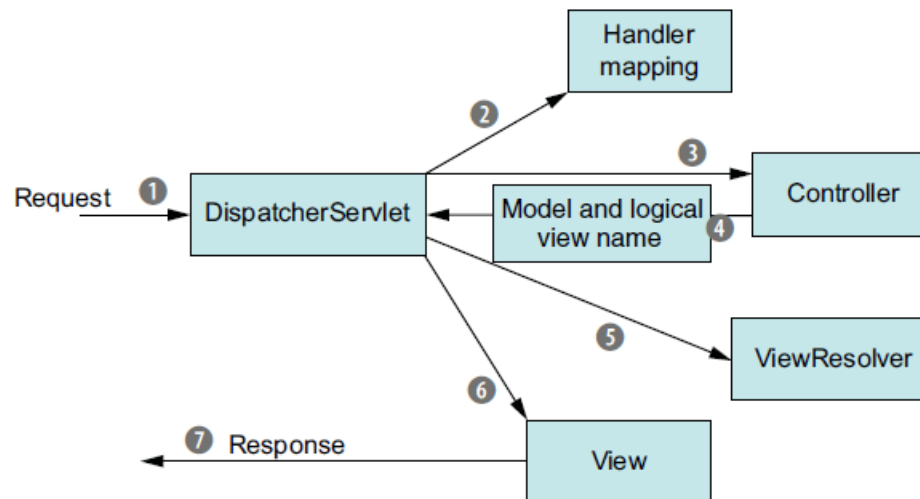
- The request carries information about what the user is asking for, e.g., requested URL, information submitted in a form.
- 1. The request arrives at Spring's **DispatcherServlet**, whose job is to pass the request to a Spring MVC controller.
- A **controller** is a Spring component that processes the request.
- A typical application may have several controllers.

Life of a Request in Spring MVC (cont')



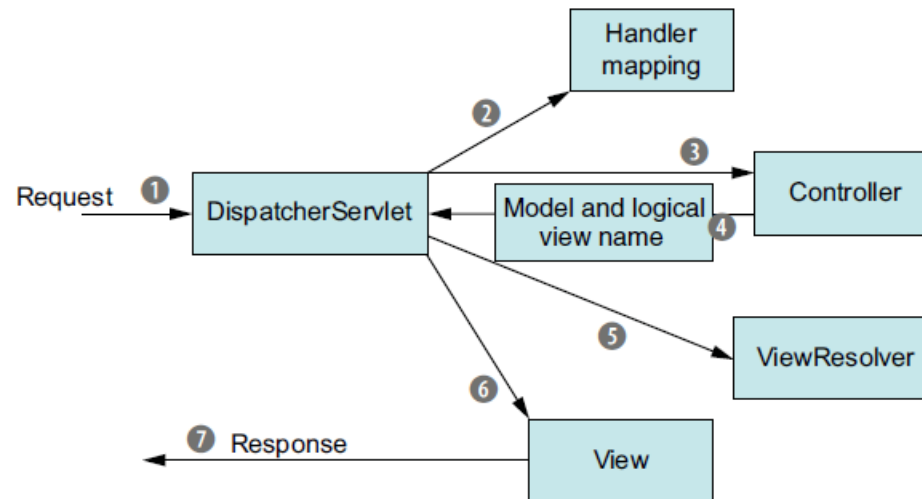
2. The DispatcherServlet needs to consult one or more **handler mappings** to figure out which controller to pass the request to.
3. The DispatcherServlet then sends the request to the chosen **controller**, which will then process the request.
- After processing, the controller may have information that needs to be carried back to the user and displayed in the browser; this information is referred to as the **model**.

Life of a Request in Spring MVC (cont')



- The model needs to be formatted in a user-friendly format, so a **view** (e.g., JSP) is needed.
- 4. The controller sends the model data and the name of a view for rendering the output, back to the DispatcherServlet.
- 5. The DispatcherServlet consults a **view resolver** to map the logical view name to a specific view implementation (e.g., a JSP).

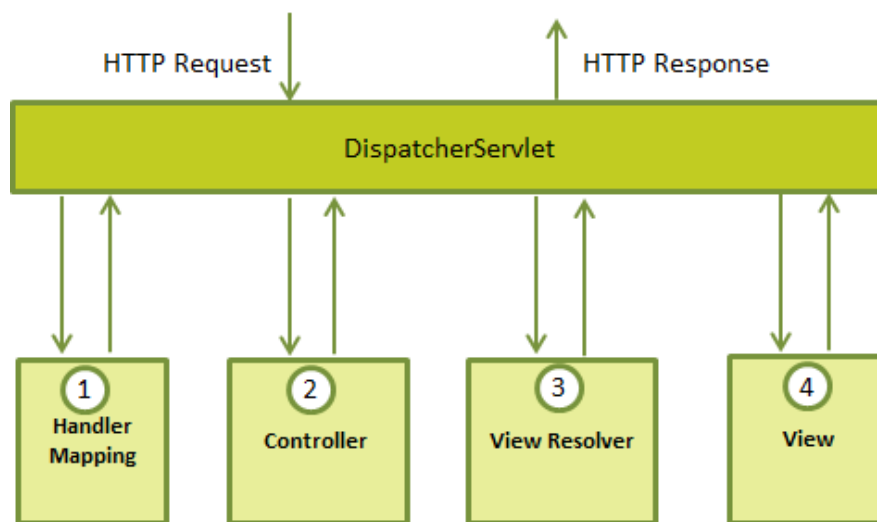
Life of a Request in Spring MVC (cont')



6. The `DispatcherServlet` sends the request's job to the view implementation.
7. The view will use the model data to render output that will be carried back to the client by the response object.

Configuration for Development of Spring MVC

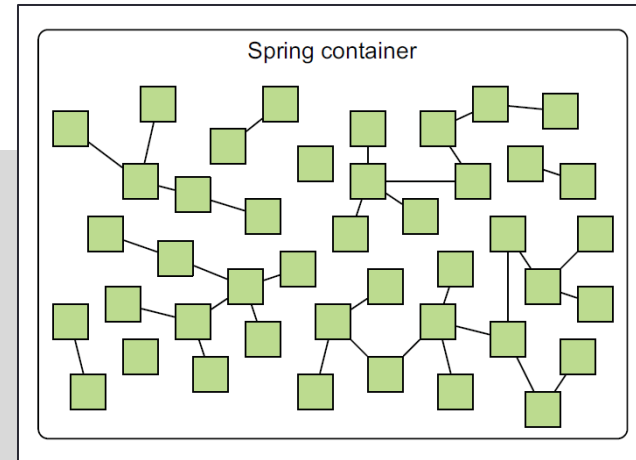
- Spring MVC is relatively easy to add to any existing Spring project
- Maven dependency: **spring-webmvc**
- We will need to configure **web.xml** and **servlet.xml** for, e.g.,
 - Configuring the DispatcherServlet
 - Defining a ViewResolver
 - Handle mappings with/without annotation
- That is basically the **wiring** of different components in the framework



Configuring Dispatcher: web.xml

We need to do some setup in the web.xml:

```
<web-app ...>  
  <context-param>  
    <param-name>contextConfigLocation</param-name>  
    <param-value>/WEB-INF/root-context.xml</param-value>  
  </context-param>  
  
  <listener>  
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
  </listener>  
  ...
```



- The above XML configures the **root Spring application context**.
- In a Spring-based application, Spring components are called **Spring beans**.
- Spring beans live in a **Spring container**, which creates the Spring beans, wires them together, configures them, and manages their complete lifecycle.
- **Spring application context** is a type of Spring containers.

Configuring Dispatcher: web.xml (cont')

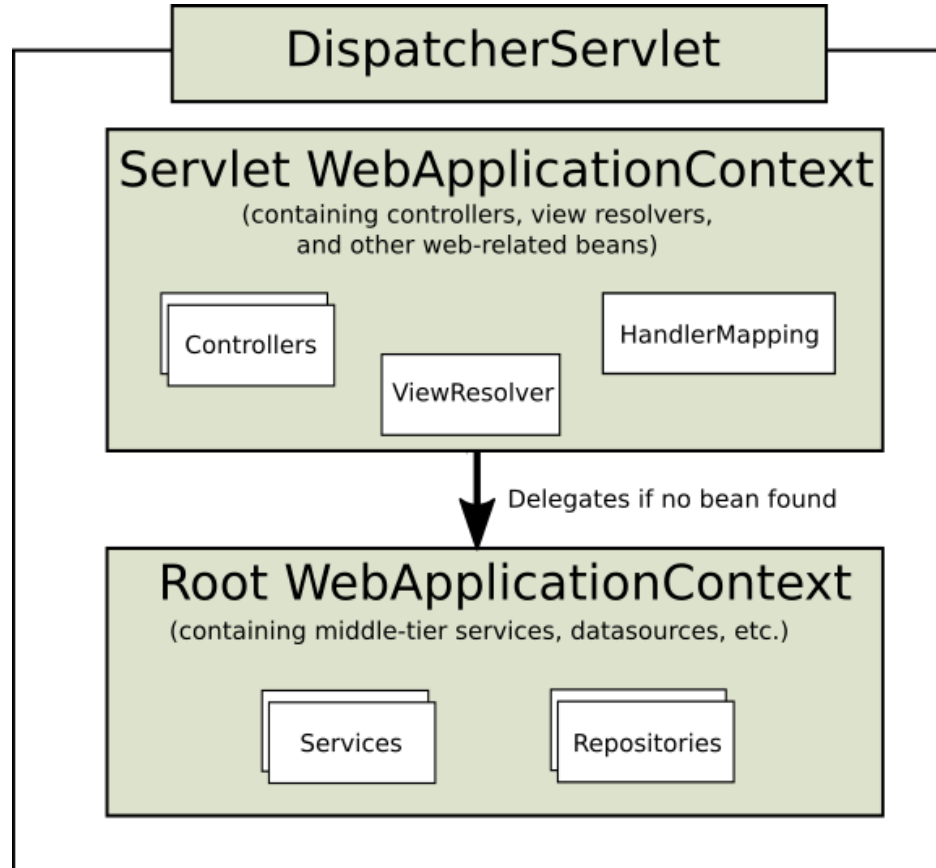
We need to do some setup in the web.xml:

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

- The above XML identifies one or more paths that DispatcherServlet will be mapped to.
- The URL pattern “/” means that **all HTTP requests** coming into the web application would go through the defined dispatcher.
- The dispatcher has the following fully-qualified class name: `org.springframework.web.servlet.DispatcherServlet`

Configuring Dispatcher: *[servlet-name]-servlet.xml*

- When the DispatcherServlet starts up, it creates **another Spring application context** only for web components (Spring beans)
 - E.g., Controllers, View resolvers, and Handler mappings.



Configuring Dispatcher: *[servlet-name]-servlet.xml* (cont')

- By default, Spring will load the configuration file of this servlet's application context from the file: `/WEB-INF/[servlet-name]-servlet.xml`
 - E.g., `/WEB-INF/dispatcher-servlet.xml`
- Spring XML configuration file has a root element **<beans>**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd">
  ...
</beans>
```

Example: *[servlet-name]-servlet.xml*

- Here is an example of dispatcher-servlet.xml:

```
<beans ...>
  <context:component-scan base-package="hkmu.comps380f" />
  <mvc:annotation-driven />

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

```
<context:component-scan base-package="package_name" />
```

- tells Spring to look for Spring components (which we call them **Spring beans**) inside the package *package_name*

Example: *[servlet-name]*-servlet.xml

- Here is an example of dispatcher-servlet.xml:

```
<beans ...>
  <context:component-scan base-package="hkmu.comps380f" />
  <mvc:annotation-driven />

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

<mvc:annotation-driven />

- enables annotation-driven Spring MVC, e.g., annotations for handling URL mapping

Example: *[servlet-name]*-servlet.xml

- Here is an example of dispatcher-servlet.xml:

```
<beans ...>
  <context:component-scan base-package="hkmu.comps380f" />
  <mvc:annotation-driven />

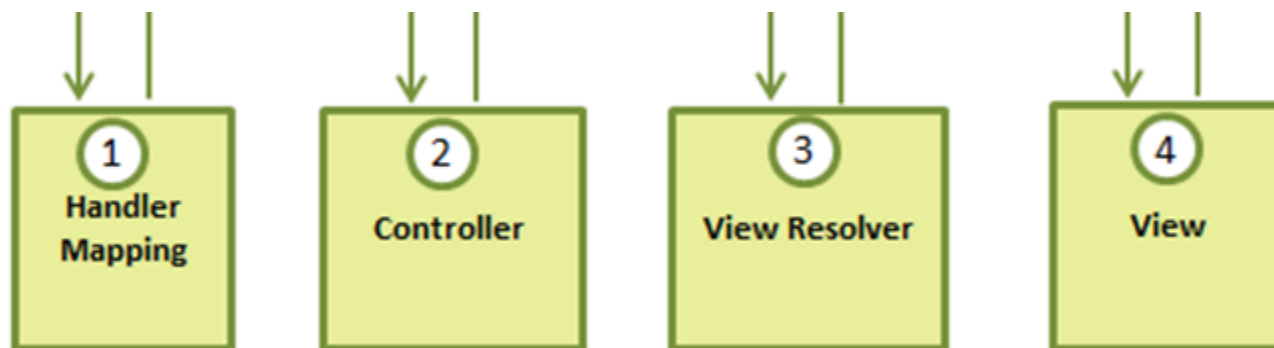
  <bean id="jspViewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
      value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
  </bean>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

```
<bean id="jspViewResolver"
class="org.springframework.web.servlet.view.InternalResourceView" >
...
```

- Define a Spring bean of class ***InternalResourceViewResolver***, which is a ***view resolver*** that maps a logical view name to a specific view implementation that may or may not be a JSP page.

Development in Spring MVC

- With the DispatcherServlet, we are ready to see how to develop our web application.
- Four components are available:
 - Controller
 - Handler Mapping
 - View Resolver
 - View



Adding Controller

Web app example: HelloSpring

- First of all, in your controller, we need to import a number of classes in the Spring Framework API
 - E.g. Controller, ModelAndView, RequestMapping

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

- A **controller** is a Spring component that processes the request.
 - Here, it has a method “index” for handling request

```
package hkmu.comps380f;
....
public class DefaultController {
    public String index() {
        return "myindexstatic";
    }
}
```

View Resolver: Resolving JSP Views

```
public String index() {  
    return "myindexstatic";  
}
```

- A controller method returns a view name, which will be resolved to a view implementation by a view resolver:

`myindexstatic` = `/WEB-INF/jsp/myindexstatic.jsp`

- Recall the “jspViewResolver” configuration:

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

dispatcher-servlet.xml



Prefix + ViewName + Suffix = JSP File

Handler Mapping

- From Spring 3, we can use annotation to define handler mappings.
- The **@Controller** annotation defines a class as a Spring MVC controller
- Requests are mapped to controller methods (or handler methods) using **@RequestMapping**
 - **value**: URL pattern(s)
 - Mapping can be further refined by using **method**, **params**, **headers**
 - <https://docs.spring.io/spring-framework/docs/5.3.16/reference/html/web.html#mvc-ann-requestmapping>

```
...  
@Controller  
public class DefaultController {  
  
    @RequestMapping(value="/", method=RequestMethod.GET)  
    public String index() {  
        return "myindexstatic";  
    }  
}
```

Shortcuts of @RequestMapping

- `@RequestMapping(value="/", method=RequestMethod.GET)` can be simplified to `@GetMapping("/")`, starting from Spring 4.3.
- The following is a list of all **request-mapping annotations** available:

Annotation	Description
<code>@RequestMapping</code>	General-purpose request handling
<code>@GetMapping</code>	Handles HTTP GET requests
<code>@PostMapping</code>	Handles HTTP POST requests
<code>@PutMapping</code>	Handles HTTP PUT requests
<code>@DeleteMapping</code>	Handles HTTP DELETE requests
<code>@PatchMapping</code>	Handles HTTP PATCH requests

JSP Views

- A request to <http://localhost:8080/HelloSpring/> maps to the index function of our controller.
- The returned view name “myindexstatic” maps to the view implementation JSP page: myindexstatic.jsp.
- The content in myindexstatic.jsp is displayed, which contains only static contents:

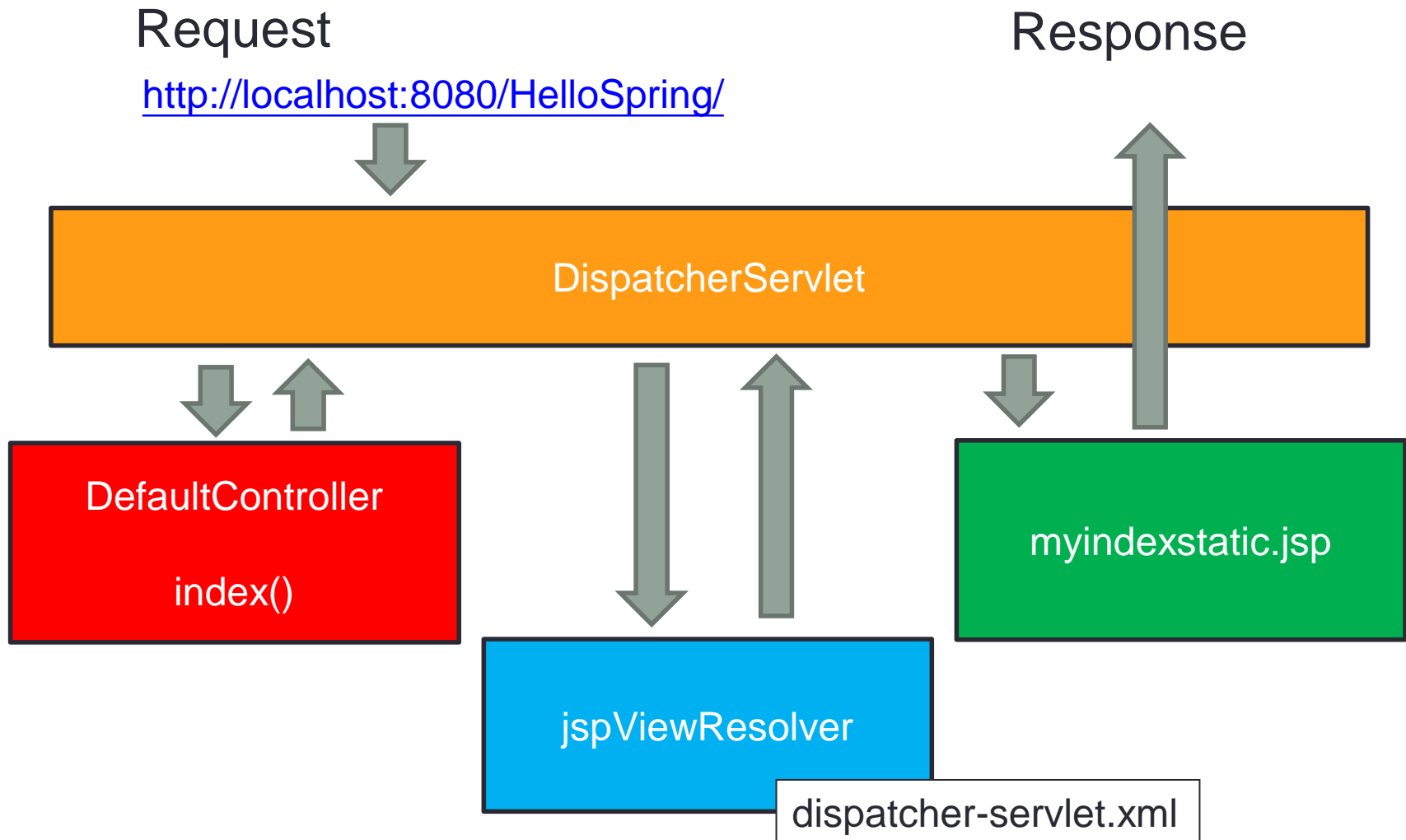
myindexstatic.jsp

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Spring MVC</title>
  </head>
  <body>
    <h1>Hello Spring</h1>
    <p>This sample show a static content page for demonstration.
      You can either use a custom controller or a predefined controller in Spring.</p>
  </body>
</html>
```

Hello Spring

This sample show a static content page for demonstration. You can either use a custom controller or a predefined controller in Spring.

Illustration



Model: ModelMap

- Only static content is displayed in the last example
- To display dynamic content in Spring, the standard way is to store it in the model component
- Model objects are passed to view using a **ModelMap object**.
- ModelMap is one of several objects that Spring can send to the method.
 - Another example is HttpServletRequest, which we are familiar with.
 - You just have to add required objects as method arguments to the handler, then Spring will load them in for you.
 - Please refer to the list in Slide 53 for other possible arguments

```
public String dynamicindex(ModelMap map) {  
    map.addAttribute("hello", "Welcome to COMP S380F Spring Lecture !");  
    return "myindex";  
}
```

ModelMap and JSP View

- The ModelMap stores attributes, which are key-value pairs.
- In the last example, we added the pair :

Key	Value
hello	Welcome to COMPS380F Spring Lecture !

- The data stored can then be referenced in the JSP view.
- We can use EL to access the ModelMap object's attributes:

```
<h1>Hello Spring</h1>
```

```
<p>This sample show you how the MVC (Model View Controller) in action  
within Spring MVC.</p>
```

```
Message to display :
```

```
<p>${hello}</p>
```

myindex.jsp

Hello Spring

This sample show you how the MVC (Model View Controller) in action within Spring MVC.

Message to display :

Welcome to COMPS380F Spring Lecture !


More on Request Mapping

- Request Mappings are flexible.
- We can define various methods for handling different requests.
- E.g., We can separate the above examples into 2 handlers.

```
@GetMapping("/")  
public String index() {  
    return "myindexstatic";  
}
```

Invoked by

<http://localhost:8080/HelloSpring/dynamic>



```
@GetMapping("/dynamic")  
public String dynamicindex(ModelMap map) {  
    map.addAttribute("hello", "Welcome to COMPS380F Spring Lecture !");  
    return "myindex";  
}
```

More on Request Mapping (cont')

- Define a **@RequestMapping** **on a class**. Then all other methods' request-mapping annotations will be relative to it.
- In following example, the method “dynamicindex” will be invoked when the URL pattern “**/global/dynamic**” is matched.
- Place the RequestMapping annotation outside of the class to make it “class level”.

```
@RequestMapping("/global")
@Controller
public class GuestBookController {

    @GetMapping("/dynamic")
    public String dynamicindex(ModelMap map) {
        map.addAttribute("hello", "Welcome to COMPS380F Spring Lecture !");
        return "myindex";
    }
}
```


More on Request Mapping (cont')

There are a number of ways to define the request-mapping annotations:

- URL patterns
- HTTP methods (GET, POST, etc)
- Request parameters
- Header values

Other related annotations are therefore available :

- `@PathVariable`
- `@RequestParam`
- `@RequestHeader`
- `@RequestBody`

@RequestMapping – HTTP Methods

Same URL as the previous example, but respond to POST request

```
@PostMapping("/")  
public String index() {  
    return "myindexstatic";  
}
```

Same URL as the previous example but respond to certain input parameter.

Here only respond to a GET request with a request parameter: details=all

```
@GetMapping(value="/view", params="details=all")  
public String index() {  
    return "myindexstatic";  
}
```

Controller Method Arguments

- Sometimes you need access to the request, session, request body, or other items
- If you add them as arguments to your controller method, Spring will pass them in automatically.
- The request parameters (String) are also converted to the corresponding data type automatically.

```
@GetMapping("/")  
public String getProject(HttpServletRequest request,  
                        HttpSession session,  
                        @RequestParam("projectId") Long projectId,  
                        @RequestHeader("content-type") String contentType) {  
    return "index";  
}
```

Controller Method Arguments: Example

- This gives you access to the request/response and session

```
@GetMapping("/")
public String index(HttpServletRequest request,
                    HttpServletResponse response,
                    HttpSession session) {
    return "myindex";
}
```

- This gives you access to request parameters and headers

```
@GetMapping("/")
public String getProject(
    @RequestParam Long projectId,
    @RequestHeader("content-type") String contentType) {
    return "index";
}
```

Get value from the request parameter "projectId"

Other Supported Method Arguments

- `ModelMap`
- Request/Response objects
- Session object
- Spring's `WebRequest` object
- `java.util.Locale`
- `java.io.Reader` (access to request content)
- `java.io.Writer` (access to response content)
- `java.security.Principal`
- `org.springframework.validation.Errors`
- `org.springframework.validation.BindingResult`

ModelMap & ModelAndView

- You populate the view with data via the **ModelMap** or **ModelAndView** (the latter has both the ModelMap and the view underneath).
- All attributes are added to the request, so they can be picked up by JSPs.
- In **ModelMap**, use “**addAttribute**” to insert new attributes.

```
public String doController(ModelMap modelMap) {  
    modelMap.addAttribute("hello", "This is a message.");  
    return "index";  
}
```

Message to display :

<p> **hello** </p>

In JSP

- We can combine the model and view into one object : **ModelAndView**
- In **ModelAndView**, use “**addObject**” to insert new attributes.

```
public ModelAndView myController() {  
    ModelAndView mav = new ModelAndView("index");  
    mav.addObject("hello", "This is a message.");  
    return mav;  
}
```

Using a POJO for Data

- Similar to JavaBean we used before
- E.g., We define a Java class called “MyData”

```
public class MyData {  
    private Integer num;  
    private String name;  
  
    // Getters and Setters of num & name  
}
```

- We can use the POJO for our model layer:

```
public String doController(ModelMap modelMap) {  
    mydata = new MyData();  
    mydata.setName("abc");  
    mydata.setNum(10);  
  
    modelMap.addAttribute("data", mydata);  
    return "myoutput";  
}
```

Spring Form: JSP page

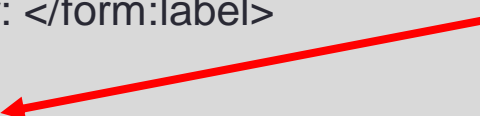
- Spring can tie HTML form parameters to **a form-backing object** (POJO).
 - A form-backing object is like JavaBean, but can be more easily used.
- We need the **Spring form tag library** in the input form:

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

- Like Java bean, the form-backing object has properties (i.e., variables).
- Spring can associate the form parameters to these properties.
- Suppose we reuse the form-backing object “MyData” with 2 properties: name & num

```
<form:form method="POST" action="/HelloSpring/formhandle" >  
  <form:label path="name">Enter a name: </form:label>  
  <form:input path="name"/><br />  
  <form:label path="num">Enter a number: </form:label>  
  <form:input path="num"/> <br />  
  <input type="submit" value="Submit" />  
</form:form>
```

You may also use plain HTML tags in a Spring form



Spring Form: Controller

- The ModelAndView object has an attribute "command" equal to a blank MyData object.
- By default, the Spring framework expects that the form-backing object in a Spring form has name "command"

```
@GetMapping("/myform")  
public ModelAndView myform() {  
    return new ModelAndView("myform", "command", new MyData());  
}
```

view name

Provide the POJO as
model

Spring Form: Generated HTML Form

- Spring will process the tag and generate input form in HTML format

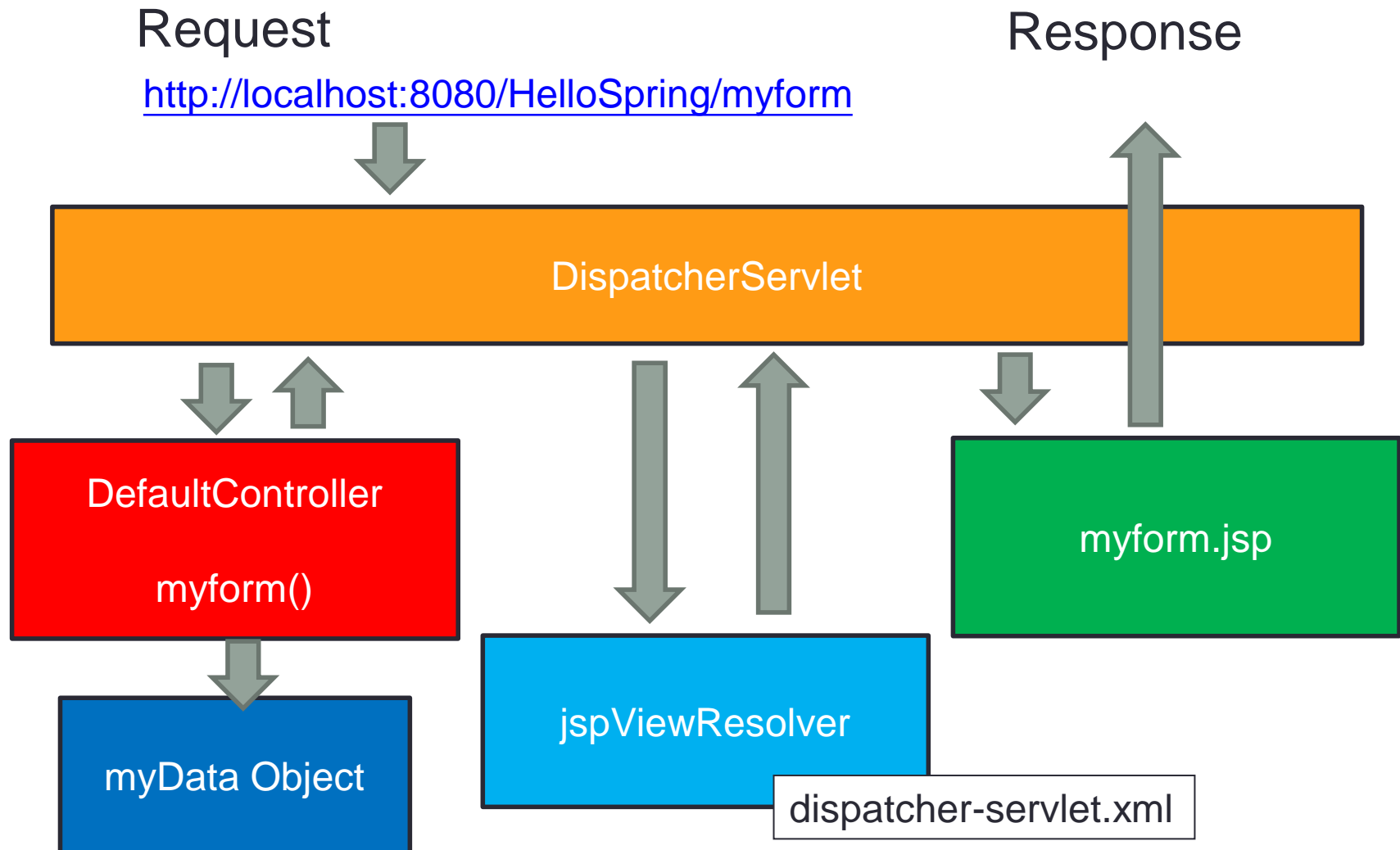
```
<form:form method="POST" action="/HelloSpring/formhandle" >  
  <form:label path="name">Enter a name: </form:label>  
  <form:input path="name"/><br />  
  <form:label path="num">Enter a number: </form:label>  
  <form:input path="num"/> <br />  
  <input type="submit" value="Submit" />  
</form:form>
```



```
<form id="command" action="/HelloSpring/formhandle" method="POST">  
  <label for="name">Enter a name: </label>  
  <input id="name" name="name" type="text" value=""/><br />  
  <label for="num">Enter a number: </label>  
  <input id="num" name="num" type="text" value=""/><br />  
  <input type="submit" value="Submit" />  
</form>
```

Enter a name:	<input type="text" value="abc"/>
Enter a number:	<input type="text" value="123"/>
<input type="submit" value="Submit"/>	

Illustration: Showing a Spring Form



Spring Form: Handling a Spring Form

- As the form will submit to “/HelloSpring/formhandle” using POST method, the “formHandle” method will be invoked.
- Add the “data” attribute to ModelMap, which is displayed in myoutput.jsp

```
@PostMapping("/formhandle")  
public String formHandle(MyData mydata, ModelMap map) {  
    map.addAttribute("data", mydata);  
    return "myoutput";  
}
```

```
<h1>Form Output</h1>  
<p>${data.name}: ${data.num}</p>
```

myoutput.jsp

Form Output

abc: 123

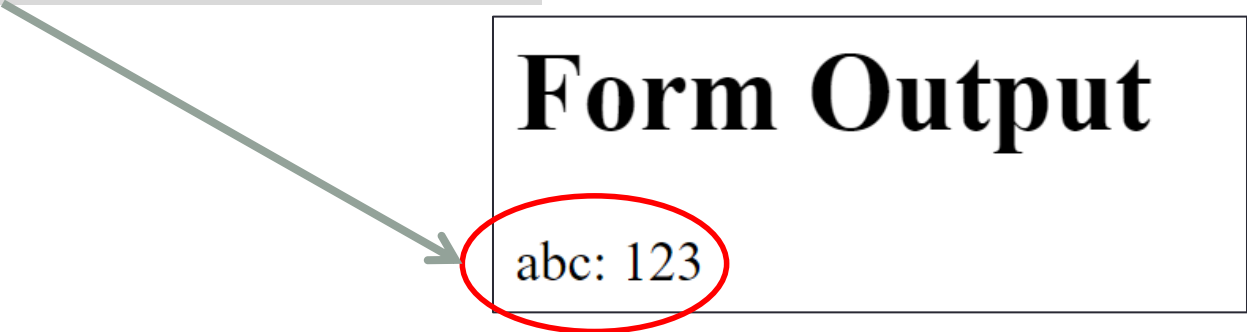
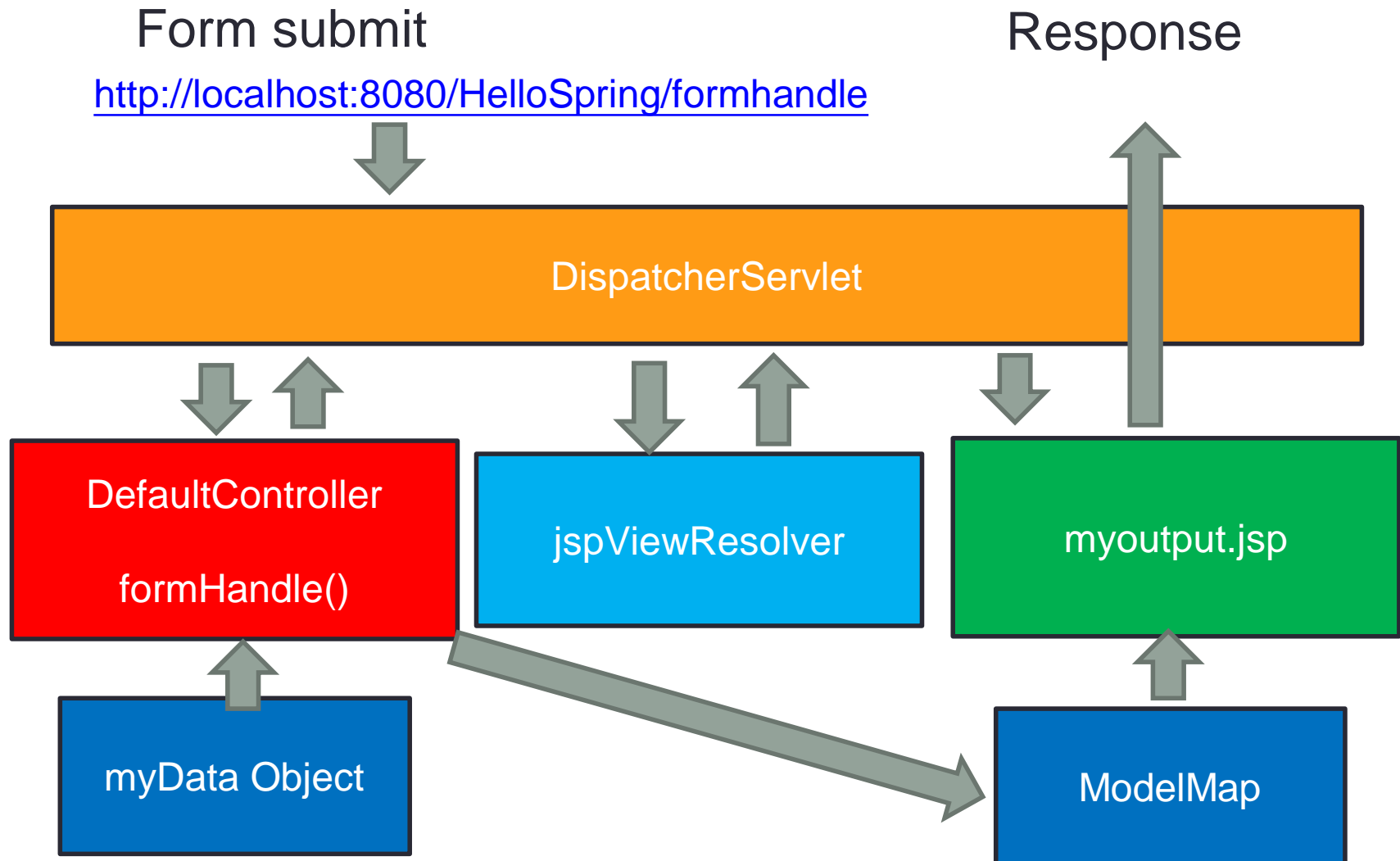
A green arrow points from the JSP template code in the block above to the rendered output in this block. The rendered output "abc: 123" is circled in red.

Illustration: Handling a Spring Form



Spring Form: Naming form-backing object

- We can customize the name of the form-backing object

```
<form:form method="POST" action="/HelloSpring/formhandle"
           modelAttribute="myData">
  <form:label path="name">Enter a name</form:label>
  <form:input path="name"/><br />
  <form:label path="num">Enter a number </form:label>
  <form:input path="num"/> <br />
  <input type="submit" value="Submit" />
</form:form>
```



```
@PostMapping("/formhandle")
public String formHandle(@ModelAttribute("myData") MyData mydata,
                        ModelMap map) {
    map.addAttribute("data", mydata);
    return "myoutput";
}
```

Spring Form Tag Library (examples)

Tag	Description
<code>form:form</code>	Generates the HTML <code><form></code> tag, which has the <code>modelAttribute</code> attribute for specifying the form-backing object
<code>form:input</code>	Represents the HTML input text tag
<code>form:password</code>	Represents the HTML input password tag
<code>form:radiobutton</code>	Represents the HTML input radio button tag
<code>form:checkbox</code>	Represents the HTML input checkbox tag
<code>form:select</code>	Represents the HTML select list tag
<code>form:option</code>	Represents the HTML option tag
<code>form:errors</code>	Represents the HTML <code>span</code> tag, generated from the error created as a result of data validations

Documentation:

<https://docs.spring.io/spring-framework/docs/5.3.16/reference/html/web.html#mvc-view-jsp-formtaglib-formtag>

One more example for self-study

In the HelloSpring project...

- **Controller:**
GuestBookController.java
 - URL patterns:
 - = current directory
 - .. = parent directory
- **Form-backing object:**
GuestBookEntry.java
- **JSP pages:**
 - /WEB-INF/jsp/view/GuestBook.jsp
 - /WEB-INF/jsp/view/AddComment.jsp
 - /WEB-INF/jsp/view/EditComment.jsp

