COMP S380F Lecture 7: Spring Boot, More on Spring MVC

Dr. Keith Lee

School of Science and Technology

Hong Kong Metropolitan University

Overview of this lecture

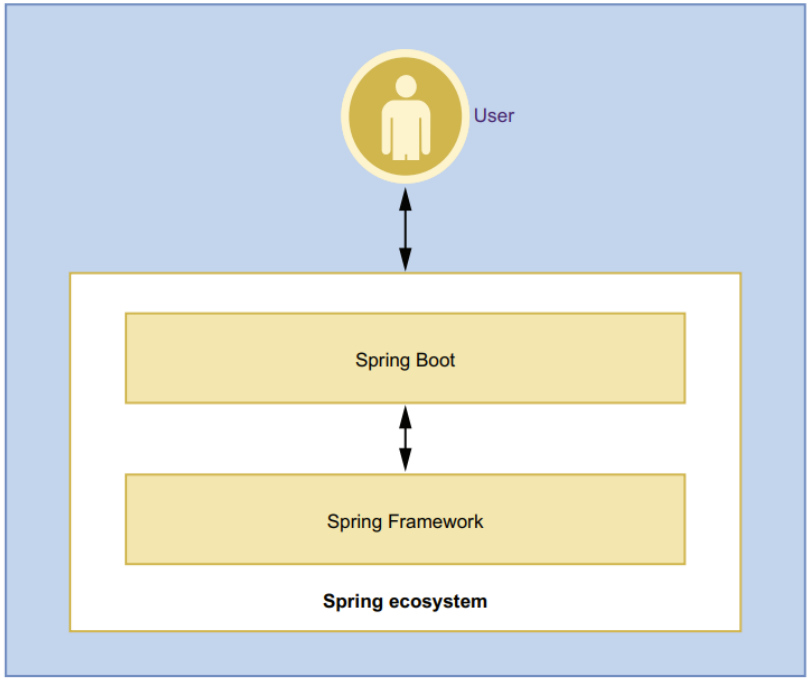
- Spring Boot
  - Spring Boot features
  - Starter dependency
  - Spring Initializr, Spring Boot project structure, Enable JSP
  - Spring Boot main class, *`application.properties`*
  - Spring Boot Actuator
- Spring MVC web framework (continue from last lecture…)
  - **Model**: ModelMap, ModelAndView
  - More on RequestMapping, Controller method arguments
  - **HTML Form in Spring:** Form-backing objects, Spring form taglib
  - **Self-study example:** GuestBook application

What is Spring Boot?

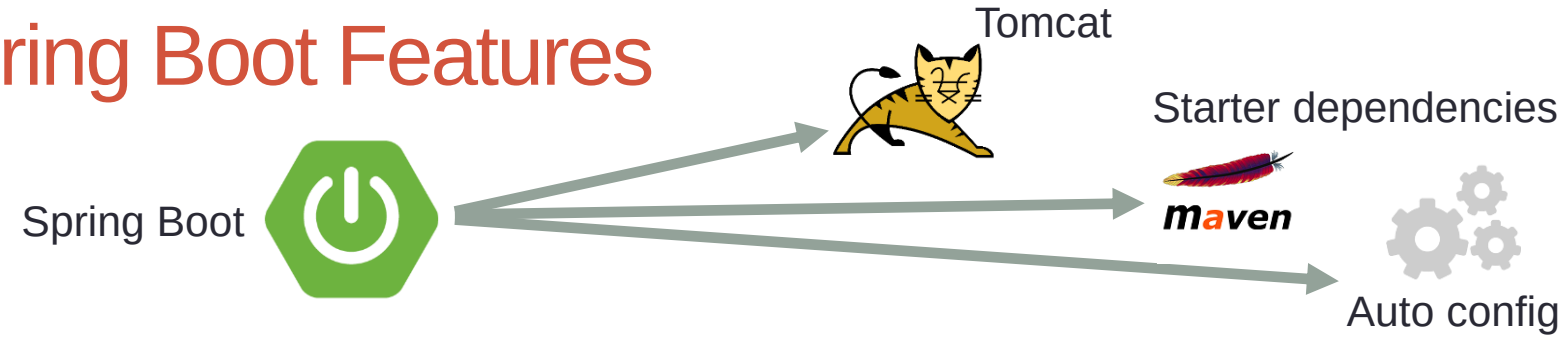
- One of the most popular Java frameworks
- Open-source **extension** (not replacement) of Spring framework for simplifying Spring application development
- Provides a quick way to create standalone, production-ready, Spring-based applications without a lot of configurations.

For web app development:

- Developers do not require understanding of Servlet and the associated web.xml concepts.
- No requirement for XML config



Spring Boot Features

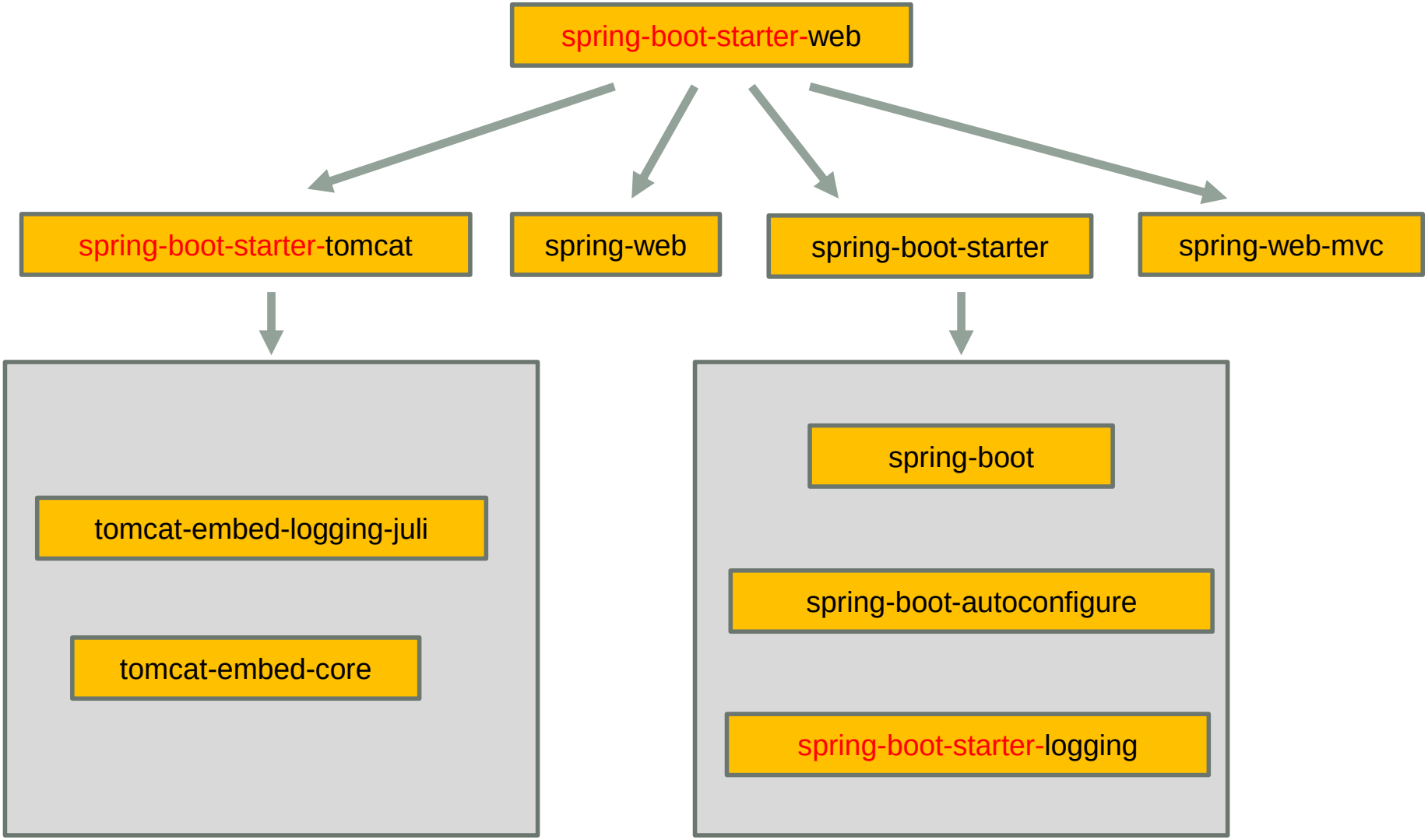


- **Fast-bootstrapping**: Provides easy, rapid, effective startup experience in Spring app development via a set of starter dependencies.
- **Opinionated auto-configuration**: Automatically configures (according to Spring Boot's opinions) the bare minimum components, based on presence of JAR files in the classpath, or properties configured in various property files.
- **Standalone**: Spring Boot applications embed a web container (Tomcat), so they can run standalone (packaged as an executable JAR file) without an external web container / application server.
- **Production-ready features**: Provides several useful features to monitor and manage the application, such as health checks, thread dumps, and other useful metrics, via Spring Boot Actuator component.

Starter dependency

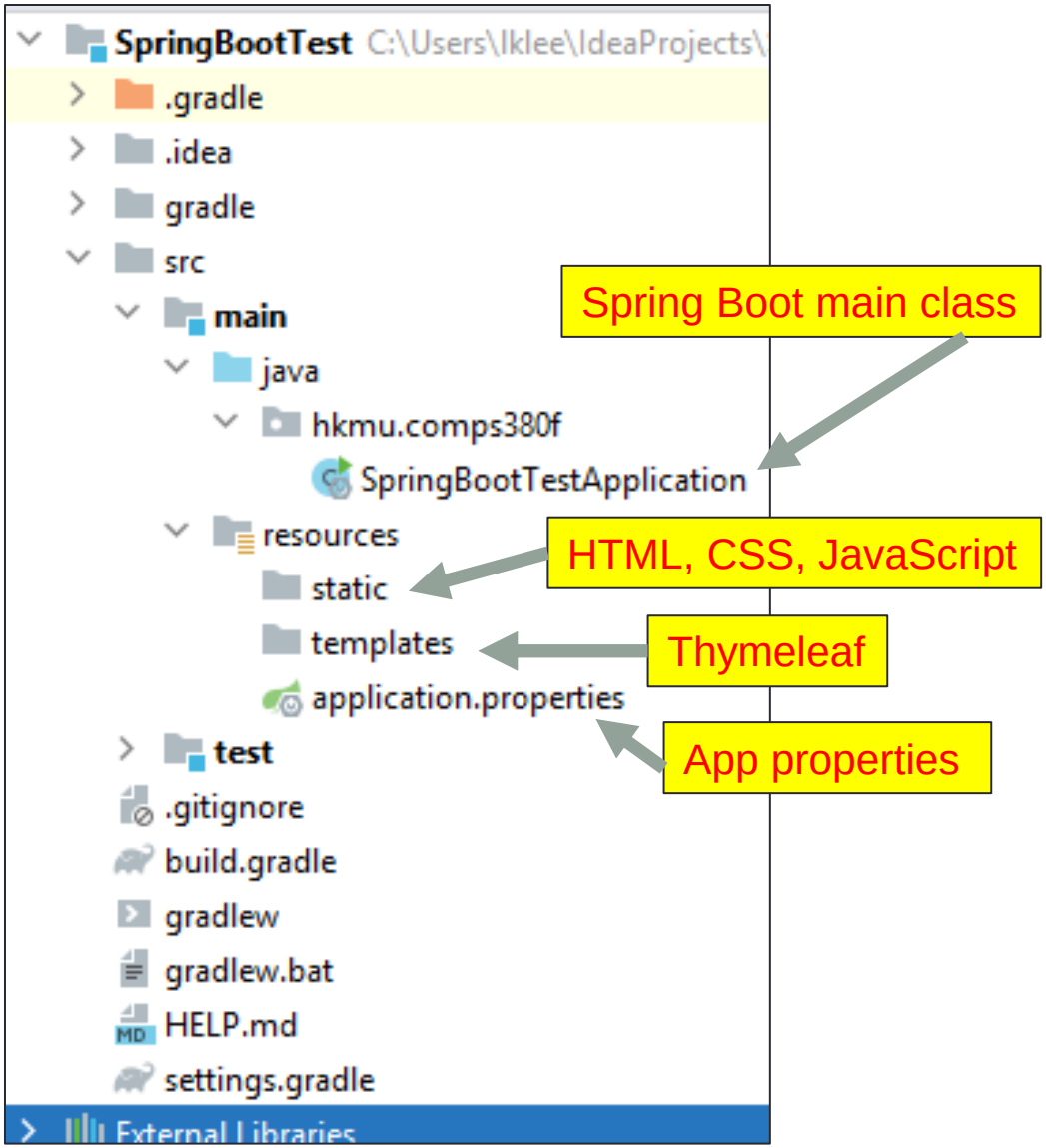
- Each starter dependency groups together a set of dependencies for a particular function
- Spring Boot also handles dependency versioning, upgrades, and many other issues for developers
- They are named in the pattern: `spring-boot-starter-`…
  - `spring-boot-starter-web` (for Spring MVC & Tomcat)
  - `spring-boot-starter-security` (for Spring Security)
  - `spring-boot-starter-data-jpa` (for Spring Data JPA)
  - `spring-boot-starter-mongodb` (for MongoDB)
  - `spring-boot-starter-logging` (for logging)
  - `...`

Starter dependency: Example



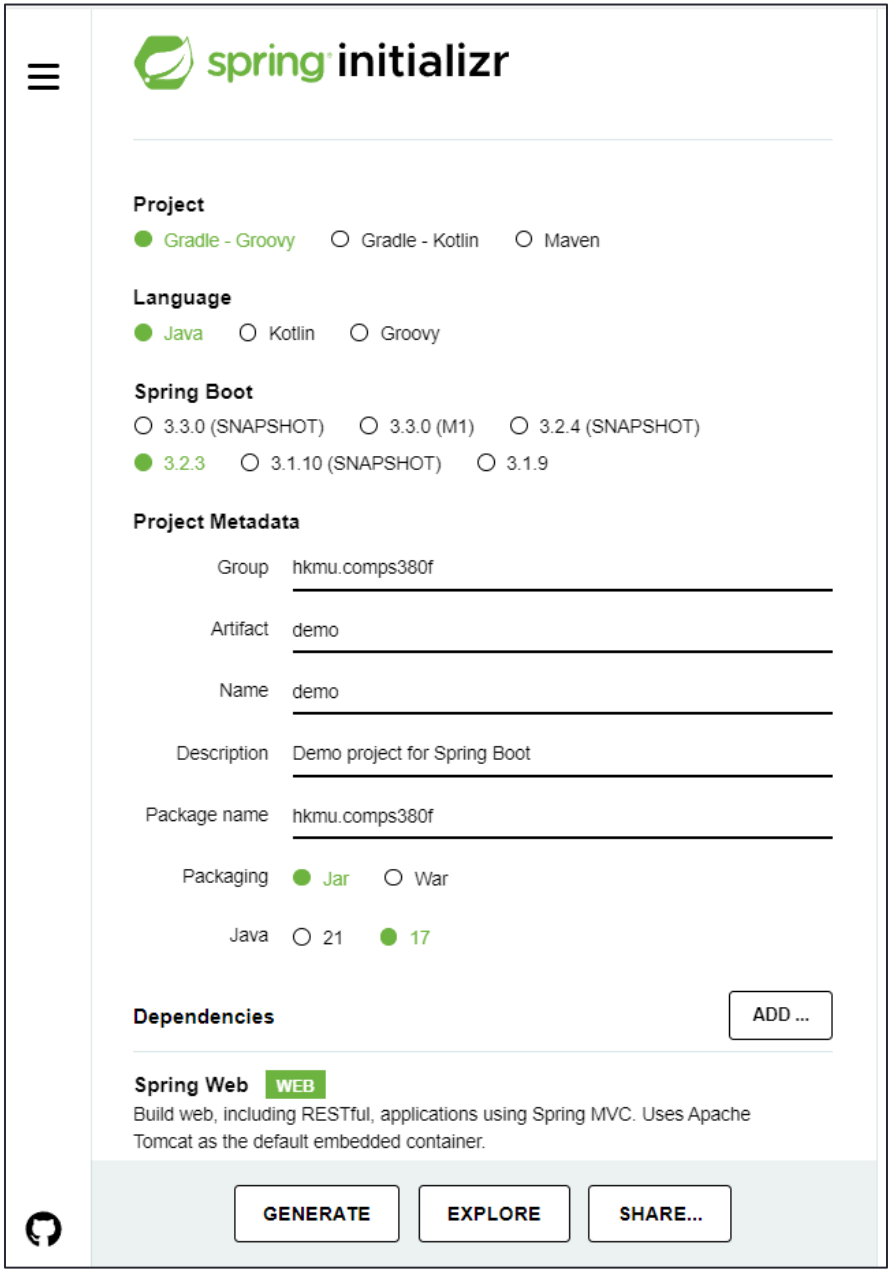
Spring Initializr & Spring Boot Project Structure

- **Spring Initializr** is a web-based tool for creating new Spring Boot projects quickly and efficiently
- Generates a **Spring Boot project structure**, with desired Spring Boot version, build system, and dependencies.
- Spring Boot supports templating engines such as Thymeleaf, Mustache, and FreeMarker, but **not JSP** by default.



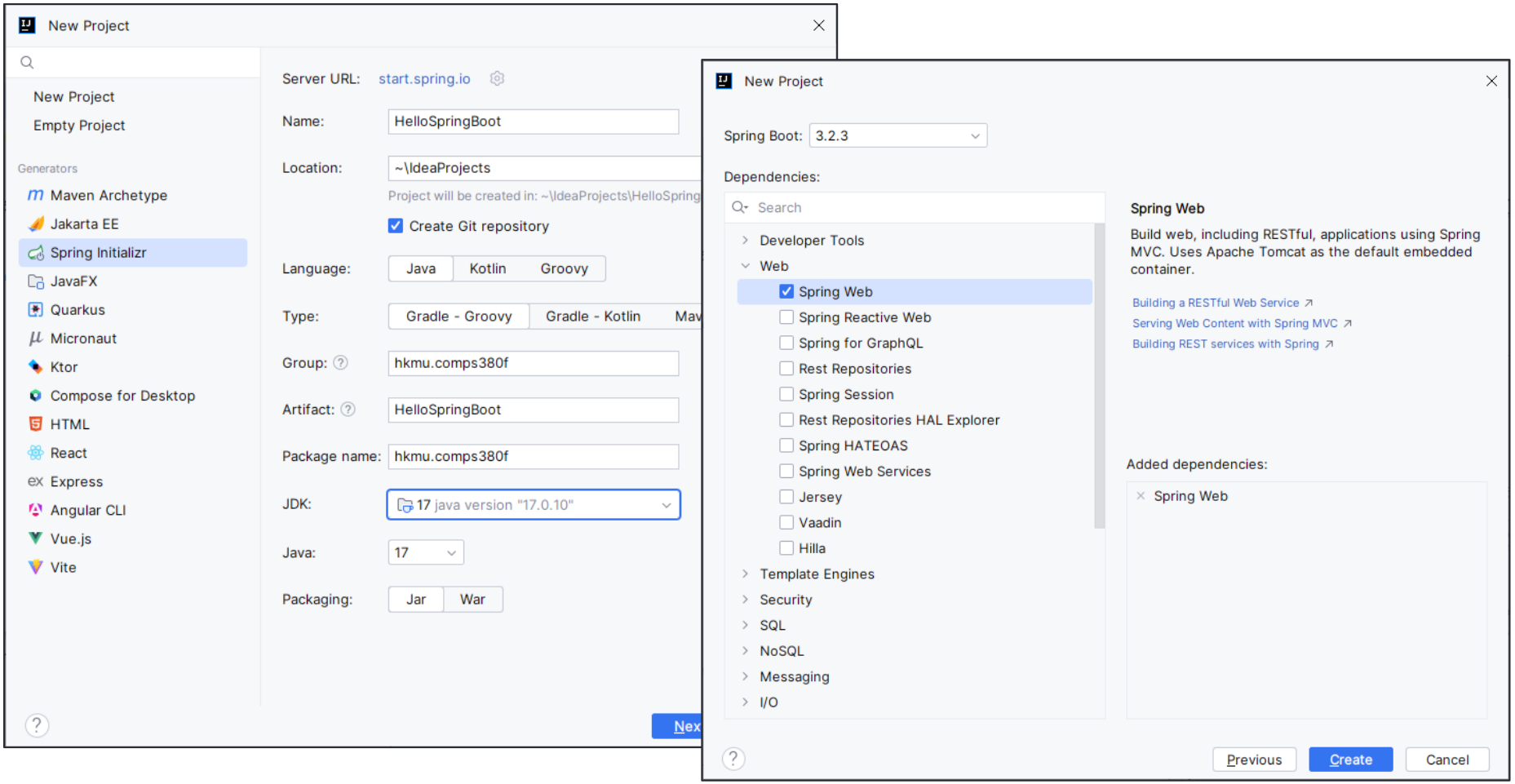
Using Spring Initializr

- Go to https://start.spring.io/
- Fill in the desired Spring Boot project information
- Generate and download the project



Using Spring Initializr (cont')

- Some IDE (e.g., IntelliJ IDEA Ultimate) has Spring Initializr integration, so you can complete this process from your IDE.



Enable JSP development in Spring Boot

- Gradle dependencies:

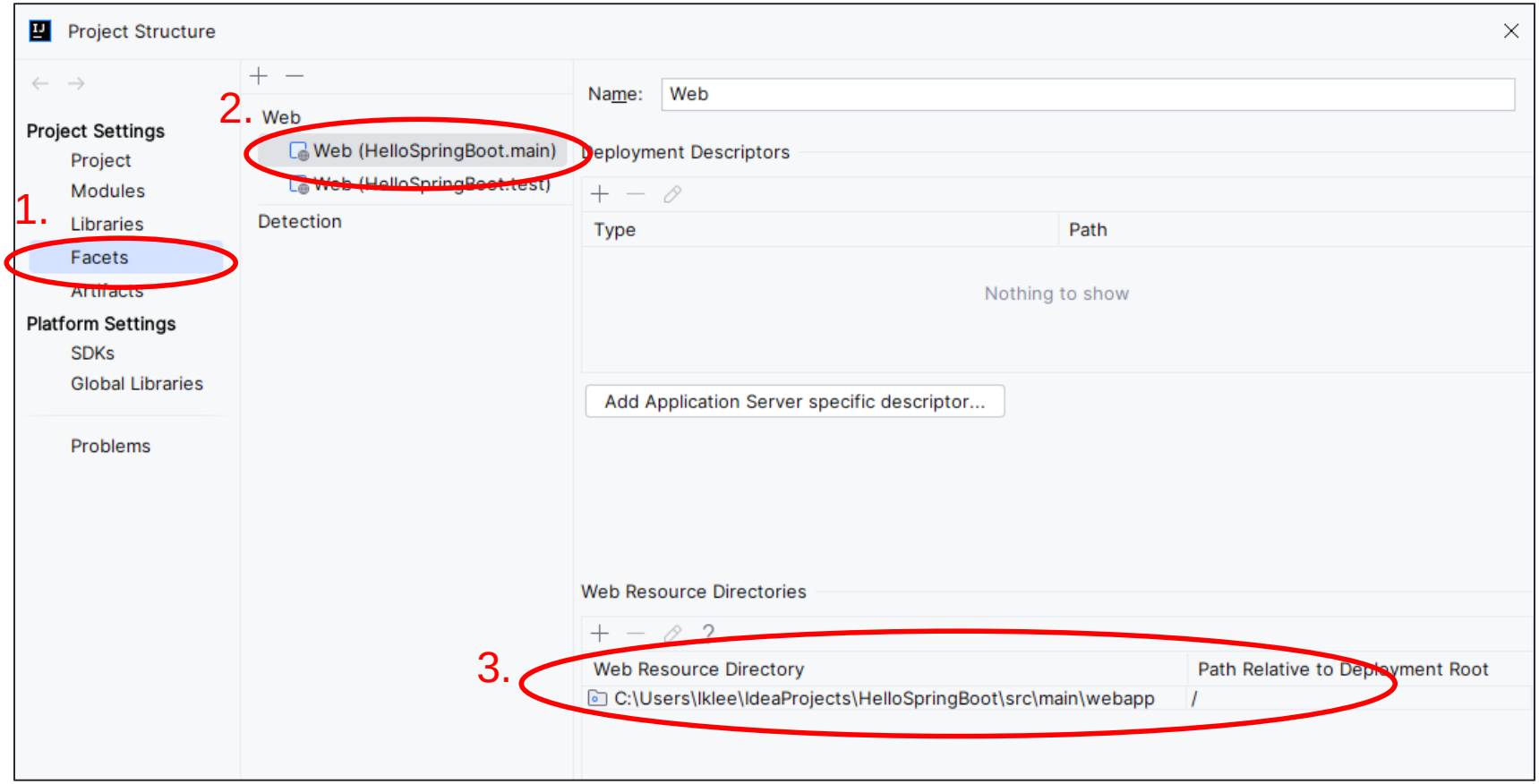
```
compileOnly 'org.apache.tomcat.embed:tomcat-embed-jasper:10.1.19'
implementation 'jakarta.servlet.jsp:jakarta.servlet.jsp-api:3.1.1'
implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0'
implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl:3.0.1'
implementation 'jakarta.el:jakarta.el-api:5.0.1'

implementation 'org.springframework.boot:spring-boot-starter-web'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
```

  - `tomcat-embed-jasper` provides support for JSP file rendering.
  - Using `compileOnly` can avoid conflicting version of the same dependency inside the embedded Tomcat web container.

Enable JSP development in Spring Boot (cont')

- Project structure:



- Put HTML/JSP pages in `webapp` (instead of `static` and `templates`)
- Create the `WEB-INF` folder in `webapp` for hidden web contents

Spring Boot Main Class

Web app example: lecture07-hellospringboot

```
package hkmu.comps380f;
//imports

@SpringBootApplication
public class HelloSpringBootApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloSpringBootApplication.class, args);
    }
}
```

- We can run a Spring Boot application like a regular Java application using the conventional `main()` method.
- The `@SpringBootApplication` annotation includes the following annotations:

➢ `@EnableAutoConfiguration`: Auto-config based on JAR on classpath.

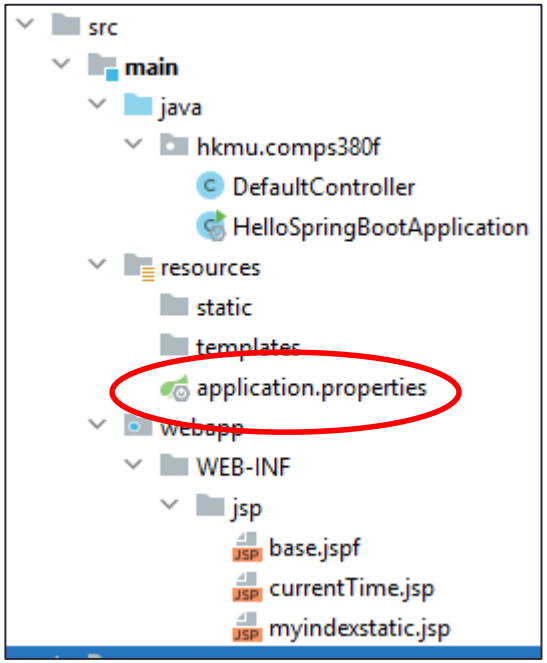
➢ `@ComponentScan`: Loads Spring Beans in the package & its child packages.

➢ `@SpringBootConfiguration`: Indicates that the class provides Spring Boot application configuration, so the beans defined in this main class can be auto-detected and loaded by Spring.

```java
@SpringBootApplication
public class HelloSpringBootApplication {
   //...
   @Bean
   public ConfigurableServletWebServerFactory configurableServletWebServerFactory() {
     return new TomcatServletWebServerFactory() {
        @Override
        protected void postProcessContext(Context context) {
            super.postProcessContext(context);
            JspPropertyGroup jspPropertyGroup = new JspPropertyGroup();
            jspPropertyGroup.addUrlPattern("*.jsp");
            jspPropertyGroup.addUrlPattern("*.jspf");
            jspPropertyGroup.setPageEncoding("UTF-8");
            jspPropertyGroup.setScriptingInvalid("true");
            jspPropertyGroup.addIncludePrelude("/WEB-INF/jsp/base.jspf");
            jspPropertyGroup.setTrimWhitespace("true");
            jspPropertyGroup.setDefaultContentType("text/html");
            JspPropertyGroupDescriptorImpl jspPropertyGroupDescriptor
                      = new JspPropertyGroupDescriptorImpl(jspPropertyGroup);
            context.setJspConfigDescriptor(
               new JspConfigDescriptorImpl(
                   Collections.singletonList(jspPropertyGroupDescriptor),
                                                Collections.emptyList()));
        }
    };
}
```

- This Spring bean replaces the `<jsp-config>` in `web.xml` (which no longer exists).

The “application.properties” file



application.properties

```
server.servlet.context-path=/HelloSpring

server.servlet.session.timeout=30m
server.servlet.session.cookie.http-only=true
server.servlet.session.tracking-modes=cookie

spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

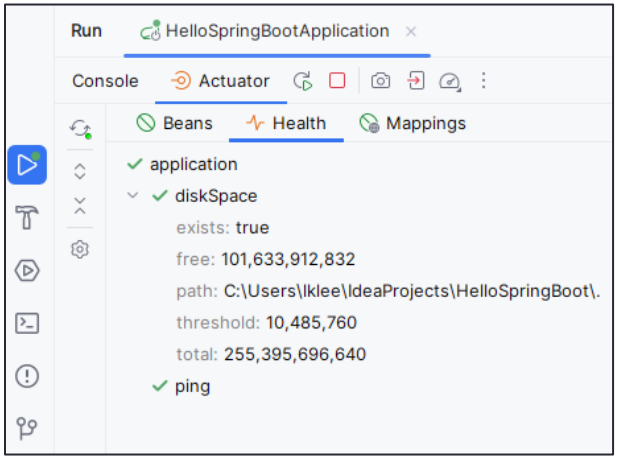
- The most common (out of many other) way to configure Spring Boot application
  - Put `application.properties` (or `application.yml`) in the classpath.
- A property file lets you specify configurations in a key-value pair format.
- DispatcherServlet and View Resolver are automatically configured.
- Properties `server.servlet.session.*` replaces `<session-config>` in `web.xml`

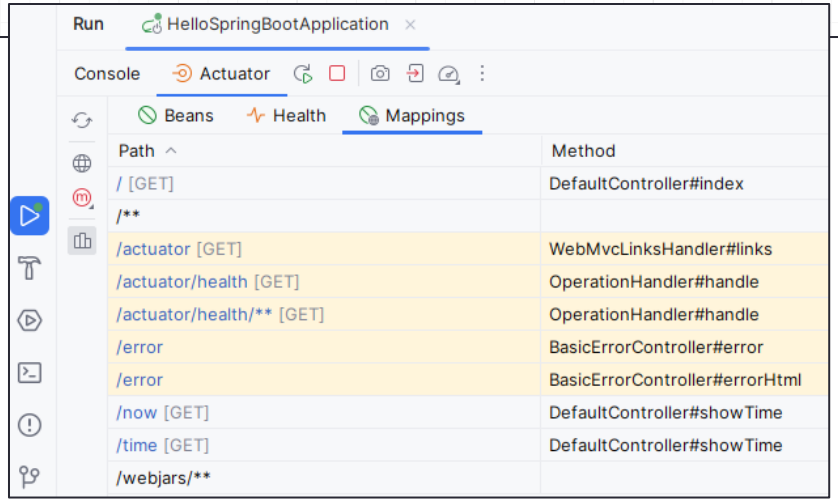
Spring Boot Actuator

- Exposes different types of information about the running Spring Boot application.
- Add the following Gradle dependency, then Actuator will be auto-configured:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
```

- IntelliJ provides a nice interface for viewing the information from Actuator:





MORE ON SPRING MVC

Model: ModelMap

Web app example: lecture07-hellospringmvc

- 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` we are familiar with.
  - You just have to add the required objects as method arguments to the handler, then Spring will load them in for you.
  - Please refer to the list in Slide 25 for other possible arguments.

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

ModelMap and JSP View

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

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:

myindex.jsp

```
<h1>Hello Spring MVC</h1>
<p>This sample show you how the MVC (Model View Controller) in action
   within Spring MVC.</p>
Message to display :
<p>${hello}</p>
```

Hello Spring MVC

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 controller methods (i.e., handlers) for handling different requests.

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

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

Invoked by
http://localhost:8080/HelloSpring/dynamic

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".

```
@Controller
@RequestMapping("/global")
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) {
    //...
}
```

Controller Method Arguments: Example

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

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

- This gives you access to request parameters and headers

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

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

Documentation:
https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/arguments.html

ModelMap & ModelAndView

- You populate the view with data via **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 myHandler(ModelMap modelMap){
    modelMap.addAttribute("hello", "This is a message.");
    return "index";
}
```

In JSP

Message to display :
 ${hello}

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

```
public ModelAndView myHandler() {
    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 myHandler(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 prefix="form" uri="http://www.springframework.org/tags/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 action="formhandle" method="POST">
   <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 the name "**command**".
- If you use a different name (e.g., "**myData**"), you need to specify it using the **modelAttribute** attribute of the <form:form> tag.

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

view name

Provide the POJO as model

```
<form:form action="formhandle" method="POST" modelAttribute="myData">
    …
</form:form>
```

Spring Form: Generated HTML Form

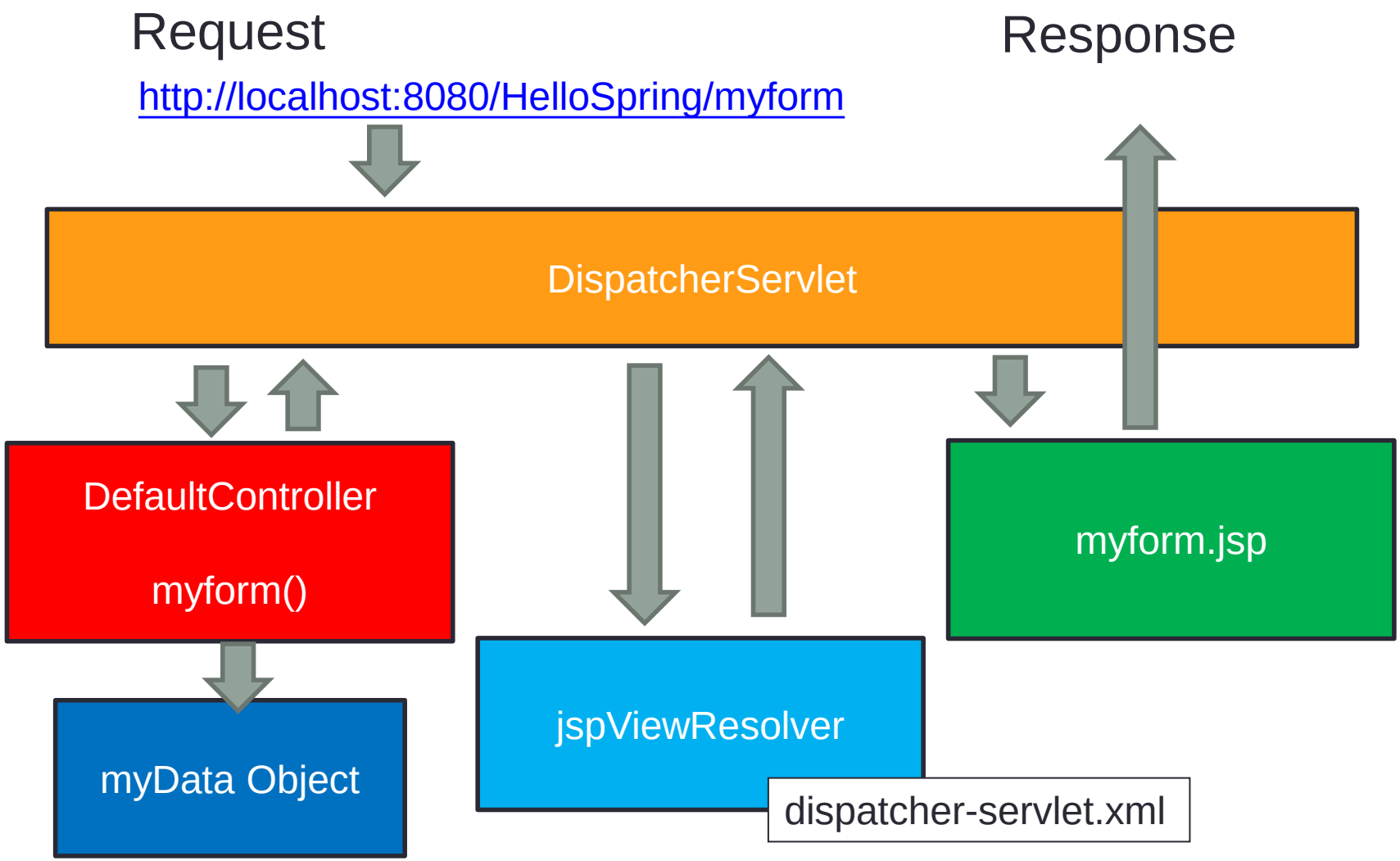
- Spring will process the tag and generate input form in HTML format

```
<form:form method="POST" action="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>
```

```
<form id="myData" action="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: abc
Enter a number: 123
Submit

Illustration: Showing a Spring Form



Spring Form: Handling a Spring Form

- As the form will submit to the URL pattern "`/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";
}
```

myoutput.jsp

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

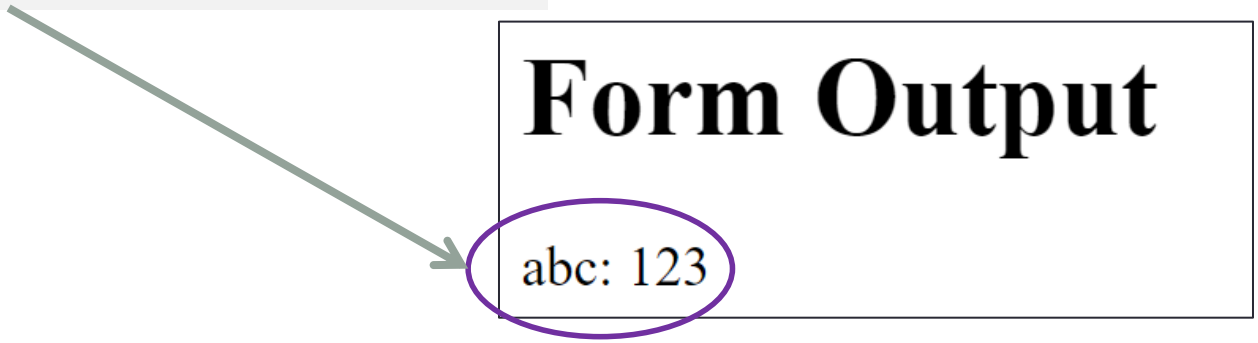
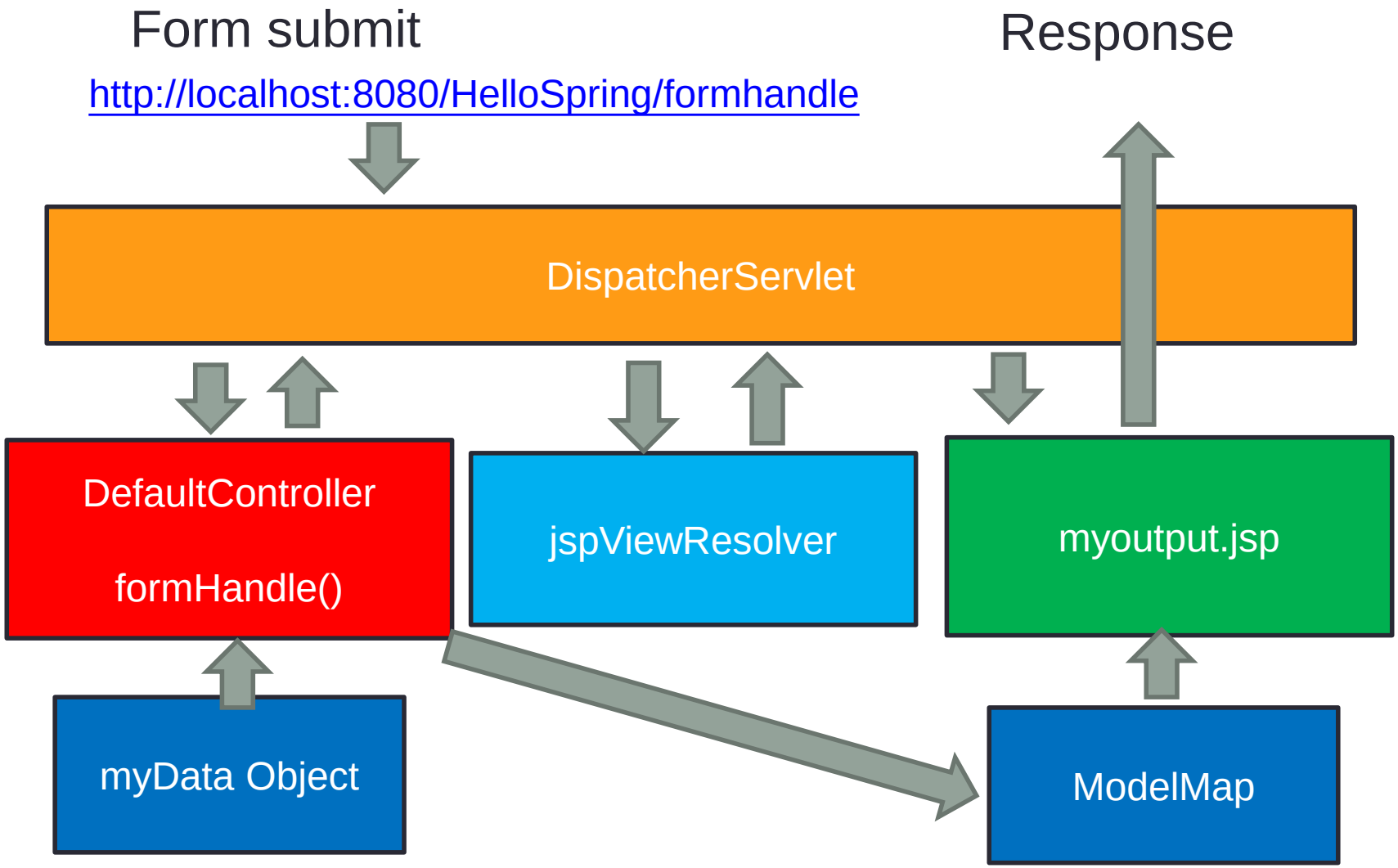


Illustration: Handling a Spring Form



Spring Form: Naming form-backing object

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

```
<form:form method="POST" action="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 abc,
                         ModelMap map) {
    map.addAttribute("data", abc);
    return "myoutput";
}
```

Spring Form Tag Library (examples)

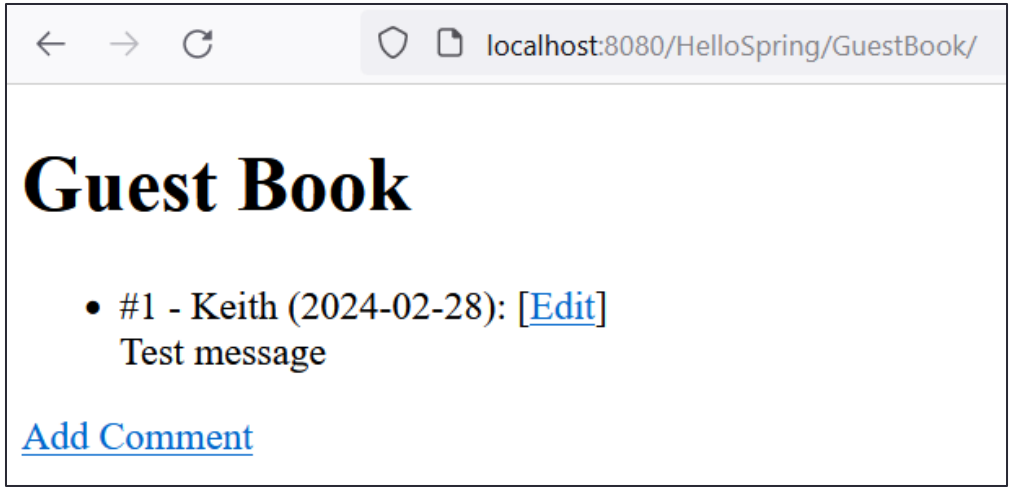
Tag	Description
`form:form`	Generates the HTML `<form>` tag, which has the `modelAttribute` attribute for specifying the form-backing object
`form:input`	Represents the HTML input text tag
`form:password`	Represents the HTML input password tag
`form:radiobutton`	Represents the HTML input radio button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

Documentation:
https://docs.spring.io/spring-framework/reference/web/webmvc-view/mvc-jsp.html#mvc-view-jsp-formtaglib-formtag

One more example for self-study

*In the **HelloSpringMVC** project…*

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



Mid-term Test

- **Date:** 8 March 2024 (Friday)
- **Time:** 9:15 – 10:30 (after the test, we will talk about the group project details)
- Please be punctual. You are suggested to arrive earlier.
- Students who are absent in the test will receive zero marks. Yet students with a strong justification for absence are allowed to participate in a make-up test.
- You may answer with pencil; modifying your answers is easier.
- **Scope:**
  - Lectures 1 – 6, Lab 2 – 7 (excluding Spring MVC)
  - **Coding questions:** Fill in the blanks to complete the code / write Java methods / predict web application output / debug a code segment.
  - **Conceptual questions:** Give definition of a term / explain differences of similar terms or concepts / identify & describe steps of a procedure
  - Check the review slides to see if you need to study harder.

Review: Lecture 1 & 2

- Differences between Web (HTTP) servers, Jakarta EE application server, Web container
- Understand Servlet's life cycle
- Understand how the web container handles a Servlet request
- Understand the difference between Attributes and Parameters
  - Context init parameter, Request parameter, Servlet init parameter
  - Context attribute, Request attribute, Session attribute
- Able to write a simple Servlet:
  - Servlet class with init(), doGet(), doPost(), destroy()
  - Deployment descriptor (/WEB-INF/web.xml)

Review: Lecture 3

- Able to write a simple JavaBean
- Understand JSP page's life-cycle
- JSP implicit objects
- JSP elements:
  - JSP directives: page, include, taglib
  - JSP scripting:
    - JSP expression, JSP comments
    - JSP scriptlet vs. declaration
  - JSP actions:
    - <jsp:include> (vs. <%@ include %>)
    - <jsp:useBean>, <jsp:getProperty>, <jsp:setProperty>
- Able to write a simple JSP page
- Forward the request & response in Servlet using RequestDispatcher

Review: Lecture 4

- Session Tracking Techniques
  - URL rewriting
  - HTML hidden fields
  - Cookies
  - HTTP Session object (HttpSession)
- Session Vulnerabilities and their Prevention
  - Copy and Paste Mistake
  - Session Fixation
  - Cross-Site Scripting (XSS)
  - Cross-Site Request Forgery (CSRF)
  - Insecure Cookies

Review: Lecture 5

- EL
  - EL implicit objects
  - Using dot and bracket [ ] operators
  - Accessing scoped variables
- JSTL
  - <c:if>
  - <c:forEach>
  - <c:choose>, <c:when>, <c:otherwise>
  - <c:set>: setting an attribute vs. setting a Map / Bean
  - <c:url>, <c:param>
  - ${fn:length(Object)}
  - ${fn:escapeXML(String)}

Review: Lecture 6

- Understand the role of each component of MVC:
  - Model
  - View
  - Controller
- MVC Model 1 vs. MVC Model 2
- Pros and cons of each model