JAVA FULL STACK DEVELOPMENT PROGRAM

Session 16: Spring MVC

OUTLINE

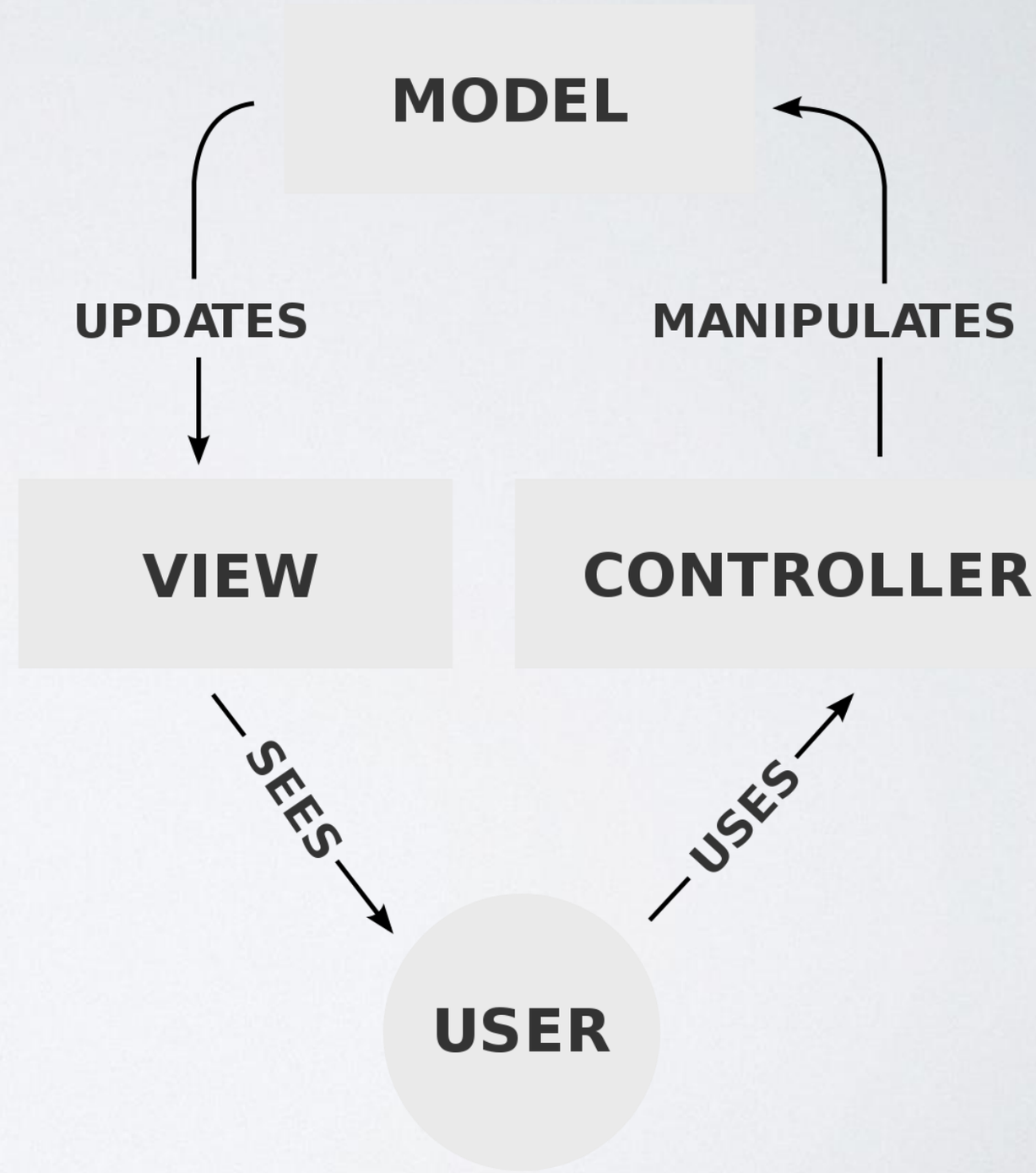
- Spring MVC introduction
- DispatcherServlet
- Controller

MVC

- MVC is an architecture that separates business logic, presentation and data. In MVC,
  - M stands for Model
  - V stands for View
  - C stands for controller.
- MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

Why MVC

- Separation of Concern
  - Loose coupling between model, view, and controller
  - Can have multiple views

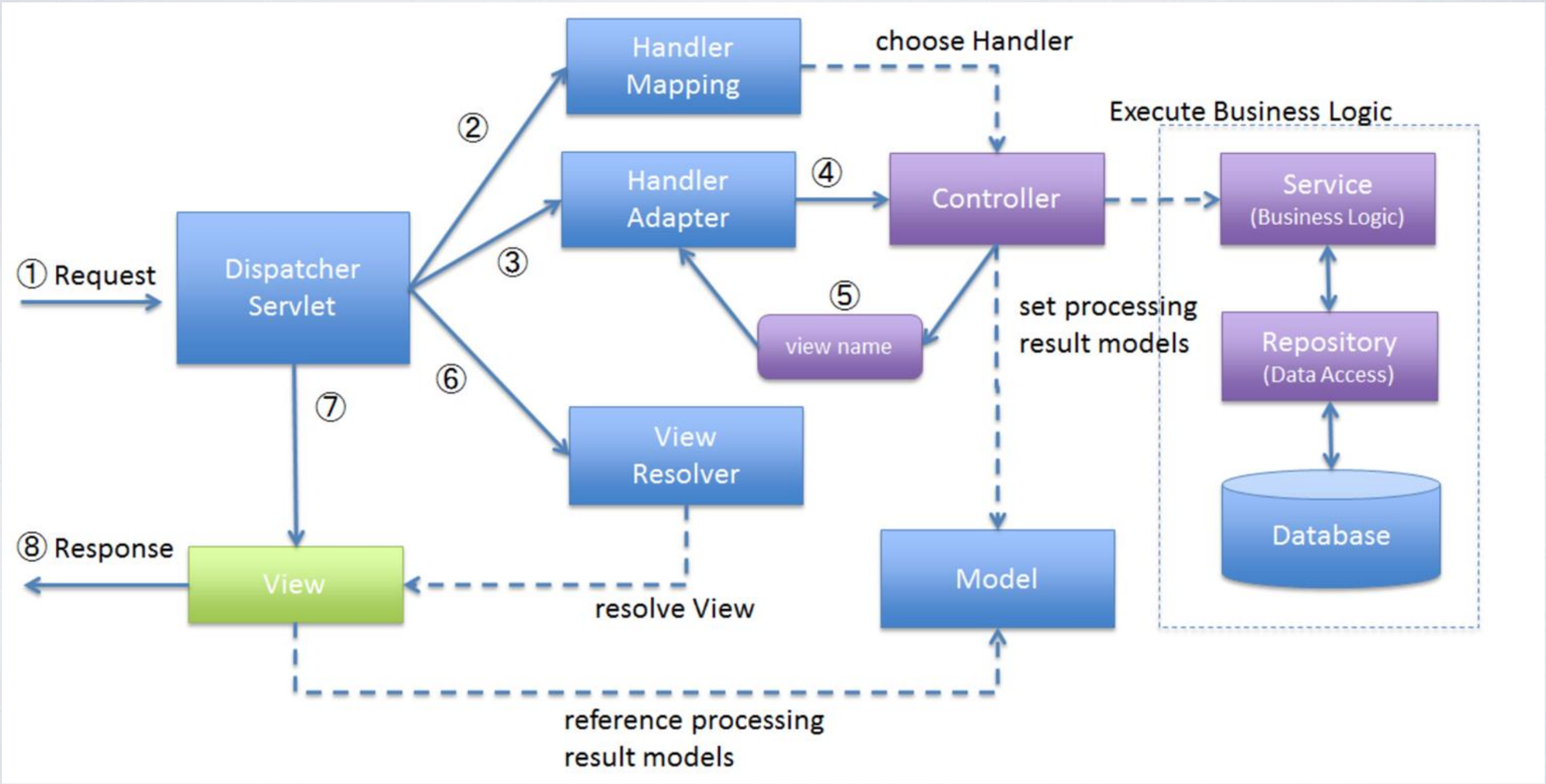


SPRING MVC

- Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.

- The formal name, “Spring Web MVC,” comes from the name of its source module (spring-webmvc), but it is more commonly known as “Spring MVC”

SPRING MVC


Handler Mapping
choose Handler
Execute Business Logic
②
Handler Adapter
④
Controller
Service (Business Logic)
① Request
Dispatcher Servlet
③
⑤
view name
set processing result models
Repository (Data Access)
⑥
⑦
View Resolver
Database
⑧ Response
View
Model
resolve View
reference processing result models

DISPATCHSERVLET

- Spring MVC, as many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate componnts

- The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification by using Java configuration or in web.xml. In turn, the DispatcherServlet uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling.

- *DispatcherServlet* expects a *WebApplicationContext* (an extension of a plain *ApplicationContext*) for its own configuration. *WebApplicationContext* has a link to the *ServletContext* and the *Servlet* with which it is associated.

DISPATCHSERVLET

- DispatcherServlet as the Heart of Spring MVC
- What we really want to do as developers of a web application is to abstract away the following tedious and boilerplate tasks and focus on useful business logic:
  - Mapping an HTTP request to a certain processing method
  - Parsing of HTTP request data and headers into data transfer objects (DTOs) or domain objects
  - Model-view-controller interaction
  - Generation of responses from DTOs, domain objects, etc.

DISPATCHSERVLET

- We can configure the DispatchServlet by either annotation or XML

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletCxt) {

        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext ac = new AnnotationConfigWebApplicationContext();
        ac.register(AppConfig.class);
        ac.refresh();

        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(ac);
        ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

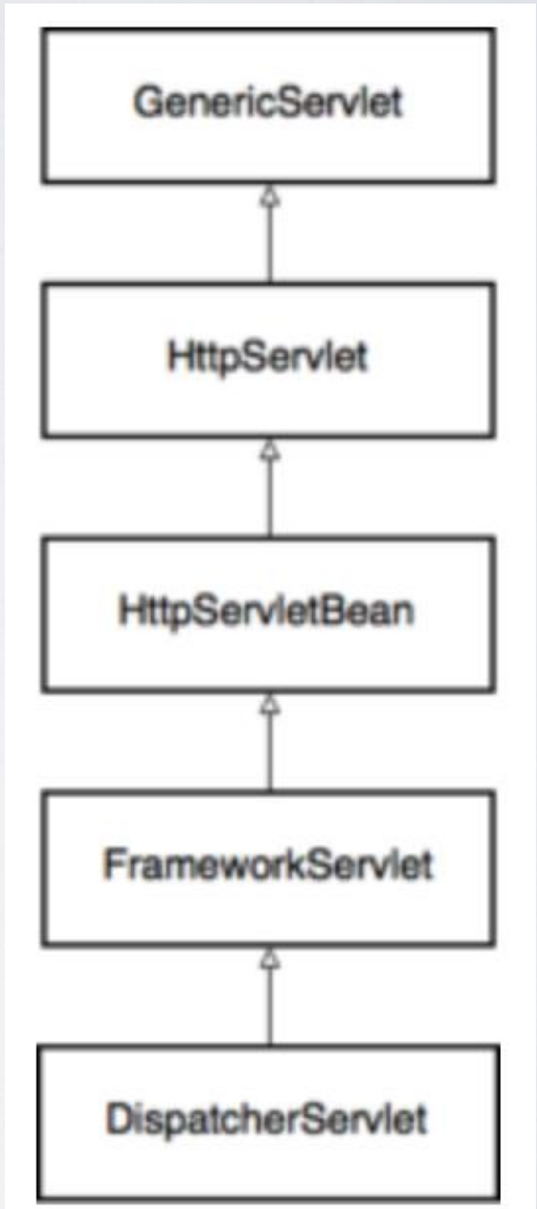
```
<web-app>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/app-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>app</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>

</web-app>
```

DISPATCHSERVLET

- How does DispatchServlet process a request?
  - Unifying the Request Processing
  - Enriching the Request
  - Dispatching the Request
  - Handling the Request
  - Processing Arguments and Return Values (Controller)
  - Rendering the View

UNIFYING THE REQUEST PROCESSING

- The *DispatcherServlet* has a long inheritance hierarchy
  - GenericServlet — service()
  - HttpServlet — doGet(), doPost()
- HttpServletBean — the first Spring-aware class in the hierarchy
  - It injects the bean's properties using the servlet *init-param* values received from the *web.xml* or from *WebApplicationInitializer*
- FrameworkServlet — *FrameworkServlet* integrates th Servlet functionality with a web application context



UNIFYING THE REQUEST PROCESSING

- DispatchServlet
  - The *HttpServlet.service()* implementation, which routes requests by the type of HTTP verb, makes perfect sense in the context of low-level servlets
- However, at the Spring MVC level of abstraction, method type is just one of the parameters that can be used to map the request to its handler
  - In J2EE servlet, we have to map the controller / servlet to its own url manually. However,

    DispatchServlet provides a unique method *processRequest*() to handle the request.

```
@Override
protected final void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}
```

ENRICH THE REQUEST

- In addition to handle the request in a unified way, DispatchServlet also adds to the request some useful objects that may come in handy down the processing pipeline
  - WebApplicationContext — configuration
  - LocaleResolver — language (English, Chinese)
  - FlashMap — Flash map is basically a pattern to pass parameters from one request to another request that immediately follows
    - This may be very useful during redirects (like showing the user a one-shot information message after the redirect)

DISPATCHING THE REQUEST

- After the enriching process, DispatchServlet will call dispatch() method
- The main purpose of the dispatch() method is to find an appropriate handler for the request and feed it the request/response parameters
- To find the handler that matches the request, Spring goes through the registered implementations of the HandlerMapping interface
- The two main *HandlerMapping* implementations
  - *RequestMappingHandlerMapping* (which supports *@RequestMapping* annotated methods)
  - *SimpleUrlHandlerMapping* (which maintains explicit registrations of URI path patterns to handlers)

HANDLING THE REQUEST

- Now that Spring determined the handler for the request and the adapter for the handler, it's time to finally handle the request. *HandlerAdapter* helps the *DispatcherServlet* to invoke a handler mapped to a request, regardless of how the handler is actually invoked
- *HandlerAdapter* return a *ModelAndView* object to be rendered by the *DispatcherServlet*
- Note: the *HandlerAdapter* won't render the view, it just return the object to the *DispatchServlet*

PROCESSING ARGUMENTS AND RETURN VALUES

- Now it is the time to handle the request and process our business logic.
- Note that the controller methods do not usually take *HttpServletRequest* and *HttpServletResponse* arguments, but instead receive and return many different types of data, such as domain objects, path parameters
- Note that you are not required to return a *ModelAndView* instance from a controller method. You may return a view name, or a *ResponseEntity* or a POJO that will be converted to a JSON response.
- The *RequestMappingHandlerAdapter* makes sure the arguments of the method are resolved from the *HttpServletRequest.* Also, it creates the *ModelAndView* object from the method's return value

RENDERING THE VIEW

- By now, Spring has processed the HTTP request and received a *ModelAndView* object, so it has to render the HTML page that the user will see in the browser. It does that based on the model and the selected view encapsulated in the *ModelAndView* object.

- In the *ModelAndView* object, we will specify the view name, such as *login.jsp,* so Spring has to look it up. This is where the V*iewResolvers* list comes into play

- Then Spring will call the view's *render()* method — Spring finally completes the request processing by sending the HTML page to the user's browser

CONTROLLER

- Spring MVC provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling

```
@Controller
public class HelloController {


    @GetMapping("/hello")
    public String handle(Model model) {
        model.addAttribute("message", "Hello World!");
        return "index";
    }
}
```

REQUEST MAPPING

- The *@RequestMapping* annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types

- It can be used at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

REQUEST MAPPING

- There are also HTTP method specific shortcut variants of *@RequestMapping*
  - *@GetMapping*
  - *@Postmaping*
  - *@PutMapping*
  - *@DeleteMapping*
  - *@PatchMapping*

REQUEST MAPPING

- Path Variable
  - *@PathVariable* — URI variables are automatically converted to the appropriate type, or *TypeMismatchException* is raised.
- Simple types (int, long, Date, and so on) are supported by default and you can register support for any other data type.

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

REQUEST MAPPING

- Request Parameter
  - @*RequestParam* — bind Servlet request parameters (that is, query parameters or form data) to a method argument in a controller
  - Type conversion is automatically applied if the target method parameter type is not String
  - By default, method parameters that use this annotation are required

```
@GetMapping
public String setupForm(@RequestParam("petId") int petId, Model model) {
    Pet pet = this.clinic.loadPet(petId);
    model.addAttribute("pet", pet);
    return "petForm";
}
```

REQUEST MAPPING

- Difference Between @RequestParam and @PathVariable in Spring MVC

  - The key difference between @*RequestParam* and @*PathVariable* is that @*RequestParam* used for accessing the values of the query parameters where as @*PathVariable* used for accessing the values from the URI template.

- Spring MVC provides support for customizing the URL in order to get data. To achieving this purpose @PathVariable annotation is used in Spring mvc framework.

REQUEST MAPPING

- *@RequestHeader* annotation is used to bind a request header to a method argument in a controller

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding, 1
        @RequestHeader("Keep-Alive") long keepAlive) { 2
    //...
}
```

REQUEST MAPPING

- The *@RequestBody* annotation is used to have the request body read and deserialized into an Object

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

REQUEST MAPPING

- The @ResponseBody annotation is used on a method to have the return serialized to the response body.

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

ANY QUESTIONS?