Spring MVC

- * The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- * The Model encapsulates the application data and in general they will consist of POJO.
- * The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- * The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.
- * Thus MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

Request Mappings

- * RequestMappings are really flexible
- You can define a @RequestMapping on a class and all methods
- * @RequestMappings will be relative to it.
- * There are a number of ways to define them:
 - * URI Patterns
 - * HTTP Methods (GET, POST, etc)
 - * Request Parameters
 - * Header values

RequestMapping - Class Level

```
package com.ameya.controllers;
```

```
@RequestMapping ("/portfolio")
@RestController
public class PortfolioController {
          @RequestMapping("/create")
          public String create() {
                return "create";
          }
}
```

The URL for this (relative to your context root)would be: /portfolio/create

3

RequestMapping - HTTP Methods

```
package com.ameya.controllers;
```

```
@RequestMapping("/portfolio")
@RestController
public class PortfolioController {
          @RequestMapping(value="/create",method=RequestMethod.POST)
          public String save() {
               return "view";
          }
}
```

* Same URL as the previous example, but responds to POSTs

RequestMapping - Request Params

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@RestController
public class PortfolioController {
  @RequestMapping(value="/view",params="details=all")
  public String viewAll() {
  return "viewAll";
  }
}

* This will respond to
  /portfolio/view?details=all
```

RequestMapping - URI Templates

Spring REST

37

Introduction to REST

REST stands for Representational State Transfer

- * It is an architectural *pattern* for developing web services as opposed to a *specification*.
- * REST web services communicate over the HTTP specification, using HTTP vocabulary:
 - * Methods (GET, POST, etc.)
 - * HTTP URI syntax (paths, parameters, etc.)
 - * Media types (xml, json, html, plain text, etc)
 - * HTTP Response codes.

Introduction to REST

- * Representational
 - * Clients possess the information necessary to identify, modify, and/or delete a web resource.
- * State
 - * All resource state information is stored on the client.
- * Transfer
 - * Client state is passed from the client to the service through HTTP.

Introduction to REST

The six characteristics of REST:

- Uniform interface
- 2. Decoupled client-server interaction
- 3. Stateless
- 4. Cacheable
- 5. Layered
- 6. Extensible through code on demand (optional)
- * Services that do not conform to the above required contstraints are not strictly RESTful web services.

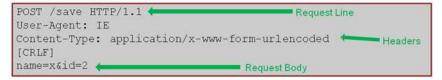
HTTP-REST Request Basics

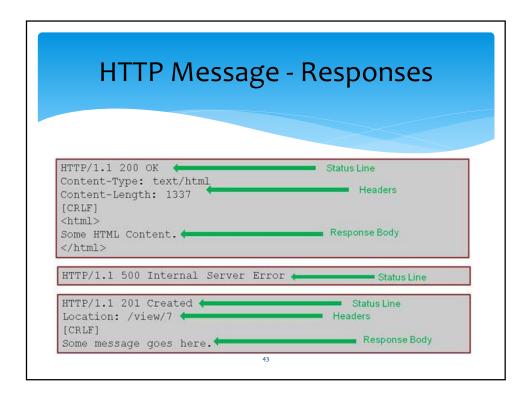
- * The **HTTP request** is sent from the client.
 - * Identifies the location of a resource.
 - * Specifies the **verb**, or HTTP **method** to use when accessing the resource.
 - * Supplies optional **request headers** (name-value pairs) that provide additional information the server may need when processing the request.
 - * Supplies an optional **request body** that identifies additional data to be uploaded to the server (e.g. form parameters, attachments, etc.)

HTTP Message

* What does an HTTP message look like?

```
GET /view/1 HTTP/1.1 RequestLine
User-Agent: Chrome
Accept: application/json
[CRLF]
```





RequestBody

 Annotating a handler method parameter with @RequestBody will bind that parameter to the request body

@RequestMapping("/echo/string")
public void writeString(@RequestBody String input) { }

@RequestMapping("/echo/json")
public void writeJson(@RequestBody SomeObject input) { }

ResponseBody

 Annotating a return type with @ResponseBody tells
 Spring MVC that the object returned should be treated as the response body

```
@RequestMapping("/echo/str±ng")
public @ResponseBody String readString(){}
```

@RequestMapping("/echo/json")
public @ResponseBody SomeObject readJson() { }

45

Other parts of the HttpMessage

- * What if you need to get/set headers?
- * Or set the status code?

@ResponseStatus

* There is a convenient way to set what the default status for a particular handler should be

```
@RequestMapping("/create")
@ResponseStatus(HttpStatus.CREATED) // CREATED = 201
public void echoString(String input) {
}
```

47

Spring REST Template

RestTemplate

- * RestTemplate communicates with HTTP server using RESTful principals.
- * RestTemplate provides different methods to communicate via HTTP methods.
- * This class provides the functionality for consuming the REST Services in a easy and graceful manner.
- * When using the said class the user has to only provide the URL, the parameters(if any) and extract the results received.
- * The RestTemplate manages the HTTP connections.

49

RestTemplate Methods

HTTP method	RestTemplate methods
DELETE	<pre>delete(java.lang.String, java.lang.Object)</pre>
GET	<pre>getForObject(java.lang.String, java.lang.Class<t>, java.lang.Object)</t></pre>
	<pre>getForEntity(java.lang.String, java.lang.Class<1>, java.lang.Object)</pre>
POST	<pre>postForLocation(java.lang.String, java.lang.Object, java.lang.Object)</pre>
	<pre>postForObject(java.lang.String, java.lang.Object, java.lang.Class<t>, java.lang.Object)</t></pre>
PUT	<pre>put(java.lang.String, java.lang.Object, java.lang.Object.50.)</pre>

HTTP GET Using RestTemplate

RestTemplate restTemplate = new RestTemplateO;

String url ="http://localhost:8080/demo/rest/employees/{id};

Map<String, String> map = new HashMap<String,String>();
map.put("id", "101");

ResponseEntity<Employee> entity =restTemplate.getForEntity(url, Employee.class, map);

System. out.println(entity.getBody());

HTTP POST Using RestTemplate

RestTemplate restTemplate = new RestTemplate();

String url ="http://localhost:8080/demo/rest/employees";

Employee employee =restTemplate.postForObject(url,
newEmployee,Employee.class);

System.out.println(employee);

SPRING BOOT and MICROSERVICES

Ameya J. Joshi

Email: ameya.joshi_official@outlook.com

Cell No: +91 9850676160

SPRING BOOT

Spring Framework Limitations

- * Huge framework
- * Multiple setup steps
- * Multiple configuration steps
- * Multiple Build and Deploy steps
- * Can We abstract these all steps?

What is Spring Boot?

* Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

What is Spring Boot?

- * Opinionated (It makes certain assumptions)
- * Convention Over Configuration
- * Stand alone
- * Production ready
- * Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

Features

- * Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- * Provide opinionated 'starter' POMs to simplify your Maven configuration
- * Automatically configure Spring whenever possible
- * Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Setup Spring Boot

- * Pre-requisites
 - * Hardware
 - * Core i5 machine
 - * 8 gb ram
 - * Software
 - * 64 bit Windows 7/10
 - * Java 1.8 or higher
 - * Spring Tools Suite (We use sts 4.x)

Setup Spring Boot

- * Install Maven
 - * Set MAVEN_HOME
 - * Add it to PATH Environment Variable
 - * Run mvn –version from command prompt to ensure maven is installed

Setup Spring Boot

- * Start STS
- * Create new Maven Project
 - * In the STS UI
 - * Check Create a simple project
 - * Click on next
 - * Enter Group Id (com.ameya)
 - * Enter Artifact Id (course-api)
 - Enter Version (Keep default)
 - * Enter Name (Ameya Joshi Course Api)

Setup Spring Boot

```
* Add following in pom.xml,
```

Few more dependencies.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derby</artifactId>
<scope>runtime</scope>
</dependency>
```

Writing First App

Type following code and run as java application

```
package com.ameya;
```

```
import\ org. spring framework. boot. Spring Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Spring Boot Application; import\ org. spring framework. boot. autoconfigure. Boot. Boot.
```

Behind the Scenes

SpringApplication.run(CourseApiApp.class, args);

This runs the CourseApiApp class

This class is annotated with @SpringBootApplication

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan

Behind the Scenes

- * As a result Spring Boot:
- * Sets up the default configuration
- * Starts Spring application context
- * Performs classpath scan
- * Starts tomcat server