Pivotal

Web Applications with Spring Boot

Getting Started with Spring MVC



Objectives

After completing this lesson, you should be able to

- Explain how to create a Spring MVC application using Spring Boot
- Describe the basic request processing lifecycle for REST request
- Create a simple RESTful controller to handle GET requests
- Configure for WAR or JAR deployment

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Web Layer Integration

- Spring Boot can help you create web applications easily
 - Based on Spring MVC
- What is Spring MVC?
 - Web framework based on:
 - Model/View/Controller (MVC) pattern
 - POJO programming
 - Testable components
 - Uses Spring for configuration
 - Supports Server-side web rendering & REST

We will focus on REST.

Types of Spring MVC Applications

- Web Servlet
 - Traditional approach
 - Based on Java EE Servlet
 Specification
 - Servlets
 - Filters
 - Listeners
 - Etc.

- Web Reactive
 - Newer, more efficient
 - Non-blocking approach
 - Netty, Undertow, Servlet 3.1+
 - Requires knowledge of Reactive programming (see appendix)

This section will focus on traditional servlet-based approach.

Traditional or Embedded Container

- Spring Boot supports embedded servlet containers
 - Packaging embedded containers within deployment artifact
 - Run web applications from command line!
 - Can be done using 'Fat' JAR

- Spring Boot can also implement traditional structure needed by Web containers
 - WAR packaging

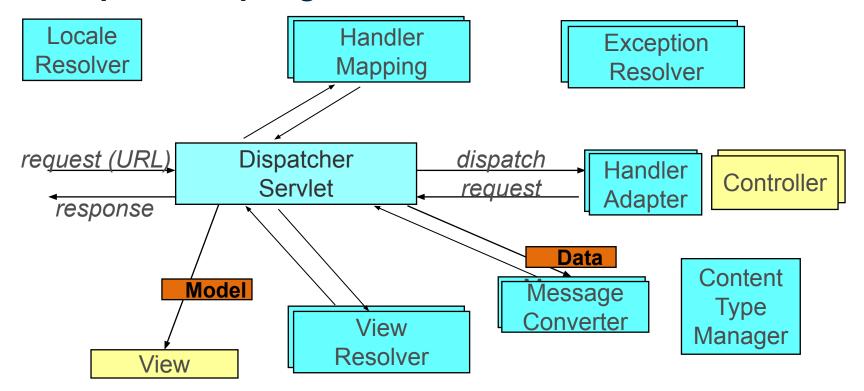
We will focus on embedded approach

Spring Web "Starter" Dependencies

Everything you need to develop a Spring MVC application

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
                                          Resolves
                                           spring-web.jar
                                           spring-webmvc.jar
                                           spring-boot-starter.jar
                                           jackson*.jar
                                           tomcat-embed*.jar
                                           hibernate-validator.jar
```

At Startup Time, Spring Boot Creates



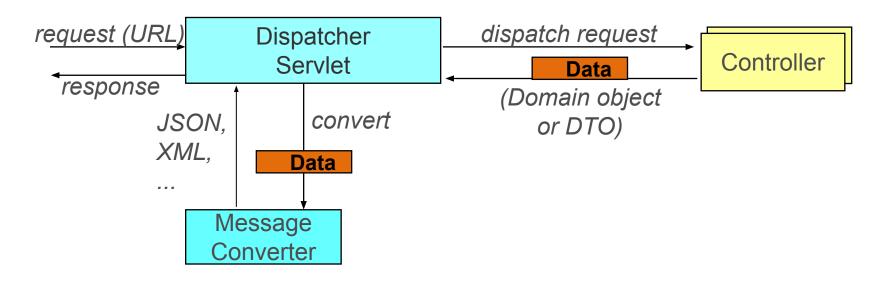
Most of these you will not need to work with for basic usage.

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Request Processing Lifecycle - REST



- Common message-converters setup automatically if found on the classpath
 - Jackson for JSON/XML, JAXB for XML, GSON for JSON ...

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Controller Implementation

- Controllers are annotated POJOs.
 - @GetMapping tells Spring what method to execute when processing a particular HTTP GET request
 - @ResponseBody defines a REST response
 - Turns off the View handling subsystem

```
@Controller is an @Component
@Controller
                                       so will be found by component-scanner
public class AccountController {
  @GetMapping("/accounts")
  public @ResponseBody List<Account> list() {...}
          Example of calling URL: http://localhost:8080 / accounts
```

@RestController Convenience

- Convenient "composed" annotation
 - Incorporates @Controller and @ResponseBody
 - All GET methods assumed to return REST response-data

```
@RestController
public class AccountController {

@GetMapping("/accounts")
public List<Account> list() {...}
}

No need for @ResponseBody

@Controller
@ResponseBody
...
public @interface RestController
```

All examples assume @RestController from now on

URL-Based Mapping Rules

- Mapping rules typically URL-based, optionally using wildcards:
 - /accounts
 - /accounts/1234567
 - /accounts/1234567/deposit
 - /accounts/*
 Wildcard *

Controller Method Arguments

- Extremely flexible!
- You pick the arguments you need, Spring injects them
 - HttpServletRequest, HttpSession, Principal ...
 - See <u>Spring Reference</u>, <u>Handler Methods</u>

```
@RestController public class AccountController {

@GetMapping("/accounts") public List<Account> list(Principal owner) {

...

Injected by Spring
```

Extracting Request Parameters

- Use @RequestParam annotation
 - Extracts parameter from the request
 - Performs type conversion

```
@RestController
public class AccountController {
    @GetMapping("/account")
    public List<Account> list(@RequestParam("userid") int userId) {
        ... // Fetch and return accounts for specified user
    }
}

Example of calling URL:
```

http://localhost:8080/account?userid=1234

Pivotal

URI Templates

- Values can be extracted from request URLs
 - Based on URI Templates
 - not Spring-specific concept, used in many frameworks
 - Use {...} placeholders and @PathVariable

```
@RestController
public class AccountController {
  @GetMapping("/accounts/{accountId}")
  public Account find(@PathVariable("accountId") long id) {
    ... // Do something
     Example of calling URL: http://localhost:8080/accounts/98765
```

17

Naming Conventions

- Drop annotation value if it matches method parameter name
 - Provided class contains method parameter names



https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html

Method Signature Examples

Pivota

Example URLs

```
http://localhost:8080/accounts
@GetMapping("/accounts")
public List<Account> getAccounts()
                                             http://.../orders/1234/items/2
@GetMapping("/orders/{id}/items/{itemId}")
public OrderItem item( @PathVariable("id") long orderId,
                       @PathVariable int itemId,
                       Locale locale.
                       @RequestHeader("user-agent") String agent )
                                          http://.../suppliers?location=12345
@GetMapping("/suppliers")
public List<Supplier> getSuppliers(
        @RequestParam(required=false) Integer location,
        Principal user,
                                                    Null if not specified
        HttpSession session )
                                                                            19
```

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



HTTP GET: Fetch a Resource

- Requirement
 - Respond only to GET requests
 - Return requested data in the HTTP Response
 - Determine requested response format

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json, ...

"id": 123,
"total": 200.00,
"items": [ ... ]

Pivotal
```

Generating Response Data







The Problem

- HTTP GET needs to return data in response body
 - Typically JSON or XML
- Developers prefer to work with Java objects
- Developers want to avoid manual conversion

The Solution

- Object to text conversion
 - Message-Converters
- Annotate response data
 with @ResponseBody

```
HTTP/1.1 200 OK
Date:
Content-Length: 756
Content-Type: application/json
   "id": 123,
   "total": 200.00,
   "items": [ ... ]
```

HttpMessageConverter







- Converts HTTP request/response body data
 - XML: JAXP Source, JAXB2 mapped object*, Jackson-Dataformat-XML*
 - GSON*, Jackson JSON*
 - Feed data* such as Atom/RSS
 - Google protocol buffers*
 - Form-based data
 - Byte[], String, BufferedImage
- Automatically setup by Spring Boot (except protocol buffers)
 - Manual configuration also possible

* Requires 3rd party libraries on classpath

What Return Format? Accept Header

```
@GetMapping("/store/orders/{id}")
public Order getOrder(@PathVariable("id") long id) {
    return orderService.findOrderById(id);
                                        HTTP/1.1 200 OK
                                        Date: ...
 GET /store/orders/123
                                        Content-Length: 1456
                                        Content-Type: application/xml
 Host: shop.spring.io
                                        <order id="123">
 Accept: application/xml
                                        </orde HTTP/1.1 200 OK</pre>
                                               Date: ...
                                               Content-Length: 756
                                               Content-Type: application/json
       GET /store/orders/123
       Host: shop.spring.io
                                                  "id": 123,
       Accept: application/json
                                                  "total": 200.00,
                                                  "items": [ ... ]
Pivota
```

Customizing GET Responses: ResponseEntity

- Build GET responses explicitly
 - More control
 - Set headers, control content returned



Subclasses **HttpEntity** with fluent API

Setting Response Data

Can use HttpEntity to generate a Response

```
@GetMapping("/store/orders/{id}")
public HttpEntity<Order> getOrder(@PathVariable long id) {
  Order order = orderService.find(id);
  return ResponseEntity
                                 HTTP Status 200 OK
               .ok()
               .header("Last-Modified", order.lastUpdated())
               .body(order);
                                                    Set extra or
                            Response
                               body
                                                   custom header
```

Pivotal.

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Spring Boot for Web Applications

- Use spring-boot-starter-web
 - Ensures Spring Web and Spring MVC are on classpath
- Spring Boot AutoConfiguration
 - Sets up a DispatcherServlet
 - Sets up internal configuration to support controllers
 - Sets up default resource locations (images, CSS, JavaScript)
 - Sets up default Message Converters
 - And much, much more



Spring Boot as a Runtime

- By default Spring Boot starts up an <u>embedded</u> servlet container
 - You can run a web application from the command line!
 - Tomcat included by Web Starter



Spring Boot's default behavior. Traditional WAR deployment available also.

Alternative Containers: Jetty, Undertow

Example: Jetty instead of Tomcat

```
<dependency>
 <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
                                                 Excludes Tomcat
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
 </exclusions>
                                                   Adds Jetty
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
                            Jetty automatically detected and used!
```

Running Within a Web Container (traditional)

```
Sub-classes Spring's WebApplicationInitializer

    called by the web container (Servlet 3+ required)

@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    // Specify the configuration class(es) to use
    protected SpringApplicationBuilder configure(
             SpringApplicationBuilder application) {
         return application.sources(Application.class);
                                    Don't forget to change artifact type to war
```



The above requires **no** web.xml file

Configure for a WAR or a JAR

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
                                                             Web container
    protected SpringApplicationBuilder configure(
                                                                support
             SpringApplicationBuilder application) {
        return application.sources(Application.class);
                                                            main() method
                                                              run from CLI
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
```

JAR or WAR

- By default, both JAR and WAR forms can run from the command line
 - Based on embedded Tomcat (or Jetty) JARs
- However, embedded Tomcat JARs can cause conflicts when running WAR inside traditional web container
 - Example: running within different version of Tomcat
- Best Practice: Mark Tomcat dependencies as provided when building WARs for traditional containers.

Spring Boot JAR Files

 Using Boot's plugin, mvn package or gradle assemble produces two JAR files

```
22M yourapp-0.0.1-SNAPSHOT.jar

10K yourapp-0.0.1-SNAPSHOT.jar.original

Traditional JAR
```

- "Fat" JAR executable with embedded Tomcat
 - using "java -jar yourapp.jar"



For details: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
#build-tool-plugins-maven-packaging

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Spring Boot Developer Tools

- A set of tools to help make Spring Boot development easier
 - Automatic restart any time a class file changes (on re-compile) - faster than "cold restart", however
 - Automatically disabled when it considers the app is running in "production"

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Quick Start
- Lab



Revision: What We Have Covered

- Spring Web applications have many features
 - The DispatcherServlet
 - Setup Using Spring Boot
 - Writing a Controller
 - Using Message Converters
 - JARs vs WARs

- But you don't need to worry about most of this to write a simple Spring Boot Web application ...
 - Typically you just write Controllers
 - Set some properties

Quick Start

Only a few files to get a running Spring Web application

Gradle or Ant/Ivy also supported pom.xml Setup Spring Boot (and any other) dependencies RewardController class Basic Spring MVC controller application.properties Setup Application class

Application launcher

1a. Maven Descriptor

```
Parent POM
<parent>
   <groupId>org.springframework.boot
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.2.2.RELEASE
</parent>
<dependencies>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
                                     Spring MVC, Embedded
</dependencies>
                                       Tomcat. Jackson ...
<!-- Continued on next slide -->
                                               pom.xml
```

1b. Maven Descriptor (continued)

Will also use the Spring Boot plugin

```
pom.xml
                                                  (continued)
<!-- Continued from previous slide -->
<build>
   <plugins>
       <plugin>
            <groupId>org.springframework.boot
            <artifactId>spring-boot-maven-plugin</artifactId>
       </plugin>
                                    Makes "fat" executable jars/wars
   </plugins>
</build>
```



Remember: Maven, Gradle, Ant/Ivy are all supported

1c. Spring Boot sets up Spring MVC

- As Spring web JARs are on classpath ...
 - Spring Boot sets up Spring MVC
 - Deploys a DispatcherServlet
- Typical URLs:

```
http://localhost:8080/accounts
http://localhost:8080/rewards
http://localhost:8080/rewards/1
```

We will implement the last example

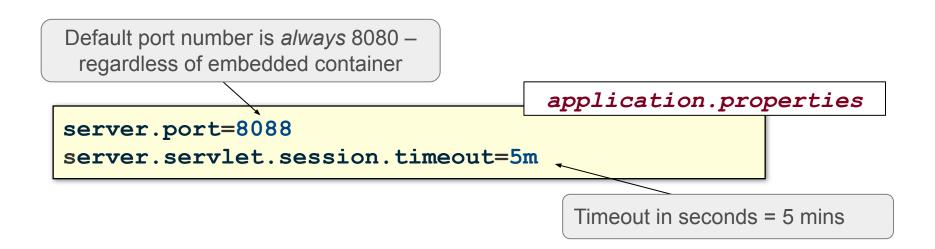
2. Implement the Controller

```
RewardController.java
@RestController
public class RewardController {
                                                    Depends on an
  private RewardLookupService lookupService;
                                                  application service
  public RewardController(RewardLookupService svc) {
    this.lookupService = svc;
                                                       Automatically
  @GetMapping("/rewards/{id}")
                                                     injected by Spring
  public Reward show(@PathVariable("id") long id) {
    return lookupService.lookupReward(id);
                                                   Returns Reward
```

Pivotal

3. Setting Properties

- Can configure web server using Boot properties
 - Not needed for this simple application
 - Let's set some common properties as examples



4. Application Class

- @SpringBootApplication annotation
 - Enables Spring Boot running Tomcat embedded

```
@SpringBootApplication
public class Application {

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

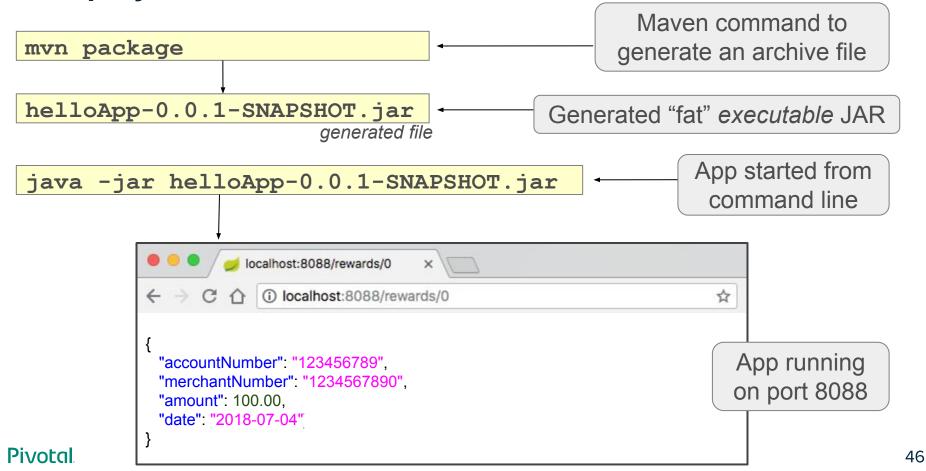
Application.java
```



Component scanner runs automatically, will find

RewardsController if it is located in same or sub-package

5. Deploy and Test



Summary

- Spring MVC is Spring's web framework
 - @Controller and @RestController handle HTTP requests
 - URL information available via @RequestParam,
 @PathVariable
- Spring Boot simplifies web application development
 - Embedded server, JAR and WAR packaging, Starters
- Multiple response formats supported
 - MessageConverters support JSON ,XML ...



Extensive documentation on Spring MVC available at:

https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#spring-web



Lab project: 36-mvc

Anticipated Lab time: 20 Minutes

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools
- Spring MVC Without Boot
- Lab



Servlet Configuration Without Spring Boot



- Servlets can be defined statically
 - Using web.xml (Servlet 2 spec)
 - Using a class that implements
 ServletContainerInitializer (Servlet 3+)
- Servlets can be defined dynamically
 - By registering them with the ServletContext as Servlet Container starts up (Servlet 3+)
 - Without Spring Boot you MUST do this

Configuring Spring MVC and DispatcherServlet

Sub-class Spring's *WebApplicationInitializer*– called by the web container (Servlet 3+ required)

```
public class MyWebInitializer implements WebApplicationInitializer {
    // Register the WebApplicationContext and DispatcherServlet
    public void onStartup(ServletContext servletCxt) {
        AnnotationConfigWebApplciationContext ac =
           new AnnotationConfigWebApplicationContext(AppConfig.class);
           DispatcherServlet servlet = new DispatcherServlet(ac);
           ServletRegistration.Dynamic registration =
               servletCxt.addServlet("app", servlet);
           registration.addMapping("/app/*");
```



Can also extend

AbstractAnnotationConfigDispatcherServletInitializer

A Note on @EnableWebMvc



- Without Spring Boot you MUST use @EnableWebMvc to get many of the same defaults Boot provides
 - Controllers and Views work "out-of-the-box"
 - REST support, message convertors, JSR-303 Validation
 - Stateless converters for form-handling, ...

```
@Configuration
@EnableWebMvc
public class AppConfig {
}
```

Pivotal. 52