Pivotal

Spring MVC with Spring Boot

Getting Started with Spring MVC

Objectives

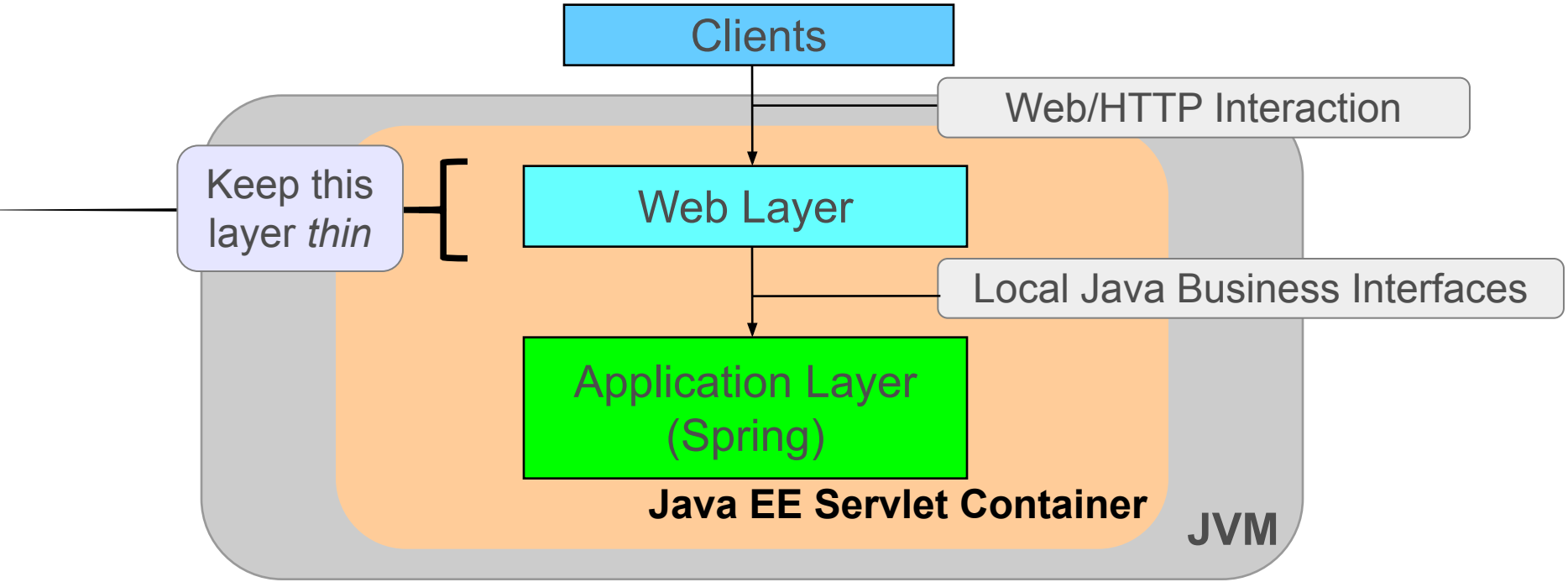
After completing this lesson, you should be able to

- Explain the Spring MVC & Request Processing Lifecycle
- Create a Simple RESTful controller to handle GET requests
- Configure Embedded Web Server using Spring Boot

Web Layer Integration

- Spring provides support in the Web layer
  - Spring MVC, REST
- However, you are free to use Spring with *any* Java web framework
  - Integration might be provided by Spring or by the other framework itself
  - Spring also integrates with many of the common REST frameworks

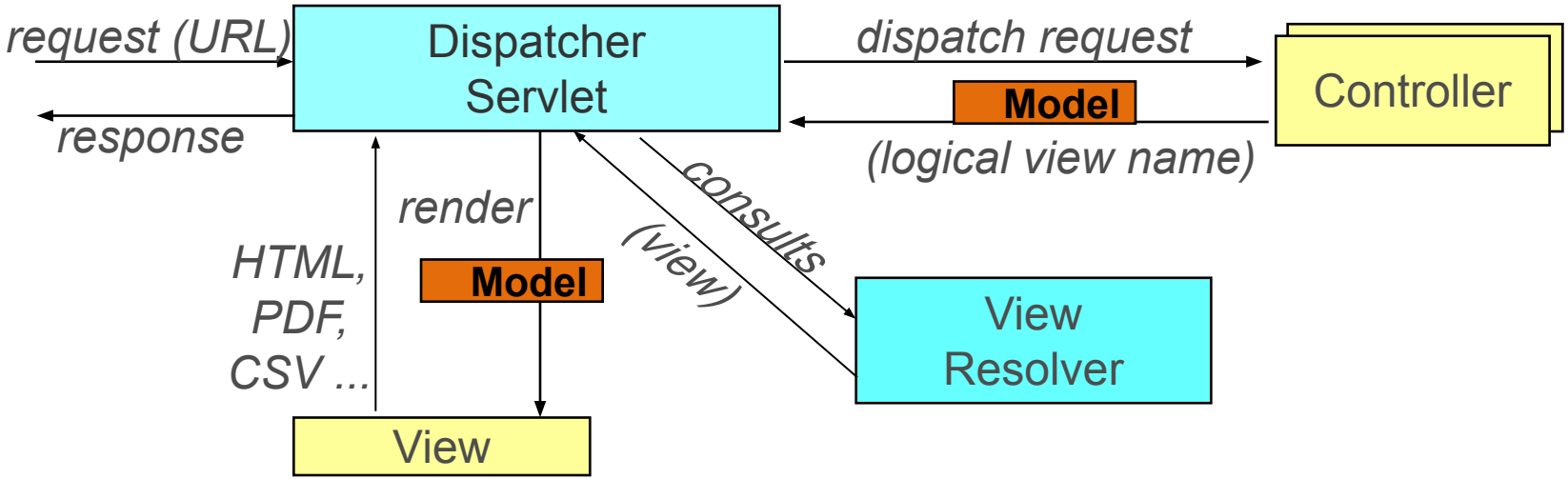
Effective Web Application Architecture



Agenda

- **Request Processing Lifecycle**
- Key Artifacts
- Spring Boot for Web Applications
- Quick Start
- Lab

Request Processing Lifecycle - Views



- You write the Controller classes and configure the right View Resolver
  - Most views are template files on disc that you also write
  - Popular view technologies: JSP, Mustache, Thymeleaf, Freemarker ...

What is Spring MVC?

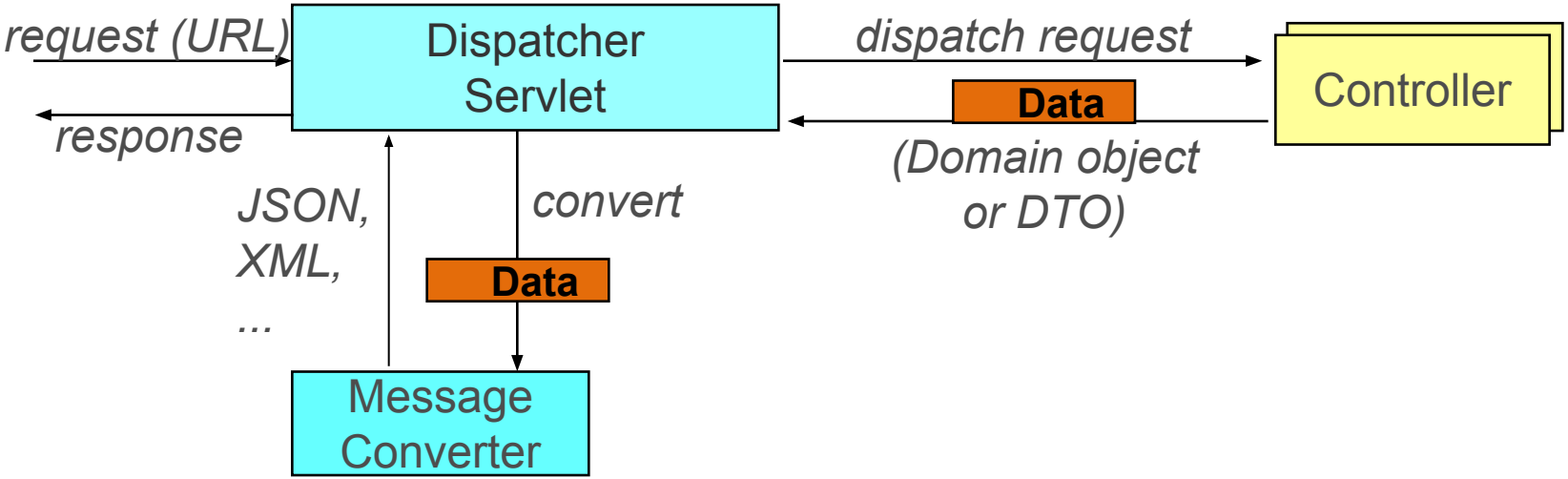
- Web framework
  - Based on *Model/View/Controller* pattern
- Based on Spring principles
  - POJO programming
  - Testable components
  - Uses Spring for configuration
- Basis for Spring's REST support
  - Multiple representations: JSON, XML, …
- And/or server-side rendering
  - JSP, XSLT, PDF, Excel, FreeMarker, Thymeleaf, Groovy Markup, Mustache …

Web Request Handling Overview

- Web request handling is rather simple
  - Based on an incoming URL...
  - ...we need to call a method...
  - ...after which the return value (if any)...
  - ...needs to be returned to the client

*The basis for generating both REST **and** HTML responses*

Request Processing Lifecycle - REST



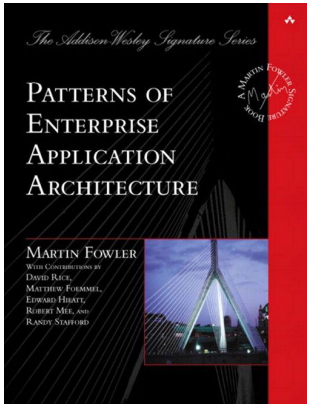
- Common message-converters setup automatically – *if* found on the classpath
  - Jackson for JSON/XML, JAXB for XML, GSON for JSON ...

Agenda

- Request Processing Lifecycle
- **Key Artifacts**
  - **Dispatcher Servlet**
  - Controllers
  - Message Converters
- Spring Boot for Web Applications
- Quick Start
- Lab

`DispatcherServlet` – The Heart of Spring Web MVC

- A “front controller”
  - Coordinates all request handling activities
  - Analogous to Struts ActionServlet / JSF FacesServlet
- Delegates to Web infrastructure beans
- Invokes user Web components
- Fully customizable
  - Interfaces for all infrastructure beans
  - Many extension points



Background: Servlet Configuration

- Servlets can be defined statically
  - Using **`web.xml`** (Servlet 2 spec)
  - Using a class that implements **`ServletContainerInitializer`** (Servlet 3+)

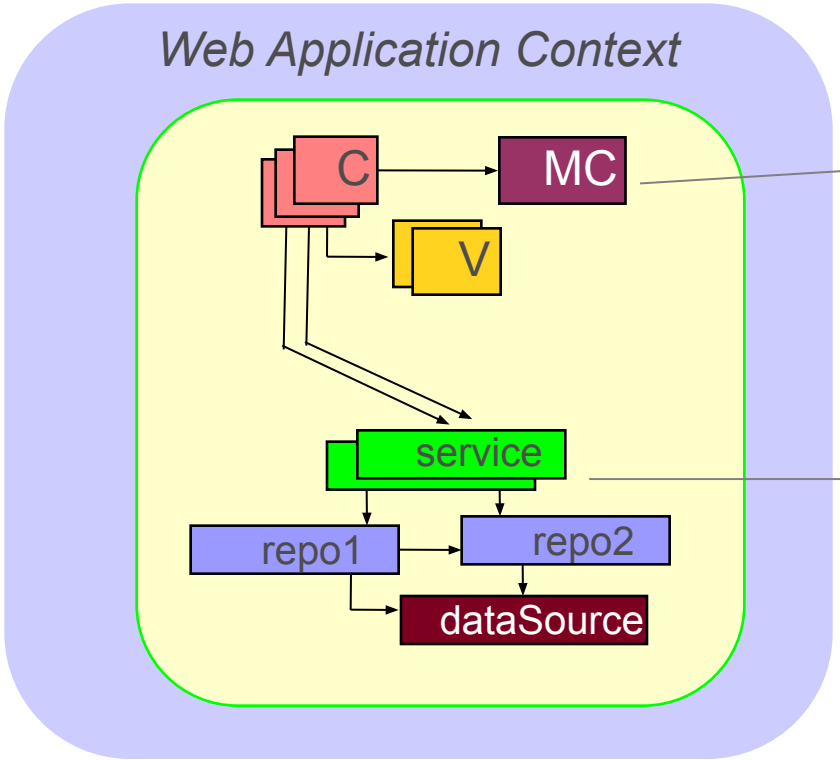
- Servlets can be defined dynamically
  - By registering them with the **`ServletContext`** as Servlet Container starts up (Servlet 3+)
  - *Spring Boot does this*

`DispatcherServlet` Configuration

- Automatically created and configured by Spring Boot
  - Uses a **`WebApplicationContext`**
    - Web specific, supports Session, Request scopes
    - Other servlet related artifacts
    - Contains both web-layer beans *and* application beans

- Web-layer beans also use Spring for their configuration
  - Programming to interfaces + dependency injection
  - Easy to swap parts in and out

Servlet Container After Starting Up



Web Beans

Controllers, Message-Converters, Interceptors ...

Application Beans

Business Services, Repositories, …

Controllers invoke methods on **Services**

Agenda

- Request Processing Lifecycle
- **Key Artifacts**
  - Dispatcher Servlet
  - **Controllers**
  - Message Converters
- Spring Boot for Web Applications
- 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
public class AccountController {

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

`@Controller` is an **`@Component`** so *will* be found by component-scanner

Example of calling URL: http://localhost:8080 / accounts

application server

request mapping

@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/*

Controller Method Parameters

- Extremely flexible!
- You pick the parameters you need, Spring injects them
  - **`HttpServletRequest`**, **`HttpSession`**, **`Principal`** …
  - See Spring Reference, Handler Methods

```
@RestController
public class AccountController {

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

Example: Find the accounts for currently logged in user

Data to return

Extracting Request Parameters

- Use **`@RequestParam`** annotation
  - Extracts parameter from the request
  - Performs any 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

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
  - Allows clean URLs without request parameters

```
@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

Using Naming Conventions

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

```
// Get user's overdrawn accounts
@GetMapping("/accounts/{userId}")
public List<Account> list(@PathVariable long userId,
                          @RequestParam boolean overdrawn)
```

http://.../accounts/1234?overdrawn=true

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

Method Signature Examples

Example URLs

http://localhost:8080/accounts

```
@GetMapping("/accounts")
public List<Account> list(HttpServletRequest request)
```

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<Customer> mySuppliers(
        @RequestParam(required=false) Integer location,
        Principal user,
        HttpSession session )
```

Null if not specified

Agenda

- Request Processing Lifecycle
- **Key Artifacts**
  - Dispatcher Servlet
  - Controllers
  - **Message Converters**
- Spring Boot for Web Applications
- 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, ...
...
```

```
HTTP/1.1 200 OK
Date: …
Content-Length: 756
Content-Type: application/json

{
    "id": 123,
    "total": 200.00,
    "items": [ … ]
}
```

Generating Response Data

- **The Problem**
  - HTTP GET needs to return data in response body
    - Typically XML or JSON
  - Prefer to work with Java objects
  - Avoid converting manually

- **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("/orders/{id}")
public Order getOrder(@PathVariable("id") long id) {
   return orderService.findOrderById(id);
}
```

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

```
HTTP/1.1 200 OK
Date: …
Content-Length: 1456
Content-Type: application/xml

<order id="123">
  …
</orde
```

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

```
HTTP/1.1 200 OK
Date: …
Content-Length: 756
Content-Type: application/json

{
   "id": 123,
   "total": 200.00,
   "items": [ … ]
}
```

Customizing GET Responses: ResponseEntity

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

```
// ResponseEntity supports a "fluent" API for creating a
// response. Used to initialize the HttpServletResponse
ResponseEntity<String> response =
      ResponseEntity.ok()
                    .contentType(MediaType.TEXT_PLAIN)
                    .body("Hello Spring");
```

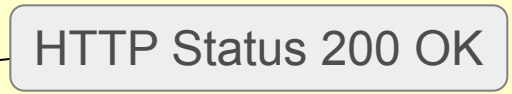
Subclasses **HttpEntity** with fluent API

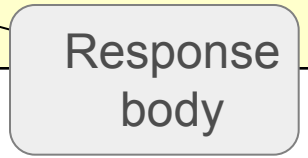
Setting Response Data

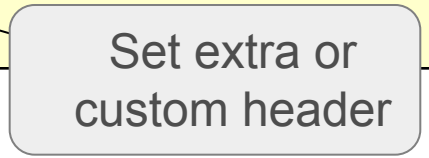
- Can use **`HttpEntity`** to generate a Response

```
@GetMapping("/orders/{id}")
public HttpEntity<Order> getOrder(@PathVariable long id) {
  Order order = orderService.find(id);

  return ResponseEntity
                .ok()
                .header("Last-Modified", order.lastUpdated())
                .body(order);
}
```







Agenda

- Request Processing Lifecycle
- Key Artifacts
- **Spring Boot for Web Applications**
- 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 *embedded* servlet container
  - You can run a web application from a JAR file!
  - Tomcat included by Web Starter

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot's default behavior. Simpler for running, testing and deploying *Cloud Native* applications. We will see WAR support soon.

Alternative Containers: Jetty, Undertow

- *Example:* Jetty instead of Tomcat

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Excludes Tomcat

Adds Jetty

Jetty automatically detected and used!

Spring Boot for Web Applications

- Boot automatically configures
  - A **`DispatcherServlet`**
  - Spring MVC using same defaults as **`@EnableWebMvc`**
- *Plus* many useful *extra features*:
  - Static resources served from classpath
    - /static, /public, /resources or /META-INF/resources
  - Default **`MessageSource`** for I18N
  - View templates served from /templates
    - If Groovy, Freemarker, Thymeleaf, Mustache ... on classpath
  - Provides default /error mapping
    - Can be overridden

A Note on `@EnableWebMvc`

- *Without* Spring Boot
  - Use **`@EnableWebMvc`** to get many of the same defaults
    - Controllers and Views work “out-of-the-box”
    - REST support, message convertors, JSR-303 Validation
    - Stateless converters for form-handling, …
- *With* Spring Boot
  - Boot assumes *you* want to control setup
  - Gives you *just* the configuration of **`@EnableWebMvc`**
  - Useful when porting legacy application to Spring Boot

Bottom Line: Most Spring Boot web applications *do not* specify **`@EnableWebMvc`**

Running as a WAR Application

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 {

    protected SpringApplicationBuilder configure(
            SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

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

WAR support

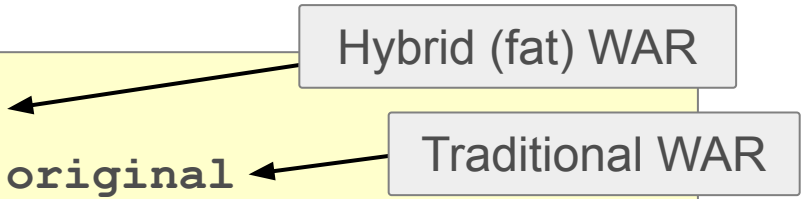
`main()` method for JAR to run

Warning: Mark `spring-boot-starter-tomcat` as "provided" (ignored by WAR)

Spring Boot WAR Files

- Using Boot's plugin, **`mvn package`** or **`gradle assemble`** produces *two* WAR files

```
22M  yourapp-0.0.1-SNAPSHOT.war
20M  yourapp-0.0.1-SNAPSHOT.war.original
```



- Hybrid "fat" WAR executable with embedded Tomcat
  - using "**`java -jar yourapp.war`**"

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

Agenda

- Request Processing Lifecycle
- Key Artifacts
- Spring Boot for Web Applications
- **Quick Start**
- Lab

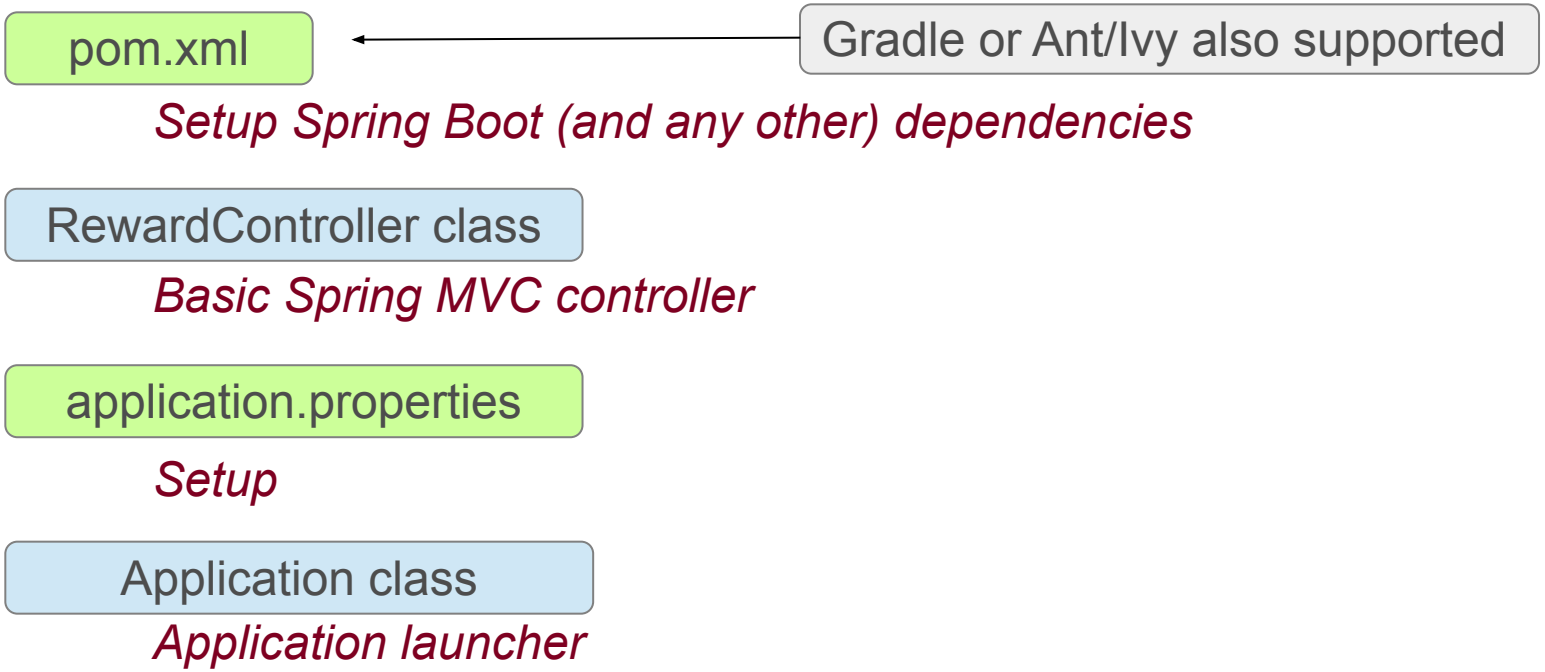
Revision: What We Have Covered

- 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



1a. Maven Descriptor

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<!-- Continued on next slide -->
```

Parent POM

Spring MVC, Embedded Tomcat, Jackson ...

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</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Makes “fat” *executable* jars/wars

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 {
    private RewardLookupService lookupService;

    @Autowired
    public RewardController(RewardLookupService svc) {
        this.lookupService = svc;
    }

    @GetMapping("/rewards/{id}")
    public Reward show(@PathVariable("id") long id) {
        return lookupService.lookupReward(id);
    }
}
```

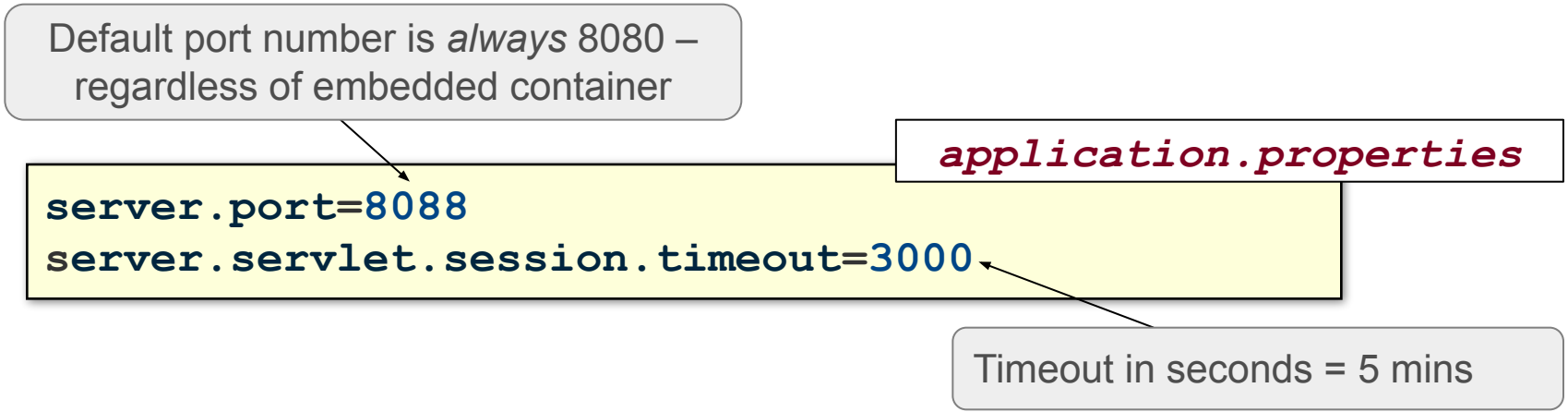
Depends on an application service

Automatically injected by Spring

Returns Reward

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`**

5. Deploy and Test

```
mvn package
```

Maven command to generate an archive file

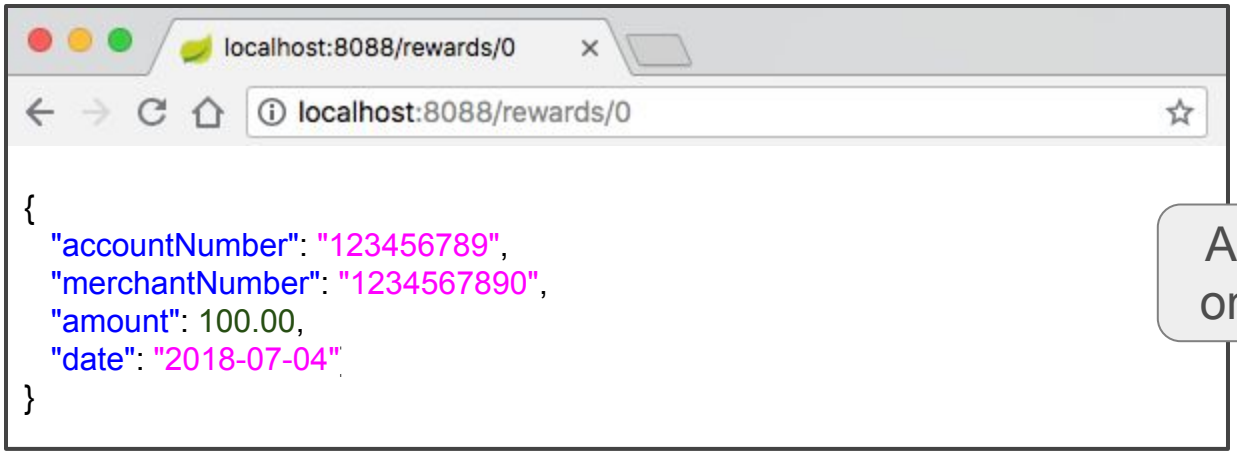
```
helloApp-0.0.1-SNAPSHOT.jar
```

generated file

Generated “fat” *executable* jar

```
java -jar helloApp-0.0.1-SNAPSHOT.jar
```

App started from command line



App running on port 8088

Summary

- Spring MVC is Spring's web framework
  - **`@Controller`** and **`@RestController`** classes handle HTTP requests
  - URL information available
    - **`@RequestParam`**, **`@PathVariable`**
- 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: Build a simple REST front end to expose Account data

Lab project:
36-mvc

Anticipated Lab time:
20 Minutes