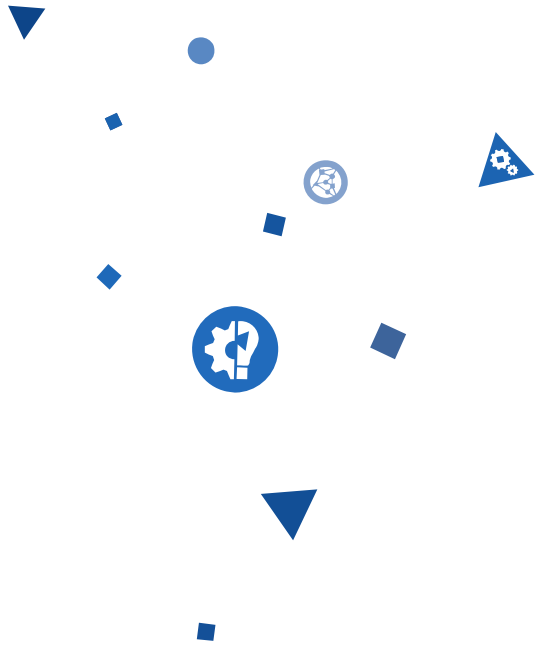


SPRING MVC

spring-mvc-5.2.7.RELEASE

spring-boot-2.3.1.RELEASE

INTRODUCTION

- Dvorzhetskiy Yuriy
- ydvorzhetskiy@luxoft.com

- http://luxoft-training.ru
- http://luxoft-training.com

TRAINING ROADMAP: OVERVIEW

Spring Boot	7
Spring MVC Overview	54
Controller	84
View	129
Request and session scopes	162

This training covers Spring MVC Framework.

The goal of this training is to get practice skills in Spring MVC.

This training is targeted to junior and regular Java developers.

Pre-requisites:

- Spring Framework
- Maven
- HTML, HTTP basics

TRAINING ROADMAP: STRUCTURE

- 8 Hour sessions
- 15-30 mins breaks every 1.5 – 2 hours
- Lunches (take your lunch card)
- In-class individual practice
- In-class group workshops
- Homework

SPRING MVC

- Spring MVC is a Web framework,
  based on Model-View-Controller architecture.
- Spring MVC is very lightweight.
- Spring MVC has simple and clean architecture.
- Simply extendable and customizable.
- Very popular, a lot of samples, easy documentation.

SPRING BOOT

- Spring Boot is a special framework for easy creation stand-alone, production-grade Spring-based applications.

- Automatically configures your Spring applications.
- Can be used with Spring MVC or without it.
- And Spring MVC works fine without Spring Boot.

SECTION 1:
SPRING BOOT

SPRING BOOT: INTRODUCTION

- What is Spring Boot?
  - Wrapper of Spring frameworks
  - Wrapper of many features
  - Special Maven/Gradle environment
- What is Spring Boot for?
  - To accelerate and facilitate application development
- Spring Boot site: http://projects.spring.io/spring-boot/

SPRING BOOT: FEATURES

- Create stand-alone applications
- Embed Tomcat, Jetty, Undertow servers in a JAR
- Provide opinionated 'starter' POMs to simplify Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- No code generation and no requirement for XML configuration

SPRING BOOT: INTRODUCTION

- Where is Spring Boot useful?
  - Startup of Spring project
  - Projects based on Microservice Architecture
  - Huge projects
  - Small projects, prototypes, Proof-of-Concept, test/mock applications

MICROSERVICE ARCHITECTURE

- Each business function (feature, component) is a separate service.
- Lightweight communication (REST or simple messaging).
- Decentralized governance, deploy, data management.
- Infrastructure automation: autotests, autodeploy, monitoring.
- Spring Boot provides or helps to develop most of these points.

- Advantages:
  - Services are easy to replace.
  - Services can be implemented using different languages, technologies, databases, hardware and software.
  - Natural decoupling between services.
  - Easy horizontal scalability.

MICROSERVICE ARCHITECTURE

- http://martinfowler.com/articles/microservices.html
- Here you can read about:
  - design principles;
  - Two Pizza Team;
  - books and links;
  - and more other information.

SPRING BOOT: REQUIREMENTS

- Spring Boot 2.3.1 requirements

  Java 8+

  Maven 3.2+ or Gradle 2 (2.9+) or Gradle 3

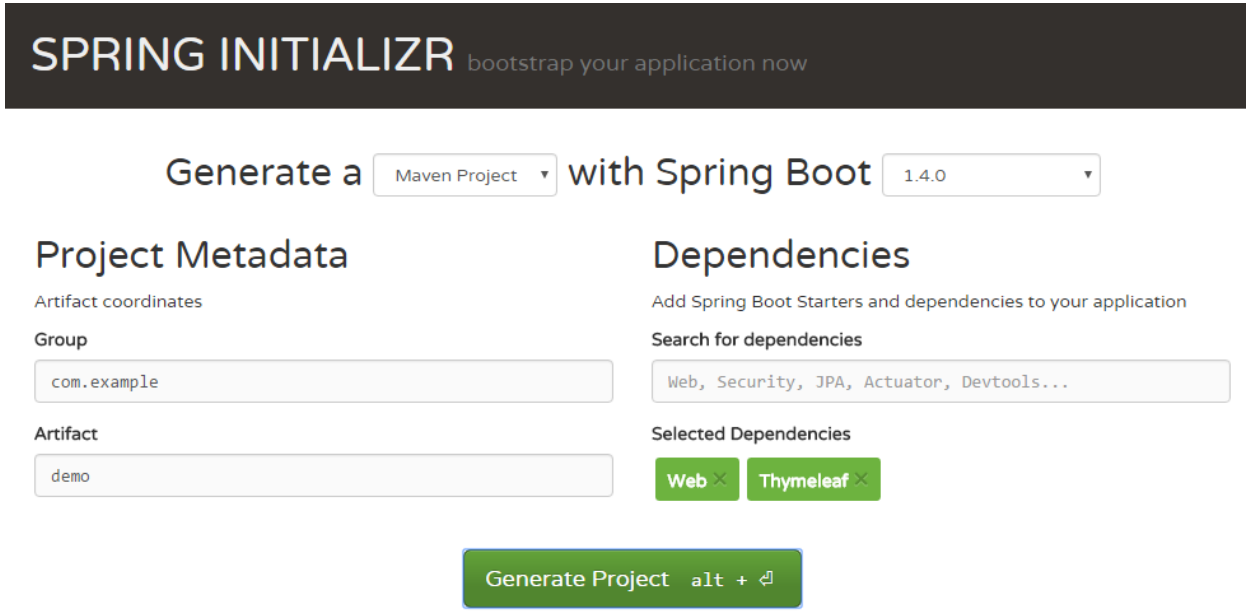
  Spring Framework 5.2.7.RELEASE or above

HOW TO CREATE

- Using Spring Initializr (http://start.spring.io/)
- Manual Maven/Gradle configuration (spring.io-recommendation):
  - add Spring Boot parent;
  - add Spring Boot dependency and plugin
- Using IDE

HOW TO CREATE: SPRING INITIALIZR

- http://start.spring.io/



HOW TO CREATE: MANUAL CONFIGURATION

```
<!-- Spring Boot 'starter' parent -->

<parent>
    <groupId>
        org.springframework.boot
    </groupId>
    <artifactId>
        spring-boot-starter-parent
    </artifactId>
    <version>
        2.3.1.RELEASE
    </version>
</parent>
```

```
<!-- Spring Boot dependencies -->

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

HOW TO CREATE: MANUAL CONFIGURATION

```
<!-- Spring Boot Maven plugin -->

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

- Manual Maven configuration of Spring Boot project is very simple.
- You have to specify only 'starter' parent, 'starter' dependency and Spring Boot maven plugin
- All versions are defined in the parent's POM

SPRING BOOT STARTER

- Spring Boot Starter
  - Special set of Maven/Gradle POMs
  - Simplify creating of Spring Boot applications
  - Contains a lot of default settings
  - Contains required dependencies
  - Spring Boot Starter is optional
- spring-boot-starter-*
  - spring-boot-starter /spring-boot-starter-parent
  - spring-boot-starter-aop
  - spring-boot-starter-jdbc
  - spring-boot-starter-web
  - spring-boot-starter-tx
  - …

SPRING BOOT DEPENDENCIES

- Parent

  common maven project settings

  dependencyManagement

  application.properties/.yaml filtering with @placeholder@

  settings for most usual maven-plugins (pluginManagement)

- You can specify your own parent

```
<parent>
    <groupId>
        org.springframework.boot
    </groupId>
    <artifactId>
        spring-boot-starter-parent
    </artifactId>
    <version>
        2.3.1.RELEASE
    </version>
</parent>
```

SPRING BOOT DEPENDENCIES

- spring-boot-starter

  (for all applications, transitive for web-applications)

  - spring-boot
  - spring-core
  - spring-boot-autoconfigure
  - spring-boot-starter-logging (logback is used instead of commons-logging)

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

SPRING BOOT DEPENDENCIES

- spring-boot-starter-web:

  (for web applications)

  spring-boot-starter and its dependencies

  spring-boot-starter-tomcat

  spring-boot-starter-validation

  jackson

  spring-web + spring-mvc

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

SPRING BOOT DEPENDENCIES

- spring-boot-starter-test:

  (test scope)

  junit

  mockito

  harmcrest

  spring-test

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

SPRING BOOT DEPENDENCIES

- spring-boot-plugin:

  creates Uber-JAR (JAR with all dependencies and embedded servlet container)

  add Main-Class entry to manifest

  automatically change version of dependencies to match Spring Boot dependencies (you can override its decision)

```
<plugin>
    <groupId>
        org.springframework.boot
    </groupId>
    <artifactId>
        spring-boot-maven-plugin
    </artifactId>
</plugin>
```

WAR VS. UBER-JAR

- By default Spring Boot applications are packed to Uber-JAR archive with all dependencies and embedded servlet container (Tomcat, Jetty, Undertow).
- Uber-JAR is simple to develop, distribute and deploy. But Uber-JAR packaging has several restrictions, such JSP support. If you use Uber-JAR you have to avoid all JSP pages.

- Uber-JAR packaging is optional. You can generate a classic WAR.

WAR PACKAGING

```
<project>
    <!-- ... -->
    <packaging>war</packaging>
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- ... -->
    </dependencies>
</project>
```

MAIN CLASS

- Main class:
  - entry point of application
  - additional initialization
  - main configuration annotations

```
@SpringBootApplication
public class DemoApplication {
    public static void main(
        String[] args
    ) {
        SpringApplication.run(
            DemoApplication.class, args
        );
        // additional initialization
    }
}
```

MAIN CLASS: ANNOTATIONS

- @SpringBootApplication is convenience equivalent of all the following annotations:

  @Configuration annotation of beans definition for application context.

  @EnableAutoConfiguration – tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

  @EnableWebMvc for Spring MVC support (Spring Boot adds it when it sees spring-mvc on the classpath)

  @ComponentScan – scan for beans is starting from current package.

CONFIGURATION

- Spring Boot uses Java-based configuration (but it is possible to configure it with deprecated XML files).
- You can place your configuration in special classes.
- @ComponentScan annotation on the main class will include it.

```
@Configuration
public class AppConfig {

  // Creates bean 'myService'
  @Bean
  public MyService myService(
    OtherService other
  ) {
    return new MyServiceImpl(other);
  }
}
```

CONFIGURATION: @CONDITIONAL*

- @Conditional* is the powerful feature of Spring Boot. Auto-configuration is based on it.
- With @Conditional* you can create a bean when:
  - The property is set/unset;
  - Class is present on the classpath;
  - Custom conditions;
  - And many other conditions.

```
@Bean
@ConditionalOnProperty(
    value = "create.socket",
    havingValue = "always"
)
@ConditionalOnClass(
    name = "OtherService"
)
@Conditional(CustomCondition.class)
public MyService myService(...) {...}
```

CONFIGURATION: PROPERTIES

- src/main/resources/application.properties:
  - Is empty in new project (all required properties are already defined).
  - You can place your application settings here.
  - This Maven resource is filtered.
  - New notation for Maven properties placeholders: @placeholder@.

```
spring.datasource.url=@db.url@
spring.datasource.username=user
spring.datasource.password=password
```

CONFIGURATION: YAML PROPERTIES

```
#application.properties

environments.dev.url
    =http://dev.bar.com
environments.dev.name
    =Developer Setup
environments.prod.url
    =http://foo.bar.com
environments.prod.name
    =My App
```

```
#application.yml

environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My App
```

HOW TO BUILD AND RUN

- Spring Boot application can be built using Maven or Maven-wrapper:

  mvn package

  or

  mvnw package

- Spring Boot application can be run:

  java -jar <your jar name>.jar

  double click on jar (don’t use this)

  mvn spring-boot:run

  or

  mvnw spring-boot:run

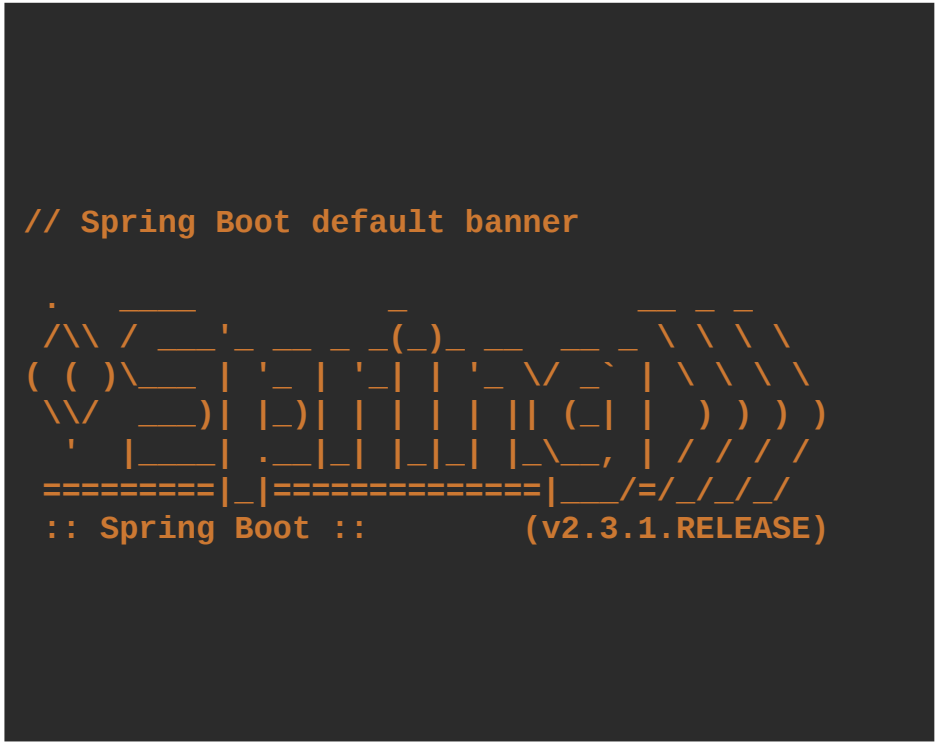
  run main class using IDE

BANNER

- With banner you can easily recognize the start of the service.
- You can show your banner:

  simply add your own banner.txt at src/main/resources

  or implement org.springframework.boot.Banner and set it programmaticaly SpringApplication.setBanner(...)



BANNER

- You can use in banner.txt:

  ${application.version} – "Implementation-Version" from MANIFEST.MF

  (by default it equals ${project.verision} from Maven POM)

  ${application.title} – "Implementation-Title" from MANIFEST.MF

  (by default it equals ${project.artifactId} from Maven POM)

  colors, Spring Boot version, etc.

- You can also disable banner.
- It will never be shown when MANIFEST.MF is unavailable (e.g. debug).

STATIC WEB RESOURCES

- You can place your static content (HTML, CSS, images) in

  src/main/resources/static

- Spring Boot automatically maps this resources

SPRING BOOT DEPENDENCIES

- spring-boot-devtools:

  test features, like H2 database

  LiveReload

```
<dependency>
    <groupId>
        org.springframework.boot
    </groupId>
    <artifactId>
        spring-boot-devtools
    </artifactId>
    <optional>true</optional>
</dependency>
```

DEBUG

- Debug mode in IDE
- Java command line args:

```
java -jar app-1.0.jar -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
```

- mvn spring-boot:run with command-line args:

```
mvn spring-boot:run -Drun.jvmArguments="-Xdebug -
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005"
```

DEBUG

- Spring Boot plugin settings
- To debug application:
  mvn spring-boot:run

```
<plugin>
  <groupId>
    org.springframework.boot
  </groupId>
  <artifactId>
    spring-boot-maven-plugin
  </artifactId>
  <configuration>
    <jvmArguments>
      -Xdebug -Xrunjdwp:transport=dt_socket,
      server=y,suspend=y,address=5005
    </jvmArguments>
  </configuration>
</plugin>
```

TESTING

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyIntegrationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void exampleTest() {
        String body = this.restTemplate.getForObject("/", String.class);
        Assert.assertTrue(body.contains("Hello, World!"));
    }
}
```

TESTING: @SPRINGBOOTTEST ENVIRONMENT

- WebEnvironment.MOCK – loads a WebApplicationContext and provides a mock servlet environment.
- WebEnvironment.RANDOM_PORT – loads an EmbeddedWebApplicationContext and provides a real servlet environment on the random port.
- WebEnvironment.DEFINED_PORT – loads an EmbeddedWebApplicationContext and provides a real servlet environment on the port from server.port property.
- WebEnvironment.NONE – does not provide any web environment.

TESTING: @MOCKBEAN

- @MockBean annotation creates mock bean, that replaces original bean.
- Test engine recreates this bean before each test method call.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Test
    public void exampleTest() {
        given(
            this.remoteService.someCall()
        ).willReturn("mock");
        // asserts
    }
}
```

PRODUCTION-READY FEATURES

- Spring Boot includes a number of additional features to help you monitor and manage your application when it's pushed to production:

  healthchecks;

  metrics;

  admin methods (shutdown);

  etc.

- These features are available in spring-boot-actuator subproject.

SPRING BOOT ACTUATOR

- spring-boot-starter-actuator
- Spring Boot Starter for spring-boot-actuator.

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

SPRING BOOT ACTUATOR: ENDPOINTS

- /actuator – "index" page – all actuator method URLs in JSON (spring-boot-starter-hateoas dependency is required)
- /beans – list of all beans in application context
- /env – application environment
- /configprops – list of all application configs
- /health – application health information
- /metrics – metrics

SPRING BOOT ACTUATOR: ENDPOINTS

- /mappings – mapping of controllers
- /liquibase – list of all database patches
- /logfile – "logging.file" or "logging.path" content
- /docs – application documentation (require spring-boot-actuator-docs)
- /info – custom application info
- /shutdown – shutdowns server (POST, disable by default)

HEALTHCHECKS

- Standard HealthIndicators:

  DataSourceHealthIndicator

  DiskSpaceHealthIndicator

  JmsHealthIndicator

  RedisHealthIndicator

  etc.

OTHER FEATURES

- Distributed transactions:

  spring-boot-starter-jta-atomikos

- Spring boot CLI

  runs Groovy scripts as web application

- Web Sockets

  spring-boot-starter-websocket

OTHER FEATURES

- Configuration through a config server

  spring-cloud-starter-config

- Configuration from a file outside of the app
- Colored console output
- Management via JMX

OTHER FEATURES

- Autoconfigurations (@Conditional*)
- Caching
  - Hazelcast
  - JCache
  - EhCache 2.x
  - and others

OTHER FEATURES

- SQL Databases support
- NoSQL databases support:
  - Redis
  - Couchbase
  - Cassandra
  - MongoDB
  - ElasticSearch

SPRING BOOT: SUMMARY

- What is Spring Boot
- Microservice Architecture
- Spring Initializr (start.spring.io)
- Spring Boot 'starter'
- Spring Boot dependencies
- Runnable Uber-JAR
- Main class
- Spring Boot Configuration
- @Conditional*
- Properties/YAML files
- Banner
- Debug/Unit Tests
- Production-ready features
- Other features overview

SPRING BOOT

QUESTIONS?

SPRING BOOT: EXERCISE

Exercise #1

- Creating your first Spring Boot application.

Discuss.

SPRING BOOT

QUESTIONS?

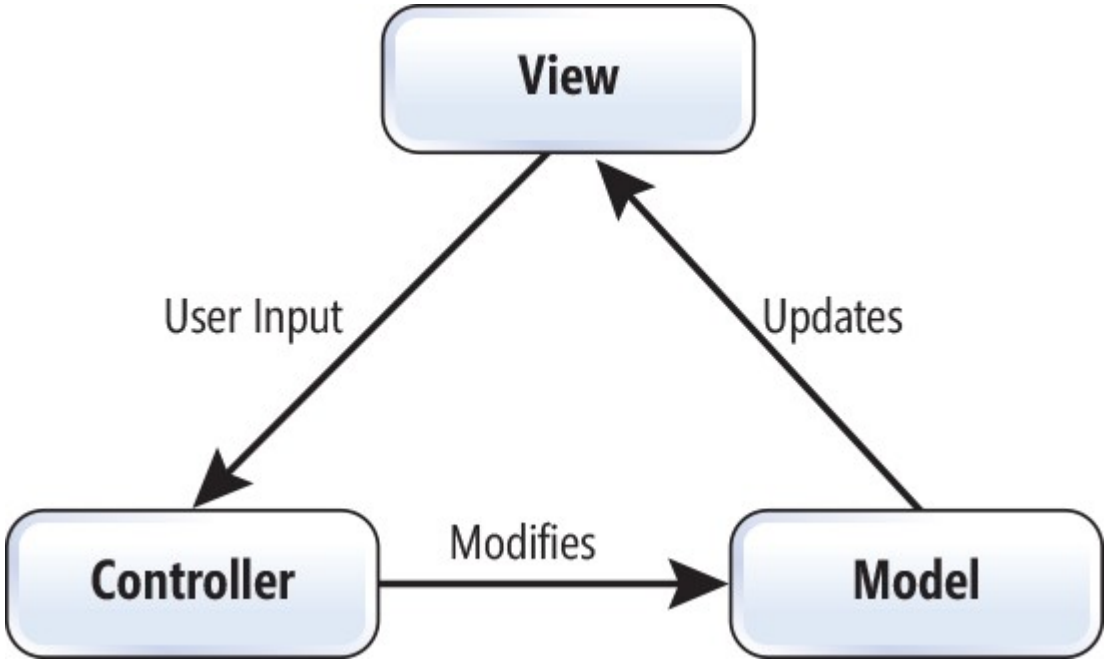
SECTION 2:
SPRING MVC OVERVIEW

SPRING MVC: OVERVIEW

- A web framework built around the principles of Spring
- POJO based and interface driven
- Based on a Dispatcher Servlet / Front Controller pattern
- Very lightweight compared to other frameworks

- Support for:
  - Themes
  - Locales/i18n
  - RESTful services
  - Annotation based configuration
  - Integration with other Spring Services/Beans

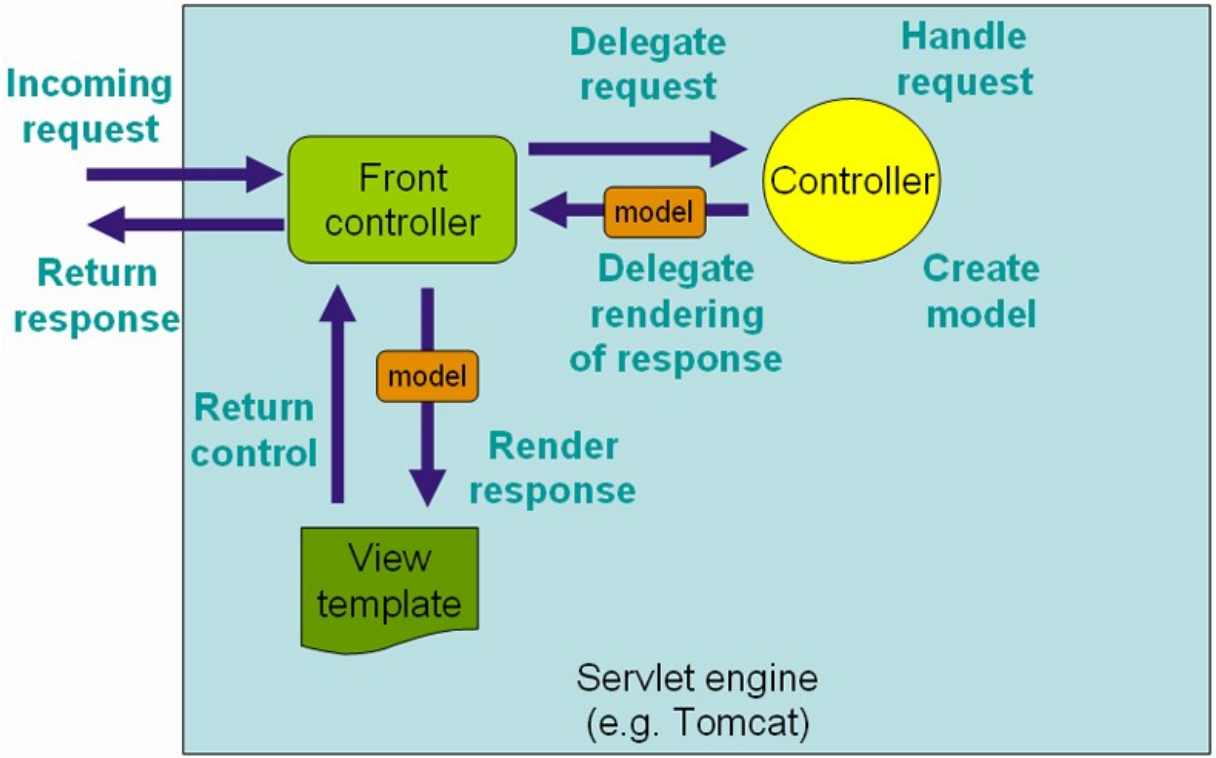
MVC ARCHITECTURE



MVC ARCHITECTURE

- The model represents application dynamic data and methods to work with it and change its state according to requests.
- The view is responsible for the information display (visualization). It can be any output representation of information.
- The controller provides communication between the user and the system. It controls user input and uses the model and the view to the implement correct response.

SPRING MVC ARCHITECTURE



SPRING MVC ARCHITECTURE: ADVANTAGES

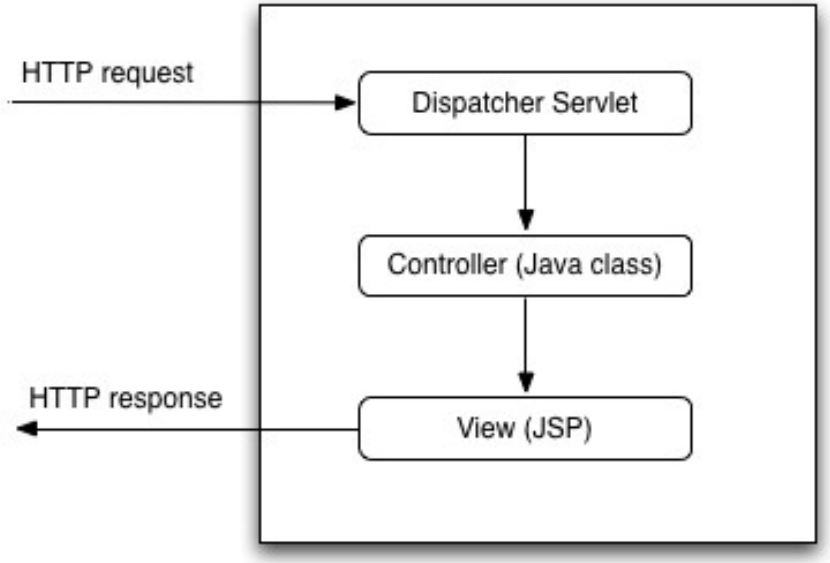
- Clean division between controllers, JavaBean models, and views.
- Spring's MVC is very flexible.
- Spring MVC is truly View-agnostic: View technology can be replaced with multiple alternatives.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans.

SPRING MVC STRUCTURE

- WebApplicationContext: ApplicationContext adapted for work in Web.
- DispatcherServlet: a servlet used in intercepting all user's requests. It implements "Front Controller" pattern.
- Model: org.springframework.ui.Model, java.util.Map, stores data as key-value pair.
- View: class that implements interface View, and corresponding template.
- Controller: class that implements Controller interface or class annotated with @Controller.

SPRING MVC CONTROLLER

- Spring MVC Controller – is not a classic MVC Controller.
- Role of classic MVC Controller is done by Spring MVC DispatcherServlet.
- DispatcherServlet receives all requests and call your @Controller's methods.



SPRING MVC CONTROLLER

- Here is an example of Spring MVC controller.
- It handles requests such as /order?id=1 and shows web page representing required business object.
- It also encapsulates business service from view details.

```
@Controller
public class OrdersController {
    @Autowired
    private OrderRepository repo;

    @RequestMapping("/order")
    public String viewOrder(
        @RequestParam(value="id")
        String id,
        Model model      // Model
    ) {
        Order order = repo.get(id);
        model.addAttribute("order", order);
        return "orderView";  // View name
    }
}
```

SPRING MVC CONTROLLER

- Annotation Based

  @Controller

  @RestController (just @Controller + @ResponseBody)

- Typically named around business domain
- Path set using annotations:

  @RequestMapping

  @GetMapping, @PostMapping, etc.

SPRING MVC VIEW: JSP

- Here is an example of JSP view.
- Note that you can't use JSP in your Spring Boot application with embedded servlet container.
- But you can use JSP, if you pack your Spring Boot application in classic WAR archive

```
<%@ page language="java"
    contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    ...
</head>
<body>
    <p>Hello, ${name}!</p>
</body>
</html>
```

SPRING MVC VIEW: THYMELEAF

- Here is an example of Thymeleaf view.
- Thymeleaf is recommended by Spring team technology.
- You can use other different view technologies:
  - Groovy Markup Templates;
  - Freemarker;
  - Mustache;
  - your own view implementation.

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    ...
</head>
<body>
    <p th:text="'Hello, ' + ${name} + '!'">
        Hello, John!
    </p>
</body>
</html>
```

SPRING MVC CONTEXT

- WebApplicationContext - this is an extension of ApplicationContext that has some extra features necessary for web applications.
- Adds three scopes of bean lifecycle that are only available in web context: request, session, global (for portlets only).
- Special bean types can only exist in WebApplicationContext.

SPRING MVC CONTEXT

- WebApplicationContext is an interface and has different implementations:

  XmlWebApplicationContext – for XML-based contexts

  AnnotationConfigWebApplicationContext – for Java-based contexts

  GroovyWebApplicationContext – for Groovy-based contexts

  XmlPortletApplicationContext – for XML-based portlets contexts

  etc.

SPRING MVC CONTEXT: SPECIAL BEANS

Bean	Description
Controller	Process user requests
Handler mapping	Analyzes URL and forwards user request to controller
View resolver	Defines which view should be returned in response to name
Locale resolver	Defines what locale should be used
Theme resolver	Defines theme (visual look)
Multipart file resolver	Provides file upload
Handler exception resolver	Defines how exceptions will be handled

SPRING MVC CONTEXT: XML-BASED CONFIGURATION

- If you are using XML-based configuration, you have to specify a context in web.xml via ContextLoaderListener.
- This configuration context for a whole web-application. Usually it contains business classes.
- Such root context is optional.

```
<?xml ...?>
<web-app ...>
  <context-param>
    <param-name>
      contextConfigLocation
    </param-name>
    <param-value>
      classpath:spring/root-context.xml
    </param-value>
  </context-param>

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

SPRING MVC CONTEXT: XML-BASED CONFIGURATION

- Here is servlet context.
- DispatcherServlet is used as a front controller.
- You must define at least one servlet and its mapping.

```
<servlet>
  <servlet-name>myservlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:mvc-context.xml</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>myservlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

SPRING MVC CONTEXT: ROOT AND SERVLET CONTEXTS

- Application beans are defined in the root context (services, DAOs, data sources, etc.)
- And all web related beans (controllers, views etc.) are defined in the servlet context.
- Beans from the root context can normally be used in servlet contexts.

SPRING MVC CONTEXT: XML-BASED CONFIGURATION

- XmlWebApplicationContext sample.
- In Spring MVC XML context you can use special extensions.

```
<beans ...>
  <mvc:view-resolvers>
    <mvc:jsp prefix="/WEB-INF/jsp/"
             suffix=".jsp"/>
  </mvc:view-resolvers>

  <bean id="vets/list.xml"
        class="...MarshallingView">
    <property name="marshaller"
              ref="marshaller"/>
  </bean>
</beans>
```

SPRING MVC CONTEXT: JAVA-BASED CONFIGURATION

- With Servlets 3.0 you can replace web.xml with Java initializer.
  - such root context is optional;
  - you can replace XMLs with Java classes, of course;
  - you may derive initializer from AbstractAnnotationConfigDispatcherServletInitializer or other.

```
import
  org.springframework.web.WebApplicationInitializer;

public class WebAppInitializer
    implements WebApplicationInitializer {
  @Override
  public void onStartup(ServletContext container) {
    // Root context
    AnnotationConfigWebApplicationContext
      rootContext =
        new AnnotationConfigWebApplicationContext();

    rootContext.register(RootConfig.class);


    container.addListener(
      new ContextLoaderListener(rootContext)
    );
  }
}
```

SPRING MVC CONTEXT: JAVA-BASED CONFIGURATION

- Here is the sample of servlet context with URL mapping.

```
public class WebAppInitializer ... {
  @Override
  public void onStartup(ServletContext container) {
    // ... (root context initialization)

    AnnotationConfigWebApplicationContext
      dispatcherServlet = new
        AnnotationConfigWebApplicationContext();
    dispatcherServlet.register(MvcConfig.class);


    // Register and map the dispatcher servlet
    ServletRegistration.Dynamic dispatcher =
      container.addServlet("dispatcher",
        new DispatcherServlet(dispatcherServlet));
    dispatcher.setLoadOnStartup(1);
    dispatcher.addMapping("/*");
  }
}
```

SPRING MVC CONTEXT: JAVA-BASED CONFIGURATION

- AnnotationConfigWebApplicationContext sample.
- Extending from WebMvcConfigurerAdapter from Spring MVC provides methods for more flexible configuration options (optional).
- @Configuration and @EnableWebMvc is required.

```
@Configuration
@EnableWebMvc
public class WebConfig extends
    WebMvcConfigurerAdapter {
  @Override
  public void addResourceHandlers(
      ResourceHandlerRegistry registry) {
    registry
      .addResourceHandler("/WEB-INF/pages/**")
      .addResourceLocations("/pages/");
  }

  @Bean
  public MyService myService() {
    return new MyService();
  }
}
```

SPRING MVC CONTEXT: SPRING BOOT JAVA-BASED CONFIGURATION

- Spring Boot configuration is very simple.
- Spring Boot applications have autoconfigured DispatcherServlet with "/*" mapping.
- @SpringBootApplication includes:
  - @Configuration
  - @EnableWebMvc (if spring-mvc on the classpath).

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

    @Bean
    public MyService myService() {
       return new MyService();
    }
}
```

SPRING MVC CONTEXT: SPRING BOOT JAVA-BASED CONFIGURATION

- You can disable this auto configuration.
- But you have to configure this servlet manually.

```
@SpringBootApplication(exclude =
    DispatcherServletAutoConfiguration.class
)
public class Application {

    ...

}
```

SPRING MVC CONTEXT: SPRING BOOT JAVA-BASED CONFIGURATION

- If you want to register multiple servlets – you can simple do it.
- In embedded servlet container you can automatically register servlets, filters and listeners using:
  - @WebServlet on servlets;
  - @WebFilter on filters;
  - @WebListener on listeners;
  - @ServletComponentScan on configuration class.

```
// will be mapped to "/myServlet/" !
@Bean
public Servlet myServlet() {
    return new MyServletClass();
}

// will be mapped to "/myServlet/*"
@Bean
ServletRegistrationBean registration() {
    return new ServletRegistrationBean(
        new MyServletClass(), "/myServlet/*"
    );
}
```

SPRING MVC CONTEXT: SPRING BOOT JAVA-BASED CONFIGURATION

- There are more classes to configure Spring Boot web-application.

  You can use them, if you need deep customization:

  WebMvcConfigurerAdapter – to configure whole MVC application.

  EmbeddedServletContainerCustomizer – interface to customize embedded servlet container.

  and others, see documentation if you need to customize default behavior.

SPRING MVC AND SPRING BOOT

- Next, we will study Spring MVC with Spring Boot and with Java-based configuration.
- All Spring MVC examples work fine without Spring Boot.
- You can normally use XML configuration.

- But we strongly recommend Spring Boot and Java-based configuration to create your web applications.

SPRING MVC

QUESTIONS?

SPRING MVC: SUMMARY

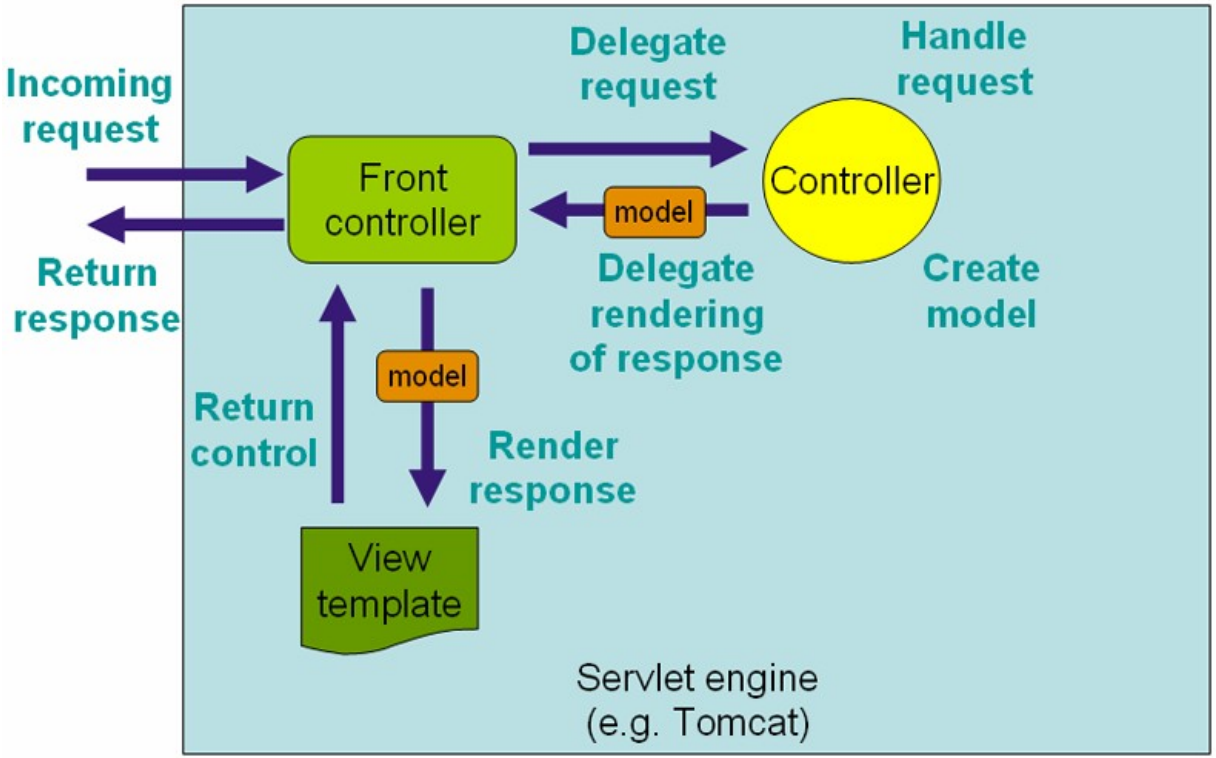
- MVC architecture
- Spring MVC architecture, structure
- WebApplicationContext
- XML-based Spring MVC configuration
- Java-based Spring MVC configuration
- Java-based Spring MVC configuration in Spring Boot applications

SPRING MVC

QUESTIONS?

SECTION 3: CONTROLLER

SPRING MVC ARCHITECTURE



CONTROLLER

- Controllers provide access to the application behavior that you typically define through a service interface.
- Controllers can also handle exceptions from the business services tier.
- Controllers interpret user input and transform it into a model that is represented to the user by the view.
- Controller can also choose required view to display model.
- Spring MVC implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

CONTROLLER: EXAMPLE

- Here is an example of Spring MVC controller.
- All controllers usually have @Controller stereotype annotation (or @RestController).
- Methods that are receive requests are usually have @RequestMapping annotation.

```
@Controller
public class OrdersController {
    @Autowired
    private OrdersRepository repo;

    @RequestMapping("/order")
    public String viewOrder(
        @RequestParam(value="id")
        String id,
        Model model          // Model
    ) {
        Order order = repo.get(id);
        model.addAttribute("order", order);
        return "orderView";  // View name
    }
}
```

@CONTROLLER

- @Controller classes have to be placed in the package that matches to @ComponentScan parameters.
- @ComponentScan is included in @SpringBootApplication annotation.

```
// scan starting from the class package
@ComponentScan(
    basePackageClasses = {MyService.class}
)
@EnableWebMvc
@Configuration
public class Config { ... }


// scan from the given package
@ComponentScan(basePackages = "com.luxoft")


// scan starting from the current package
@ComponentScan
```

@REQUESTMAPPING

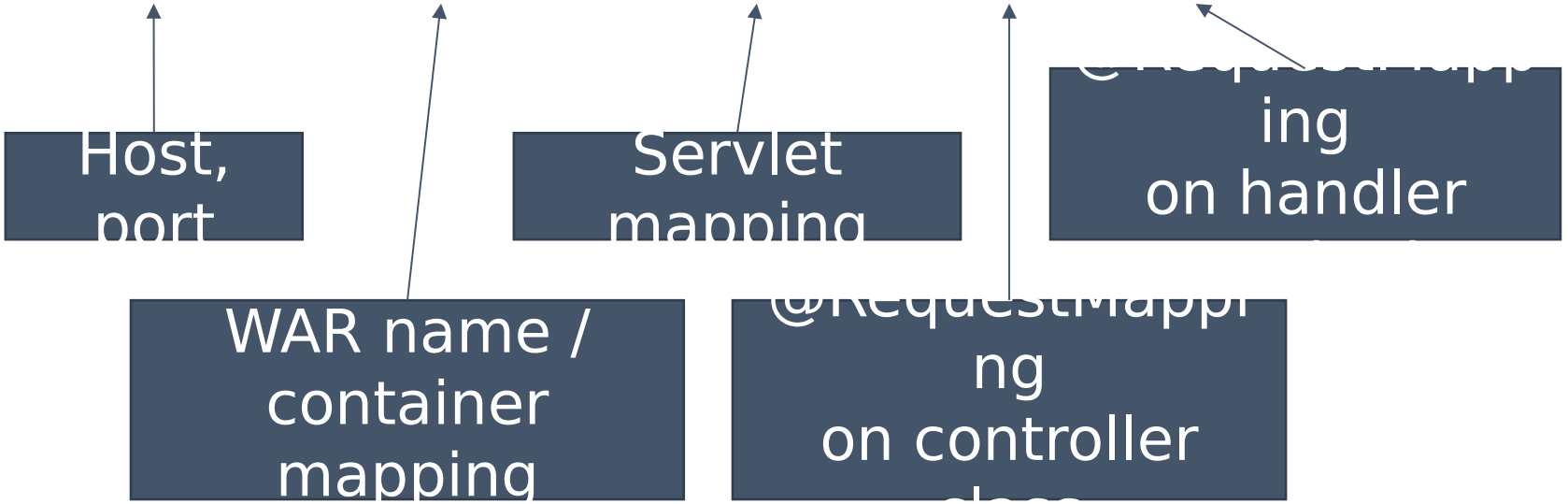
- @RequestMapping annotation can be used onto:
  - controller class (optional);
  - handler method (required).

```
@Controller
@RequestMapping("/orders") // optional
public class OrdersController {

    @RequestMapping(method = RequestMethod.GET)
    public String orders(...) {...}


    @RequestMapping("/view")
    public String viewOrder(...) {...}

    @RequestMapping("/new/normal")
    public String newNormalOrder(...) {...}
}
```

@REQUESTMAPPING

- Full mapped path may be longer:

```
localhost:80/sample-app/api/orders/new/normal
```



@REQUESTMAPPING: @PATHVARIABLE

- With @RequestMapping and @PathVariable you can pass parameters through the URL path.
- This is useful for building RESTful web services.

```
@Controller
public class OrdersController {

    @RequestMapping("/view/{orderId}")
    public String viewOrder(
        @PathVariable String orderId
    ) {
        ...
    }
}
```

@REQUESTMAPPING: MULTIPLE @PATHVARIABLE

- You can use path variables in such annotation on class.
- Spring MVC binds variables according to the constructor argument names.
- In case of different names you can specify required name in @PathVariable.

```
@Controller
@RequestMapping("/orders/{clientId}")
public class OrdersController {

    @RequestMapping("/view/{orderId}")
    public String viewOrder(
        @PathVariable("orderId") String order,
        @PathVariable("clientId") String client
    ) {
        ...
    }
}
```

@REQUESTMAPPING: WILDCARDS

- You can use Ant-style wildcards in @RequestMapping.
- It can be combined with path variables.

```
@Controller
public class OrdersController {

    @RequestMapping("/**/index")
    public String index() { ... }

    @RequestMapping("/*/info/{id}")
    public String info(
        @PathVariable String id
    ) { ... }
}
```

@REQUESTMAPPING: HTTP METHODS

- In @RequestMapping you can use different HTTP methods:

  GET, POST, PUT, PATCH, DELETE.

  HEAD mapped implicitly with GET methods.

  To handle OPTIONS specify dispatchOptionsRequest as true in your servlet. By default Spring MVC handles it themself.

  TRACE method is denied by the container.

```
@Controller
public class SampleController {
    // all available methods
    @RequestMapping("/all")
    public String all() { ... }

    // GET method
    @RequestMapping(
          "/get",
          method = RequestMethod.GET
    )
    public String get() { ... }
}
```

@REQUESTMAPPING: MULTIPLE MAPPINGS

- Note that path (value element) and method are array parameters.
- Spring MVC supports following construction.

```
@Controller
public class OrdersController {
    @RequestMapping(
        value = {"/a", "b"},
        method = {
            RequestMethod.GET,
            RequestMethod.POST
        }
    )
    public String info() { ... }
}
```

@REQUESTMAPPING: HTTP-METHODS ANNOTATIONS

- Since Spring MVC 4.3 you can use following annotations:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

```
@Controller
public class OrdersController {

    @GetMapping("/order")
    public String info() { ... }
}
```

@REQUESTMAPPING: PARAMS

- Here is the sample of using params argument of @RequestMapping annotation.

```
// /write?day=monday

@RequestMapping(
    value = "/write",
    params = "day")
public void writeSomeDay() { ... }


// /write

@RequestMapping(
    value = "/write",
    params = "!day")
public void writeNoDay() { ... }
```

@REQUESTMAPPING: HEADERS

- You can map methods using request headers.

- In what cases these methods are receive requests?

```
@RequestMapping(
  value = "/process",
  headers = "Accept=text/html"
)
public void processTextData(...) {
  ...
}

@RequestMapping(
  value = "/process",
  headers = "Accept=application/json"
)
public void writeNoDay(...) {
  ...
}
```

POSSIBLE ARGUMENTS: @REQUESTPARAM

- To receive request parameters you can use @RequestParam annotation.
- Spring MVC automatically parse arguments and converts to required types.
- You can specify whether or not a parameter is required.

```
@RequestMapping(method = RequestMethod.GET)
public String showOrder(
    @RequestParam("id") long id
) {
    // ...
    return "viewName";
}

@RequestMapping(method = RequestMethod.POST)
public void createOrder(
    @RequestParam(
         value = "amount",
         required = false
    ) Integer amount
) {
    // ...
}
```

POSSIBLE ARGUMENTS: @REQUESTPARAM ON MAP

- If you use @RequestParam on Map<String, String> or MultiMap<String,Sting>, Spring MVC place there all request parameters.

```
@RequestMapping("/anyUrl")
public void anyMethod(
    @RequestParam Map<String,String> params
) {
    String id = params.get("id");
    String name = params.get("name");
    // ...
}
```

POSSIBLE ARGUMENTS: POJO

- You can specify plain objects in methods arguments.
- Spring MVC maps parameters to POJO fields.

```
class MyRequest {
    private String id;
    private int amount;
    // ...
}


// method?id=1&amount=2


@RequestMapping("/method")
public void method(
    MyRequest request
)
```

POSSIBLE ARGUMENTS: @REQUESTBODY

- If you want to receive complex objects (JSON / XML / Other), you can use @RequestBody annotation with your method (that have a request body).
- Spring Boot has preconfigured Jackson to convert JSONs.

```
class User {
    private String name;
    private int age;
    // ...
}

// {"name": "John", "age": 45}

@RequestMapping(
    value = "/create",
    method = RequestMethod.POST
)
public void method(
    @RequestBody User user
) { ... }
```

POSSIBLE ARGUMENTS: MODEL

- Controllers can receive model.
- org.springframework.ui.Model is an interface to store data in the model as key-value.
- After method call this Model will be passed to view.
- ModelMap and java.util.Map parameters have the same behavior.

```
@RequestMapping("/order/view")
public String showOrderPage(
    @RequestParam String orderId,
    Model model
) {
    Order order =
        orderRepository.get(orderId);
    model.addAttribute("order", order);
    return "orderPage";
}
```

POSSIBLE ARGUMENTS: WEBREQUEST

- org.springframework.web.context.request.WebRequest is the most powerful argument of controller's method:
  - request headers,
  - request description,
  - localization,
  - session object Principal,
  - etc.

```
@RequestMapping(value = "/webrequest")
public String webRequest(
    WebRequest webRequest, Model model
) {
    model.addAttribute(
        "content",
        "Session id: " +
            webRequest.getSessionId()
    );
    return "viewName";
}
```

POSSIBLE ARGUMENTS: @REQUESTHEADER

- You can get specified request header with @RequestHeader annotation.

```
@RequestMapping(value =
"/requestheader")
public String requestHeader(
    @RequestHeader("User-Agent")
    String userAgent,
    Model model
) {
    model.addAttribute(
        "content",
        "User-Agent: " + userAgent
    );
    return "viewName";
}
```

POSSIBLE ARGUMENTS: HTTPSESSION AND LOCALE

```
@RequestMapping(value = "/httpsession")
public String httpSession(
    HttpSession session,
    Model model
) {
    model.addAttribute(
        "content",
        "Session id:" + session.getId()
    );
    return "view";
}
```

```
@RequestMapping(value = "/locale")
public String locale(
     Locale locale,
     Model model
) {
    model.addAttribute(
        "content",
        "Locale language: "
            + locale.getLanguage()
    );
    return "view";
}
```

POSSIBLE ARGUMENTS: MULTIPART DATA

- Here is the sample how to process multipart data (files) with Spring Boot.
- Note that without Spring Boot you have to configure some beans.

```
@RequestMapping(
    value = "/upload",
    method = RequestMethod.POST
)
public void uploadFile(
     @RequestParam("file")
     MultipartFile file
) {
    LOG.trace(file.getBytes());
}
```

POSSIBLE CONTROLLER ARGUMENTS

Argument	Description
@PathVariable annotated	Parameter from the path
@MatrixVariable annotated	Name-value parameters from the path
@RequestParam annotated	Request parameters
Your request POJO	To map request parameters to POJO fields
@RequestBody annotated	Object converted from the HTTP request body by a special converter.
@RequestHeader	With it you can access to HTTP request

POSSIBLE ARGUMENTS

Argument	Description
@RequestPart annotated	For "multipart/form-data" support (file uploading)
java.util.Locale	User locale (works with configured LocaleResolver only)
java.util.TimeZone (Java 6+) / java.time.ZoneId (Java 8)	User time zone (LocaleResolver is required)
java.io.InputStream / java.io.Reader	For access to the request’s content
java.io.OutputStream / java.io.Writer	For generating the response’s content

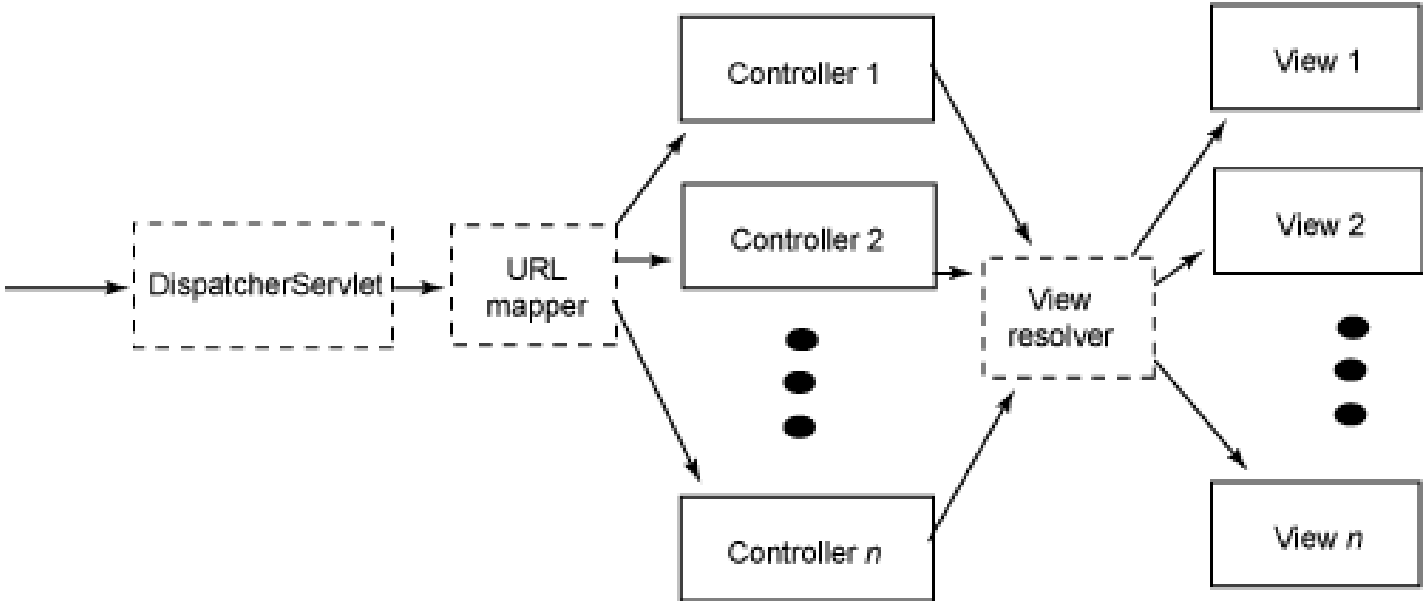
POSSIBLE ARGUMENTS

Argument	Description
HttpServletRequest	Servlet API request object
HttpServletResponse	Servlet API response object
HttpSession	Servlet API session object
HttpEntity<?>	To get request as HttpEntity
org.springframework.web.context .request.WebRequest	More powerful request object
org.springframework.http.HttpMethod	To get information about method
@SessionAttribute, @RequestAttribute	For session/request attributes

POSSIBLE CONTROLLER ARGUMENTS

Argument	Description
@CookieValue annotated	To get value of an HTTP cookie
org.springframework.web .servlet.mvc.support .RedirectAttributes	For passing data to redirect targets
org.springframework .validation.Errors or .BindingResult	To get validation errors or binding results
org.springframework.web .util.UriComponentsBu	For preparing a URL relative to the current request's URL

CONTROLLER AND VIEW



CONTROLLER AND VIEW

- Usual Web-page controllers return a logical name of view.
- This name is translated by ViewResolver to template file and corresponded class/technology to render a webpage.
- In this sample "orderPage" may mean "orderPage.jsp".

```
@Controller
public class OrdersController

  @RequestMapping("/order/view")
  public String showOrderPage(
    @RequestParam String orderId,
    Model model
  ) {
    Order order =
      orderRepository.get(orderId);
    model.addAttribute("order", order);
    return "orderPage";
  }
}
```

CONTROLLER AND VIEW

- You cannot return view name in your controller method if ViewResolver is properly configured.

- Or you can render page manually using HttpServletResponse argument.

```
@Controller
public class OrdersController

  @RequestMapping("/method")
  public void renderPage(
    @RequestParam String orderId,
    HttpServletResponse response
  ) {
    Order order =
      orderRepository.get(orderId);
      // ...
  }
}
```

@RESPONSEBODY

- But if you want to return data (JSON / XML / Other) in response, you can use @ResponseBody annotation on return type.
- Spring Boot has preconfigured Jackson to return JSON.

```
@Controller
public class OrdersController

  @RequestMapping("/order")
  public @ResponseBody Order get(
    @RequestParam String orderId,
  ) {
    Order order =
      orderRepository.get(orderId);
    return order;
  }
}
```

@RESTCONTROLLER

- To create such REST controller:
  - @Controller on class;
  - @ResponseBody on each method.
- Or simply annotate class with @RestController.
- @RestController interface annotated with both @Controller and @ResponseBody

```
@RestController
public class OrdersController

  @RequestMapping("/order")
  public Order get(
    @RequestParam String orderId,
  ) {
    Order order =
      orderRepository.get(orderId);
    return order;
  }
}
```

@CONTROLLER AND @RESTCONTROLLER

- @Controller
  - returns logical View name
  - receives pure HTTP parameters
  - shows pages or performs form submission
  - on error displays corresponding error page (e.g. 404-page, 500-page with exception info)
- @RestController
  - returns object, that converts to JSON
  - receives HTTP parameters or JSON (using @RequestBody) , that will be converted to an object
  - processes Ajax requests from pages
  - on error returns JSON with

@CONTROLLER: RESPONSEENTITY

- You can do the same with ResponseEntity object.
- ResponseEntity can be used as a builder to specify:
  - HTTP status codes;
  - response body;
  - headers;
  - ETags
  - ...

```
@RequestMapping("/ping")
public ResponseEntity<String> ping() {
    return ResponseEntity.ok("OK");
}

@RequestMapping("/badRequest")
public ResponseEntity error() {
    return ResponseEntity
        .status(HttpStatus.BAD_REQUEST)
        .build();
}
```

POSSIBLE RETURN TYPES

Return type	Description
String	Logical view name. ViewResolver determines view by this name.
View	View object.
ModelAndView	View and model to render page.
Model, Map, void	Model, required view will be determined by RequestToViewNameTranslator
void	When response are created through HttpServletResponse argument

POSSIBLE RETURN TYPES

Return type	Description
@ResponseBody annotated object	Object that will be converted to JSON/XML/Other using a corresponded converter from the context.
HttpEntity<?> / ResponseEntity<?>	Response entity, possible contained you response object.
Any POJO	Will be considered as a single-object model.
HttpHeaders	To return a response with no body.
Callable<?> / DefferedResult<?> / ListenableFuture<?>	If you want to produce the return value asynchronously in a thread managed by Spring MVC or from custom thread.

POSSIBLE RETURN TYPES

Return type	Description
ResponseBodyEmitter	To write multiple objects to the response asynchronously
SseEmitter	To write Server-Sent Events to the response asynchronously
StreamingResponseBody	To write to the response OutputStream asynchronously

@EXCEPTIONHANDLER

- Here is the sample how to handle exceptions which can be thrown in requests handlers.
- @ExceptionHandler annotation specifies an exception that should be handled.
- The object of the same class should be present in arguments.

```
@Controller
public class MyController {
    // requests handlers
  @ExceptionHandler(
      NullPointerException.class)
  public ModelAndView handleNPE(
      NullPointerException e
  ) {
      ModelAndView modelAndView
          = new ModelAndView("err500");
      modelAndView.addObject(
          "message", e.getMessage()
      );
      return modelAndView;
    }
}
```

@EXCEPTIONHANDLER AND @CONTROLLERADVICE

- You can specify exception handlers for all controllers.
- @ExceptionHandler can be used in @ControllerAdvice classes.
- @ControllerAdvice is the special stereotype to add auxiliary functionality.

```
@ControllerAdvice
public class GlobalExceptionController
{
  @ExceptionHandler(Exception.class)
  public ModelAndView handleAll(
    Exception ex
  ) {
    ModelAndView model
      = new ModelAndView("err500");
    model.addObject(
      "errMsg", ex.getMessage()
    );
    return model;
  }
}
```

SPRING BOOT UNIT TESTS: MOCKMVC

- Spring Boot provides some features for writing unit test for Spring MVC components.
- MockMvc class allows you write unit tests for your controllers.

```
@RunWith(SpringRunner.class)
@WebMvcTest(VehicleController.class)
public class ControllerTests {
  @Autowired
  private MockMvc mvc;

  @MockBean
  private VehicleService vehicleService;

  @Test
  public void test() throws Exception {
    given(vehicleService.getDetails("id"))
      .willReturn(new Vehicle("Honda"));
    this.mvc.perform(get("/id/vehicle"))
      .andExpect(status().isOk())
    .andExpect(content().string("Honda"));
  }
}
```

ex.12

CONROLLERS: SUMMARY

- What is Spring MVC Controller
- @Controller
- @RequestMapping
- @RequestParam
- Possible method arguments
- Model usage
- Logical view name usage
- @ResponseBody
- @RestController
- Possible method return types
- @ExceptionHandler

CONTROLLERS

QUESTIONS?

CONTROLLERS: EXERCISE

Exercise #2

- Creating Spring MVC REST application based on Spring Boot.

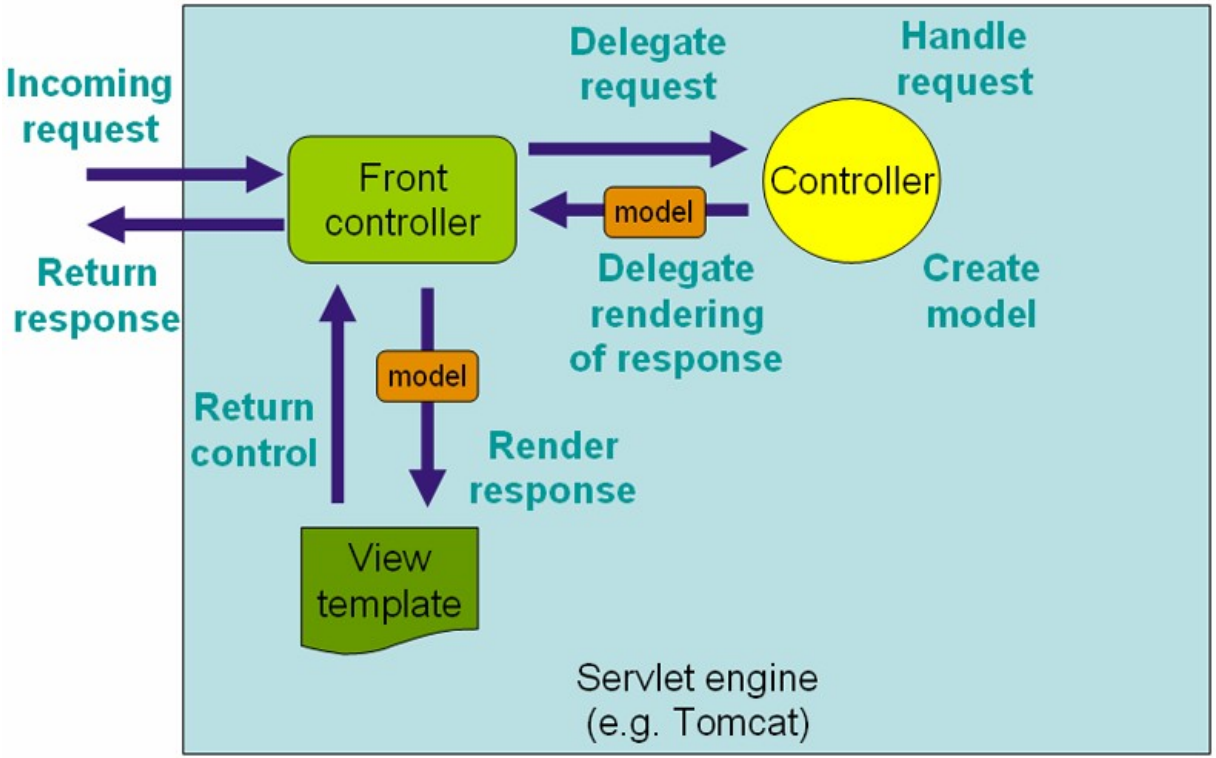
Discuss.

CONTROLLERS

QUESTIONS?

SECTION 4:
VIEW

SPRING MVC ARCHITECTURE



VIEW

- View is intended to present model, which was built in controller.
- Controller can return a logical name of view, which can be resolved to a particular view page by means of ViewResolver.
- Also, you can render a webpage manually without any view.

- There are many view technologies supported by Spring MVC:
  - JSP (with limitations in Spring Boot);
  - Thymeleaf;
  - Groovy Markup Templates;
  - Freemarker;
  - Velocity (not supported in Spring Boot);
  - Or your own technology.

THYMELEAF

- Spring Boot does not support JSP in embedded servlet container mode.
- Spring community recommends Thymeleaf template engine for your Spring Boot application.
- You can use JSP in classic WAR archive, but not in embedded servlet container.
- And no one restricts you to use other view technology.

THYMELEAF AND JSP

Thymeleaf Template

```
<!DOCTYPE HTML>
<html
   xmlns:th="http://www.thymeleaf.org">

...

<span th:text="Hello, ${name}">
    Hello, Username
</span>
```

page.jsp (JSP)

```
<%@page ...>
<%@taglib ...>

...

<span>
    Hello, ${name}
</span>
```

THYMELEAF AND JSP

- Thymeleaf
  - Valid HTML with sample text
  - File is displayed correctly in browsers
  - Easy design
  - Stronger team collaboration
- JSP
  - Invalid HTML
  - Can’t be correctly viewed without application
  - Professional web design is hard to perform

THYMELEAF: SPRING BOOT DEPENDENCIES

- spring-boot-starter-thymeleaf:
  - thymeleaf-spring4
  - thymeleaf-layout-dialect
- Can be added manually or with start.spring.io

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

THYMELEAF: SYNTAX

- th:text

  defines text which will be placed instead of the text in the tag

- #{localized.hello} – denotes localized message

```
page.html:

<span th:text="#{localized.hello}">
    Hello!
</span>



messages.properties:

localized.hello=Hello!
```

THYMELEAF: SYNTAX

- You can use parameterized messages.

- What will be displayed at this page?

page.html:

```
<span th:text="#{localized.hello('Mike')}">
    Hello, Username!
</span>
```

messages.properties:

```
localized.hello=Hello, {0}!
```

SPRING MVC LOCALIZATION

- To show localized page Spring MVC can perform 3 kinds of actions:
  - Determine current user's locale
  - Change user's locale
  - Display localized text/messages at the page
- These actions can be done using following components:
  - LocaleResolver
  - LocaleChangeInterceptor
  - With localized messages and placeholders on the view.

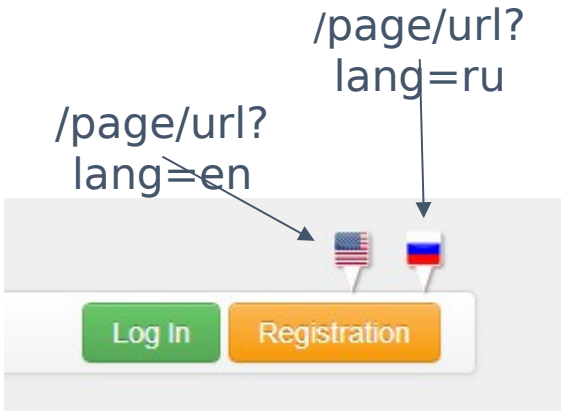
SPRING MVC LOCALIZATION: LOCALERESOLVER

- To determine locale you can use locale resolvers:
  - AcceptHeaderLocaleResolver (uses "Accept-Language" header)
  - CookieLocaleResolver (current locale stores in the cookie)
  - SessionLocaleResolver (current locale stores in a session)

```
// Here is CookieLocaleResolver

@Bean
public LocaleResolver resolver() {
  CookieLocaleResolver resolver
    = new CookieLocaleResolver();
  resolver.setDefaultLocale(
    new Locale("en"));
  resolver.setCookieName("locale");
  return resolver;
}
```

SPRING MVC LOCALIZATION: LOCALECHANGEINTERCEPTOR

- With LocaleChangeInterceptor user may change the current page locale following by links:




```
// In your web context

@Bean
public LocaleChangeInterceptor lci()
{
  LocaleChangeInterceptor lci
    = new LocaleChangeInterceptor();
  lci.setParamName("lang");
  return lci;
}
```

SPRING MVC LOCALIZATION: LOCALECHANGEINTERCEPTOR

- LocaleChangeInterceptor (like all other interceptors) have to be added in servlet configuration.

```
@Configuration
public class WebConfig extends
    WebMvcConfigurerAdapter {

  // ...

  private LocaleChangeInterceptor lci;

  @Override
  public void addInterceptors(
    InterceptorRegistry registry
  ) {
    registry.addInterceptor(lci);
  }
}
```

SPRING MVC LOCALIZATION: MESSAGESOURCE

- MessageSource is used to specify localization sources
- In this sample localization messages are stored in files:

  /i18n/bundle_en.properties

  /i18n/bundle_ru.properties

  etc.

```
@Bean
public MessageSource messageSource() {
  ReloadableResourceBundleMessageSource ms
    = new
    ReloadableResourceBundleMessageSource();
  ms.setBasename("/i18n/bundle");
  ms.setDefaultEncoding("UTF-8");
  return ms;
}
```

SPRING MVC LOCALIZATION: MESSAGES AND PLACEHOLDERS

bundle_en.properties:

```
localized.hello=Hello!
```

page.html:

```
<span th:text="#{localized.hello}">
    Hello!
</span>
```

SPRING BOOT LOCALIZATION

- Spring Boot has preconfigured LocaleResolver and MessageSource.
- By default bundle files are placed in src/main/resources :

  src/main/resources/messages.properties – default locale (required!)

  src/main/resources/messages_en_US.properties – other locales
- All files are interpreted in UTF-8 encoding
- You can change bundle location in application.properties:

  spring.messages.basename=path/to/i18n/messages

THYMELEAF: SYNTAX

```
// Controller

@RequestMapping("/hello")
public String hello(
  @RequestParam String name,
  Model model
) {
  model.addAttribute(
    "username", name
  );
  return "helloPage";
}
```

helloPage.html:

```
<span th:text="${username}">
    Hello, Username!
</span>
```

MODEL AND VIEW

- You can put into your model more complex objects.
- ${user.name} – denotes Spring Expression Language (SpEL).
- With SpEL you can simply get object fields.

```
helloPage.html:

<span th:text="${user.name}">
    Hello, Username!
</span>

// Java code:

class User {
    private String name;
    ...
}
model.addAttribute("user", user);
```

THYMELEAF: SYNTAX

- You can combine expressions
- th:utext – an unescaped text

```
page.html:

<span th:utext="
   #{localized.hello(${user.name})}
">
    Hello, John Doe!
</span>


messages.properties:

localized.hello=Hello, {0}!
```

SPEL

- SpEL gives you different ways to get fields and variables
- Also you can call methods from objects

```
${person.father.name}

${person['father']['name']}

${personsArray[0].name}

${person.createCompleteName()}

${person.createWithSeparator('-')}
```

THYMELEAF: SYNTAX

- *{name} same as ${user.name} but shows expression in context of object user
- th:object tells us about context of *{...}
- If object is not set, ${...} and *{...} are equivalent

```
page.html:

<div th:object="${user}">
    <span th:text="*{name}">
        John Doe
    </span>
</div>


class User {
    private String name;
    ...
}
```

THYMELEAF: *{} AND ${} EXPRESSIONS

```
<div th:object="${user}">
   <span th:text="*{name}">
      John Doe
   </span>
   <span th:text="*{age}">
      25
   </span>
</div>
```

```
<div>
   <span th:text="${user.name}">
      John Doe
   </span>
   <span th:text="${user.age}">
      25
   </span>
</div>
```

THYMELEAF: SYNTAX

- With @{/url} you can place correct URLs when app running and when HTML-page design is performing.

```
page.html:

<a href="account.html"
   th:href="@{/account(id=${id})}">
      Account Page
</a>


/page in app:

<a href="/account?id=1">
      Account Page
</a>
```

THYMELEAF: SYNTAX

- th:action – configure forms action
- th:value – value of input

page.html:

```
<form th:action="@{/main}">
   <input th:value="${name}"/>
</form>
```

SPRING BOOT: THYMELEAF SYNTAX

- th:each – Thymeleaf foreach
- th:if – rendered if condition is true

page.html:

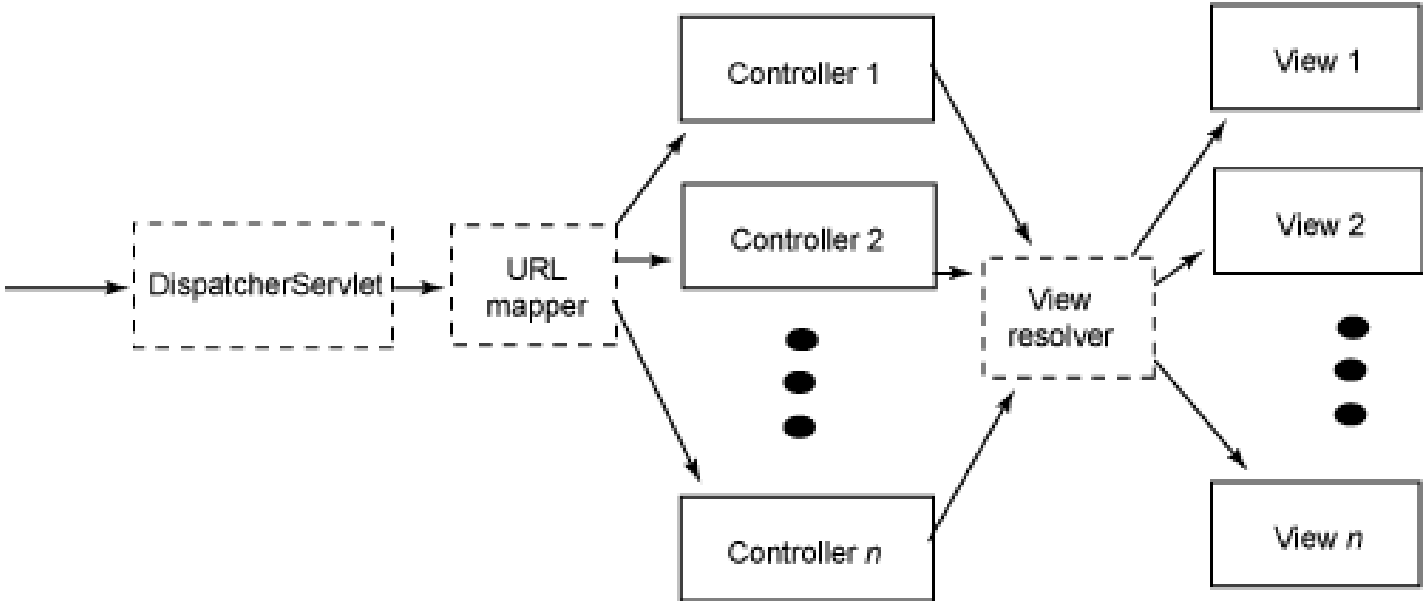
```
<li th:each="account : ${accounts}">
   <span th:text="${account.owner}">
      John Doe
   </span>
</li>


<div th:if="${@myBean.hasErrors()}">
   <span>ERROR!</span>
</div>
```

SPRING BOOT: THYMELEAF SYNTAX

- @beanName – Spring bean with name "beanName"
- #locale – current locale
- #ctx – current context
- Concatenation, arithmetic operations, conditional operators.

- and many other predefined constants and features

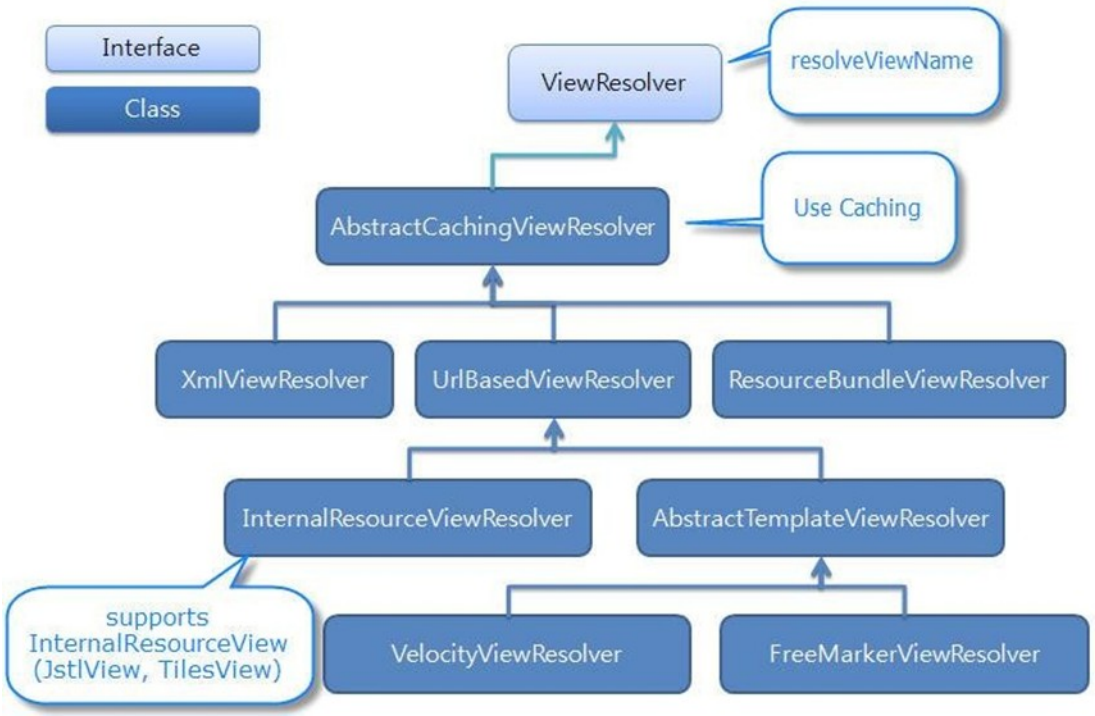
VIEW RESOLVERS



VIEW RESOLVERS

- There is the special bean interface ViewResolver in Spring MVC
- These beans resolve logical view names to templates and technologies for rendering.
- Spring Boot's ViewResolvers are autoconfigured.
- If you add spring-boot-starter-thymeleaf dependency to you POM, Spring Boot creates ViewResolver, that resolves the "home" view name to "classpath:templates/home.html" template and renders it with Thymeleaf technology.

VIEW RESOLVERS



VIEW RESOLVERS: MANUAL CONFIGURATION

- You can create ViewResolver manually.
- Note that you can specify multiple ViewResolvers.
- For some ViewResolvers you can specify "order" property that defines its priority.

```
@Bean
public ViewResolver viewResolver() {
  InternalResourceViewResolver r
    = new
      InternalResourceViewResolver();
  r.setPrefix("/template/");
  r.setSuffix(".html");
  return r;
}
```

QUESTIONS?

VIEW: EXERCISE

Exercise #3

- Creating Spring MVC Web application using Thymeleaf.

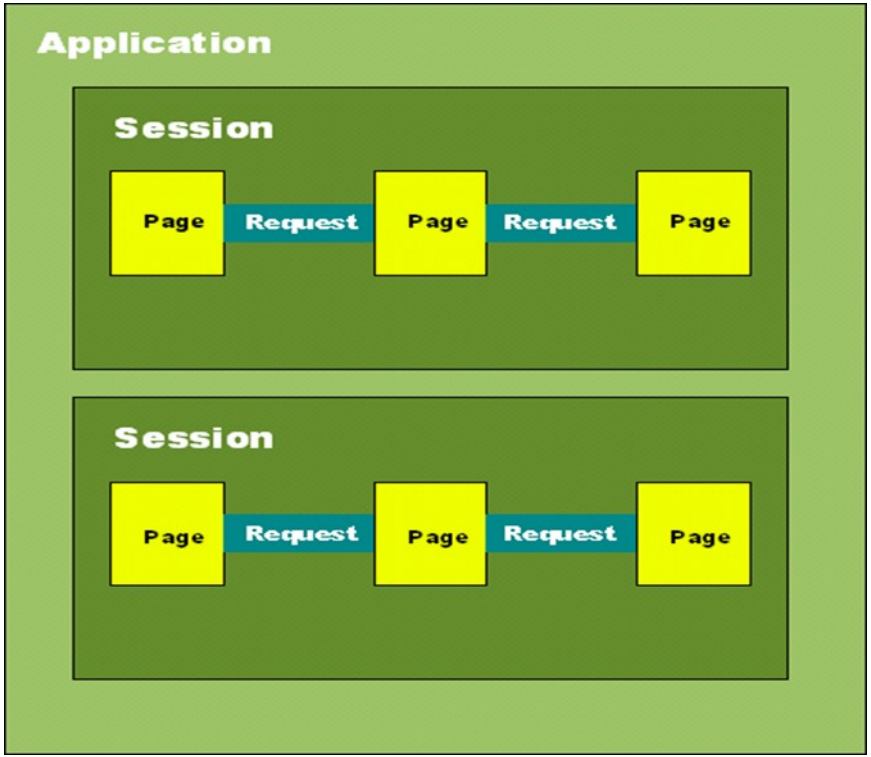
Discuss.

QUESTIONS?

SECTION 4:
REQUEST AND SESSION SCOPES

SPRING MVC BEAN SCOPES

Scope	Description
singleton (default)	one instance in whole application
prototype	one instance for each demanding bean
request	one instance in each request
session	one instance in each web-session (in servlet container)
global session	one instance in global HTTP-session (for portlets)



BEAN SCOPES: SINGLETON

```
class OrderRepository {
  public Order get(String id) {
    return ...;
  }
}


@Configuration
public class Config {
  @Bean
  public OrderRepository repo() {
    return new OrderRepository();
  }
}
```

```
@Controller
public class OrderController {
  @Autowired
  private OrderRepository repo;

  @RequestMapping("/order")
  public String viewOrder(
     @RequestParam(value="id")
     String id,
     Model model
  ) {
    Order order = repo.get(id);
    model.addAttribute("order", order);
    return "orderView";
  }
}
```

BEAN SCOPES: REQUEST

- Request scoped beans creates every time for each request.

```
@Bean
@Scope("request")
public UtilityClass util() {
    return new UtilityClass();
}
```

BEAN SCOPES: SESSION

- Session scope is more popular.
- Usually session is used to store:
  - User preferences
  - Security data
  - Forms data
  - Cross-page data

```
@Bean
@Scope("session")
public ShoppingCart userCart() {
    return new ShoppingCart();
}
```

USING SESSION BEANS: HTTPSESSION

- To get session beans use HttpSession object.
- Note, that you cannot autowire these beans directly.

```
@RequestMapping("/add")
public String addToCart(
    @RequestParam("id") int id,
    HttpSession session
) {
  ShoppingCart cart =
(ShoppingCart)session.getAttribute(
    "userCart"
  );
  cart.add(id);
  return "catalogPage";
}
```

USING SESSION BEANS: PROXY

- To autowire session scoped beans in a singleton controller, you have to use AOP proxies.
- Spring creates proxy that redirects all method calls to the required session instance.

```
@Bean
@Scope(value = "session", proxyMode =
    ScopedProxyMode.TARGET_CLASS)
public ShoppingCart userCart() {
  return new ShoppingCart();
}

@Controller
public class CartController {
  @Autowired
  private ShoppingCart cart;
  ...
}
```

@MODELATTRIBUTE

- If we need to create any object and place it to the model, there is a special annotation @ModelAttribute.
- Annotated method creates object and places it to the model directly before handler call.
- Here is two usage styles.

```
@Controller
public class AccountController {
  @ModelAttribute
  public Account account(
      @RequestParam String number) {
    return manager.find(number);
  }

  @ModelAttribute
  public void populate(
      @RequestParam String number,
      Model model) {
    model.addAttribute("account", ...
  }
}
```

@MODELATTRIBUTE: GET FORM DATA

```
// show page and write to model

@RequestMapping(
    value = "/edit/{id}",
    method = RequestMethod.GET
)
public String edit(
    @PathVariable int id,
    Model model
) {
    model.addAttribute(
      "account", manager.get(id)
    );
    return "editPage";
}
```

```
// get form data and store it

@RequestMapping(
    value = "/edit/*",
    method = RequestMethod.POST
)
public String save(
    @ModelAttribute("account")
    Account account,
    Model model
) {
   manager.save(person);
   return "successPage";
}
```

@SESSIONATTRIBUTES

- To store attributes in the session you can use @SessionAttributes annotation on the controller

```
@Controller
@SessionAttributes("account")
public class AccountController {
    @RequestMapping("/edit/{id}")
    public String edit(
        @PathVariable int id,
        Model model
    ) {
        // store in session
        model.addAttribute(
            "account", manager.get(id)
        );
        return "editPage";
    }
}
```

REQUEST AND SESSION SCOPES

QUESTIONS?

REQUEST AND SESSION SCOPES: EXERCISE

Exercise #4

- Working with sessions.

Discuss.

REQUEST AND SESSION SCOPES

QUESTIONS?

THANK YOU!