Spring Boot Fundamentals Course Projects

This directory contains projects for the Advanced Java/Spring Boot course. These projects are mostly the intermediate states of building a REST application. Each one highlights a few concepts, and they build on each other. The final destination is the **LarkU_Boot** project. Project contents are further described below.

The application we are going to be building is a simple school Registration system. The domain model consists of **Student**s, **Course**s and **ScheduledClass**es. The application allows you and add and delete **Student** and **Course** resources. You can schedule a course for specific dates using a **ScheduledClass**, and you can register a student for a scheduled class. The focus of this course is primarily on REST applications, but the final project also has a web front end which has examples of Javascript access.

1. PreSpring: The basic Java application before Spring has been added.
   a. Get familiar with the pieces of the application and how they are wired together.
   b. Why Spring? Problems and issues with hand wiring application. The need for Dependency Injection.
   c. The Factory pattern. Implementing a simple Factory. Changing wiring based on a property.

2. PostSpring: The home grown Factory above has been replaced by Spring. This is where we will talk about the nuts and bolts of Spring.
   a. What is a Bean?
   b. **Lifecycle** and **Scope** of Beans.
   c. Configuration – XML based, Java based. We will mostly concentrate on Java configuration.
   d. **@Configuration**, **@Bean**, **@Component**, **@Service**, **@Repository**
   e. Spring **ApplicationContext**
   f. Dependency Injection techniques.
      1. Property Injection
      2. Constructor Injection
      3. Setter Injection
   g. Using Spring **Profiles** to selectively wire an application.
   h. An introduction to testing using Spring.

3. Spring Boot: At this point we want to see what Spring Boot can do for us. We took the long route to get here, but normally creating the Spring Boot shell would be your first step in building an application. Spring Boot is basically an uber configurer that looks at your configuration files and class path and sets up your application. For very simple projects you only need to set up the build configuration (pom.xml or build.gradle) and fire up the application. For most projects though, you will normally have to provide it with configuration information, often in the form or properties in an **application.properties** or an **application.yml** file.

   The other interesting thing about Spring Boot is that it brings the server “inside” the application. In the previous project, we managed the server independently of our application. With Spring Boot, the server is most often an embedded server. The build process creates a jar and you start your application as a normal java application: java -jar MyApp.jar. Very convenient. You can, of course, also build a standard war file and deploy to a server in the normal way.

What to look for in this project:

a. Create a Spring Boot application – http://start.spring.io
b. Explore the new application and build file
c. Migrate our code to Spring Boot – discuss ways to do that
d. Adjust our application layout for Spring Boot
e. Create Spring Boot configuration files as needed
f. Controllers
    1. The role of Controllers
    2. Brief tour of Spring MVC architecture – **DispatcherServlet** etc.
    3. Controller configuration – **@Controller, @RestController**.
    4. Content Negotiation - **@Produces**, **@Consumes**.
    5. Controller mappings.
    6. Controller method parameter/return types - **@RequestBody**, **@ResponseBody**, **ReponseEntity**, **UriComponentBuilder**.
g. Create a Spring Boot main class - **@SpringBootApplication**, **SpringApplication.run**
h. Return wrapper types from REST methods - **RestResult**
i. Testing testing and more testing – **@SpringBootTest**, **Mockito**, **MockMvc**, **RestTemplate**

4. SpringDB: Here we add support for a real database. We will use an embedded H2 database for the *development* profile and a derby database for the *production* profile. The H2 database will not persist between application runs. We are going to use JPA to interact with the database.
    a. A 10,000 foot view of JDBC and JPA.
    b. Spring Boot and databases.
    c. Datasource configuration.
    d. Profile specific configuration.
    e. Initializing databases: **schema.sql/data.sql,** Hibernate initialization.
    f. Testing: **@Sql.**
    g. Spring Repositories: **CrudRepository, PagingAndSortingRepository, JpaRepository, @EnableJpaRepositories.** Examples in ttl.larku.dao.repository
        1. Custom findMethods
        2. Custom Repositories – CustomBaseRepo
        3. Paging and Sorting – examples in **StudentRepoTest.testPaging**
    h. Spring Data REST: Exposing repositories as REST resources.
        1. **@RestResource/RepositoryRestResource**
        2. Customize paths and exposure of methods
        3. Projections and Excerpts – expose subsets of resource properties. Domain classes in the **ttl.larku.domain** package. Code examples in **StudentRepo** and **StudentRepoTest**.

5. Spring Boot Actuators: Monitor your application
    a. General Info
    b. Health Checks
    c. Configure logging
    d. Custom Actuators

6. SpringSecurity: A filter and proxy based security mechanism for Spring applications
    a. AOP Examples

1. Filter Example – **TimingFilter**
2. AOP Example – **TimingAspect**

b. Spring Security Architecture
1. AuthenticationProvider
2. UserDetailsService
3. InMemoryUserDetailsService and JDBCUserDetailsService

c. Configuring Spring Security
1. WebSecurityConfigurerAdapter
2. Configure **HttpSecurity**
3. Create custom UserDetailsService
4. Method level Security
1) @EnableGlobalMethodSecurity
2) @Secured

d. Setting up SSL

7. LarkU_SpringBoot: This is the completed application.
Some goodies here that we have not seen before:
a. Exception handling: **LastStopHandler** implements a REST exception handler to deal with all Exceptions that are not caught elsewhere in the application.
b. **@RestControllerAdvice** for declaring the exception handler.
c. Extending **ResponseEntityExceptionHandler** to be able to deal with errors that occur before Spring has called our controllers, e.g. bad MediaType errors, or Validation errors.
d. Creating a custom **HttpMessageConverter** to convert your domain objects to custom media types.

8. Optional: SpringReactive: An introduction to reactive programming and project Reactor
a. Whyfor Reactive?
b. Reactive Streams interfaces: **Publisher**, **Subscriber**, **Subscription, Processor**
c. Project Reactor interfaces: **Flux**, **Mono**
d. Reactive REST services – **Webflux, WebClient**