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 PostSpring: The home grown Factory above has been replaced by Spring.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**.
    7. HandlerInterceptors
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. 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

6. 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.

7. Intro to Cloud Computing Concepts. The projects in the **eureka** are a demo of some cloud computing concepts and libraries. We will be looking at a collection of libraries and components from the Spring Cloud Netflix project. Lots of good info at: https://cloud.spring.io/spring-cloud-netflix/reference/html/
   a. Discovery Service: **Eureka.** In a micro services architecture, you are may need to interact with several services. The services will be running in containers some where "out there", and it is not feasable for a client to have prior knowledge of ip addresses and ports. A **discovery service** allows a client to discover service address using well know service names. **Netflix Eureka** is the library we use as an example. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance
   b. Clustering and Load Balancing: **Feign** and **Ribbon.** You may want to run several instances of a service. The client should have some way to reach the instances without needing each ip address. There should be some mechanism to load balance between the servers. **Feign** and **Ribbon** do the trick.
   c. Some https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-feign.html.
   d. To run the examples:
      1. Start the Eureka server in the **eureka** directory
         1) Look over the application.yaml file. Note the port number the server will run on. If you have a conflict on your machine, you may need to change the port number.
         2) Start the server by running **ttl.springboot.eurekaserver.EurekaServerApplication**
         3) Point a browser at localhost:port_number_your_server_is_running_on
         4) You should see the main Eureka landing page with nothing registered.
      2. Start the micro services that will register themselves with the Eureka server you just started. We have three services:
         1) **TrackPriceService** – A service that takes a track name and gives us back a price.
            1. As before, check out the **application.yaml** file in the **trackpriceservice** directory.
               a) We give the service an application name. This is what clients will use to look up the service through Eureka.

b) We specify a port of 0, meaning a random port will be chosen
c) This service will be a Eureka Client, so we tell it the URL of the Eureka Server
d) The convoluted **instanceid** is a way to get a unique id for each instance of this service that we run, so we can run and register multiple instances with Eureka.

2. Run the service from **ttl.springboot.trackpriceservice.TrackPriceService**.
3. Observe the log while starting up. You should see messages at the end about registering with the Eureka server.
4. Refresh the Eureka server page. You should see the newly registered TrackPriceService

2) TrackInfoService – A service that takes a track name and gives back some "info". For now all it gives back is a rating for the track.

1. Go through the same steps as above. The service is in **ttl.springboot.trackinfoservice.TrackInfoService**.
2. Refreshing the Eureka web page should show the TrackInfoService also registered.

3) TrackIntegrationService – An integration service. The entry point into your application. This is what the outside user will get a reference to from Eureka. This service in turn will interact with the other two services to process requests.

1. The service is in **ttl.springboot.trackservice.TrackIntegrationService**.
2. Start up is as for the other services

4) TrackEndUserApplication – The client application. There are examples of two types of clients, one using Feign, and the other using the **DiscoveryClient**. The default is to run with Feign. A command line argument of "discovery" will run the DiscoveryClient version.

1. Client is in **ttl.springboot.eurekaenduser.TrackEndUserApplication**
2. The Feign client (the default) is set to call the track service 30 times, once every 10 seconds. This behavior will be useful to demonstrate Load Balancing and Circuit Breaking.

5) Load Balancing – The Feign client does client side load balancing by default.

1. Run more than one instance of any (or all) of your services. For example, maybe your TrackInfoService gets a lot of traffic so you want to balance the load across multiple servers. Note – Eclipse is fine with letting you run multiple instances of a program, but IntelliJ needs some encouragement. Go into the "Edit Configurations" menu item from the drop down that shows the running program, and check the "Allow parallel run" checkbox in the upper right corner.
2. Check on the Eureka server web page that multiple instances have been registered for your services.
3. Run the TrackEndUserApplication in Feign mode and observe the uuid of the responding servers. They should be switching from one to the other.
4. Kill one of the services.
5. You should now see only the single remaining service being used.
6. Start up another instance again and, after a minute or so, you should see the new service come into play. You may need to run the client again if you time it wrong.

6) Circuit Breaker – The idea here is that if a particular micro service starts missbehaving, for example gets really slow, or goes down completely, this can eventually lead to cascading failures throughout your system. The application uses **Hystrix** as the circuit breaker implementation by including it on the pom file. Putting it on the class path and the using a bunch of annotations does some pretty nice magic.

1. In the **trackservice** directory, look at the two Feign client classes – **ttl.springboot.trackservice.TrackInfoClient and ttl.springboot.trackservice.TrackPriceClient.**
2. The **@FeignClient** annotation allows you to pass in a fallback class, which is also in the same file. The method in the fallback class serves as the fallback method, in case the call to the actual service errors or takes too long.
3. Make sure all the services are running, then run the **EurekaEndUserApplication** with a command line argument of "feign". After it starts up and goes through a cycle or two, shutdown all running instances of the Price or Info service.
4. You should see the fallback method being invoked for that service.
5. Start up the service again.
6. After some delay (~50 seconds on my machine) you should see the service start producing results again.

7) API Gateway – Currently our end clients get in touch with the TrackIntegrationService directly. It is often useful and safer to have end clients get in touch with an *edge* service, also called an API Gateway. The client has no knowledge of actual service and where they run. We are using **Zuul** as our edge service. The code is in the **zuulservice** directory.

1. Start the Zuul server by running **ttl.springboot.zuul.TrackGateway**. That's it for Zuul.
2. Now we need to have the client get in touch with the gateaway rather than find it's services through Eureka.
3. Check out the **ttl.springboot.eurekaenduser.ZuulClient** in the eurekaenduser directory. The FeignClient annotation has a *url* parameter that points at the gateway. Note that the *name* parameter is a dummy, to keep Feign happy.
4. Start the **EurekaEndUserApplication** again, this time with a command line argument of 'zuul'.
5. The client is now talking to the gateway rather than Eureka to get the address of the track-service. Life is good.
6. You can play the same games with starting and stopping servers to see how the application behaves.