Kevin Chen

5/30/17

Learning Spring with Spring Boot

**Getting Started with Spring Boot**

Introduction to Spring

* The Spring framework provides comprehensive infrastructural support for developing enterprise Java applications.
  + This means that Spring removes lots of boilerplate code.
* Promotes good OOP practices and DRY (do not repeat) principles.
* POJO vs. Bean
  + POJO: an object that has both attributes and behavior
  + JavaBean: a simple POJO whose only behavior is getters and setters. To conform to the JavaBean specification, y9ou must have a default nullable constructor on your object.
  + DTO: a JavaBean that functions as a transport of data from one layer to another. Does not need to have a nullable constructor.
* Boilerplate Actions that Spring handles
  + Dependency management and injection
  + Data access and connection management
  + JEE integrations (security, messaging, remote access, and caching)
  + Controller integrations through MVC
* Aspecting: adds common behavior to a class at runtime or compile time
  + Spring uses aspecting for most of its abstractions.
  + All classes managed in the ApplicationContext are proxied or aspected in some way
* ApplicationContext is the heart and soul of a Spring application.
  + A wrapper around the BeanFactor, which stores all of the objects that Spring manages along with their definitions and proxies.
  + Serves Spring definitions via the factory pattern to the application at runtime
  + Contains all of the information about the application

Welcome to Spring Boot

* Provides rapid application development by removing much of the boilerplate code of just getting an app to run.
* Promotes cloud native or 12-factor app development, but it can also run in traditional environments.
* Achieves rapid application development by having an opinionated way of doing things. Most of these opinions should match with the way you do app development. But if your opinions differ, Spring Boot graciously gets out of the way.

Create a project with Spring Initializr

* Go to start.spring.io, the Spring Initializr website
* Input various configuration settings (click Switch to Full Version for more options). Group name typically should be com.*firstNamelastName*
* Various selections at the bottom that allow us to bring in opinionated dependency stacks into our application.
* The project this tutorial will create requires Web, Thymeleaf, JPA, H2 (provides embedded support for a database within our application), and Actuator (allows your container to have its own metrics, monitoring, and framework behavior).

Examine the Spring Boot skeleton project

* In the pom.xml, take a look at the dependencies. Each of the checkboxes we have selected when creating this project has translated into a starter project that is a dependency of our application.
* The Application file can be found in src/main/java.
* Running the project now will create a process on port 8080.
* Create resource files in src/main/resources/static
  + Create an HTML file (name it index.html).
* Run the application, and this HTML page will be loaded.
  + The console log shows a lot of mappings, which come from the actuator plugin
  + Go to localhost:8080/env to get the JSON payload that contains many of the environment specific variables and settings for the application
  + Go to localhost:8080/beans to get the JSON payload of all the beans loaded in the application context.
* The above information (env and beans) is very powerful. A bad actor could leverage this information to find a compromise in your application. So in terms of security, don’t expose the actuator endpoints to the external world (require a login to see this info).
* The application.properties file (located in src/main/resources) is where we can override any of the defaults within the application. Can also use YAML by changing this file to an application.yaml file.
  + Example property: server.port=8000 changes the port from 8080 to 8000.

**Data Access**

Spring Boot: Embedded database and configuration

* The example files provide a data.sql file (contains data to populate the database) and schema.sql file (contain database definitions).
* Running the application produces console output that shows it executed SQL scripts (both the schema script and the data script)
* In the application.properties file, the spring.jpa.hibernate.ddl-auto property is set to none. We need to stop hibernate from generating its own schema because it would wipe you’re your schema and data after it is loaded.

Welcome to Spring Data

* Spring Data provides a set of interfaces for a variety of datasources and technologies.
  + These interfaces connect with both RDBMS and NOSQL databases seamlessly.
* Provides a common method naming convention that enables you to swap datasources behind the scenes almost effortlessly.
* Provides a common interface for repository and data mapping abstractions (reduces boilerplate code).
* Allows you to focus on the business logic, not the data access code.
* Key Components: repository (an interface that adheres to the repository data pattern) and the entity (the object that represents the data structure of the table)

Build your first Spring Data Repository with Spring Data JPA

* Create a class that contains an attribute for each column of the table. These are entity classes.
  + Put the annotations @Entity and @Table(name=“*tableName*”) above the class name, where *tableName* is the name of the table in the database.
  + Put the annotation @Column(name=“*columnName*”) above each attribute, where *columnName* is the name of the corresponding column in the table.
  + Above the ID attribute, put an additional @Id annotation and a @GeneratedValue(strategy = GenerationType.AUTO) to have the ID field auto-generate
  + Add getters and setters for each attribute
* Create a repository interface that extends CrudRepository. CrudRepository takes in two type parameters: the type of object it stores and the type of the ID. These are repository classes.
  + There are various naming conventions (e.g. findByNumber) that builds query statements during runtime. Find more information about naming conventions here: <http://docs.spring.io/spring-data/jpa/docs/1.10.6.RELEASE/reference/html/#repositories.query-methods.query-creation>
* You can now test these methods, such as by creating a RESTful service controller.

**The Service Tier**

Build a service abstraction

* Why use a service abstraction
  + Has fewer class files (though may result in less clean code)
  + Allows you to expose data in more than one way
  + In RDBMS scenarios, it allows you to build complex transaction boundaries (e.g. where the rollback should occur if a failure occurs).
* How to build an abstraction
  + Start with a POJO to perform algorithms that will be exposed through the service abstraction layer
  + Annotate the POJO object with @Component or one of its various stereotypes
  + Can component scan or define the service abstraction

Develop a service object with Spring

* Create a new class for the service object. The service object combines multiple repositories
* Add the @Service tag above the class name.
* Add a private instance field repository you want in the service abstraction layer. These are our dependencies.
  + For example, if you want to keep track of room reservations, you would add a RoomRepository and a ReservationRespository instance variables into this class.
  + These could have been autowired, but these dependencies cannot be tested unless directly exposed through getters and setters.
* Add a constructor that takes in a parameter for each dependency and sets them.
  + Add the @Autowired tag to tell Spring that these are dependencies of the class. Spring will construct these first and inject these dependencies.
* Now create a domain class to avoid exposing the entities.
  + This class will combine the attributes of the relevant entities (again, use private instance variables, and generate getters and setters).
* Now go back to the service class. You can create various methods to obtain various lists of data in the format of domain objects.
  + For example, you can create a list of RoomReservations (a domain object) by a particular date using the Room and Reservation repositories.
  + Spring gives you a findAll() instance method in the repository variables, which returns an Iterable of all the entity instances within the repository.
  + You can create a collection of domain objects by using the iterable to iterate over the relevant repositories to obtain data. Create a new instance of the domain object, and set this object’s data to the data obtained from iterating over the repositories.

**Web Pages with Spring**

Introduction to the controller

* Model: the data that is being served to our dynamic view
* View: visual aspect of how we are presenting our data.
* Controller: traffic cop that collects the data for the model and passes it to the appropriate view via the template engine
  + The main abstraction of the Spring MVC application. (Has attribute @Controller)
  + Responds to all requests through RequestMapping. The DispatcherServlet uses the request mapping along with the HTTP verb to direct all traffic to a specific method.
  + Controls building the model and resolving the view.
* For Spring MVC, model = any data structure, view = HTML document, controller = Thymeleaf template engine.

Build our first controller

* Create a class for the controller.
* Annotate the class with @Controller.
* Add @RequestMapping(value=“*URL*”) to make the class accessible via the specified *URL*
* Now add a method that will respond to a GET request. Annotate this method with @RequestMapping(method = RequestMethod.GET)
  + First add a template HTML file inside src/main/resources/templates to show display this file to provide to Thymeleaf to render and display.
  + In the method, return the name of the HTML file. The value you return is going to get translated by the Thymeleaf engine into the name of a template.

Thymeleaf as a rendering engine

* Thymeleaf provides a variety of operators that can be embedded in HTML
  + ${*variableName*} is used to access a variable that is from the model or created in the HTML.
  + th:each=“*currentObject*: ${*collectionOfObjects*}” creates an iterator that will loop through the collection.
  + th:text=“*someString*” sets the text of the HTML element to *someString*, overwriting what is originally there.

Put it all together

* Access parameters from the URL using @RequestParam(value=*valueName*) String *nameOfParameter*.
  + Example:

public String getReservations(@RequestParam(value=“date”) String dateString) {

//Lines of code…

}

* + @RequestParam(value=“date, required=false) would make this variable optional.

**Notes for this section is incomplete.**