**: SPRING Boot:**

* **Introduction :**

Spring Framework helps to develop enterprise applications whereas Spring Boot helps to quickly startup the spring application (simple and complex applications) end to end ready to production environment.

* **In this course:**



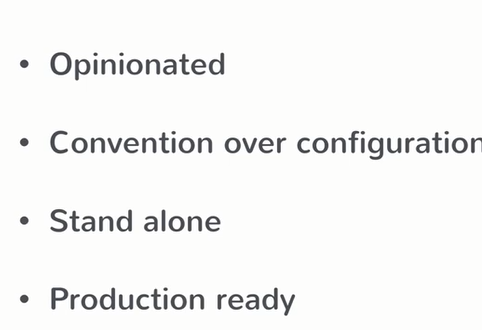
* **What is Spring Boot:** Spring Boot is nothing but something helps to boot strap Spring framework for building your enterprise applications. **Spring Boot it’s easy to create stand alone, production grade spring based applications that you can “just run”. Tool that creates Spring Based applications.**
* **What is Spring and it’s Problems:** It’s an application framework, provides the template to build enterprise applications / Modular programming – just concentrate on modules – Programming and configuration model. Provides Infrastructure support (helps to integrate with Hibernate / struts / Mongo DB ..other existing frameworks)

**Problems:** Huge Framework / Multiple setup steps (to integrate with any existing frameworks) – you need to understand the various configuration details/steps / Multiple Configuration steps / Multiple steps to build and deploy.

If we overcome these problems, Spring framework is a beautiful framework. Spring boot helps to overcome these hurdles to build a production ready spring applications.

* **What Spring Boot gives you :**

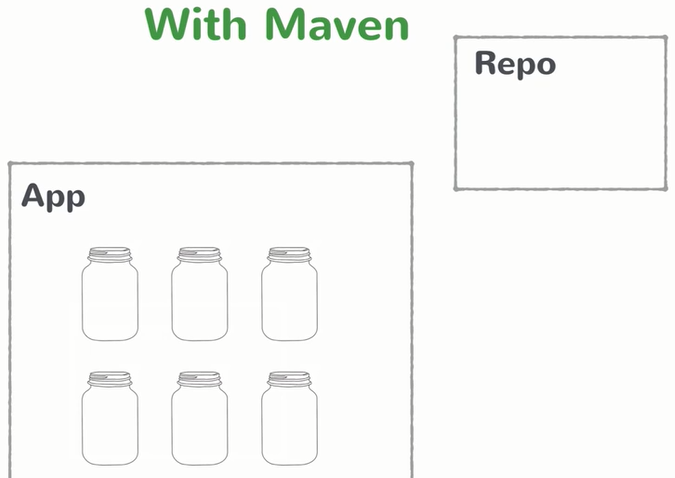
Spring boot is opinionated – start from one point and try to improve / add from that. With Spring boot it helps for easy configuration / You can build Standalone application (With regular spring application is not standalone – you need to build the web application – build WAR and deploy in Tomcat or other App servers). Whereas in Spring boot helps to build the standalone application (PRODUCTION ready).



* **Setting up Spring Boot: IDE**

**IDE : STS (Spring Tools Suite) (Download from spring.io)**

* **Maven :**

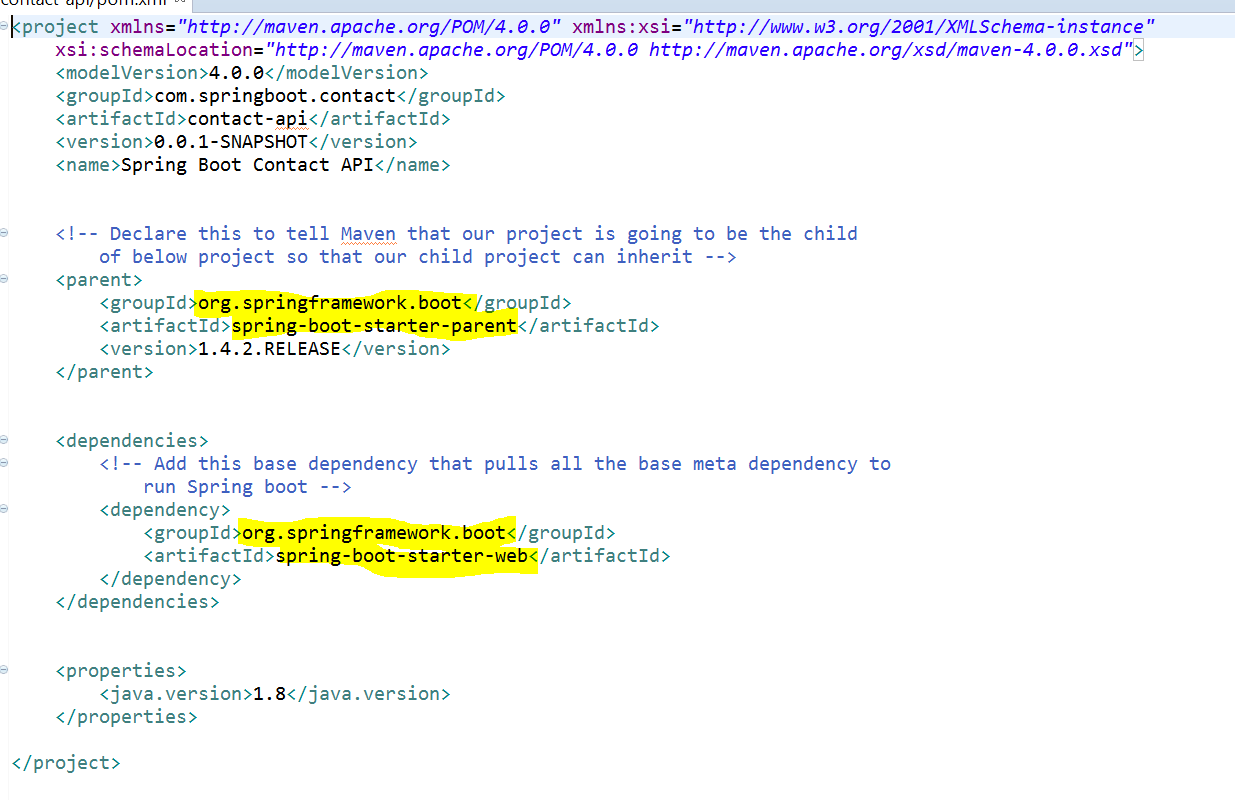


For this project we are going to use MAVEN – going to add all our dependencies in pom.xml requires to run our project / Helps to download the jars from REPO and added to project class path.

* **Creating Spring Boot Project:** Multiple approaches are there to create a Spring boot project, we are going to try each approach.

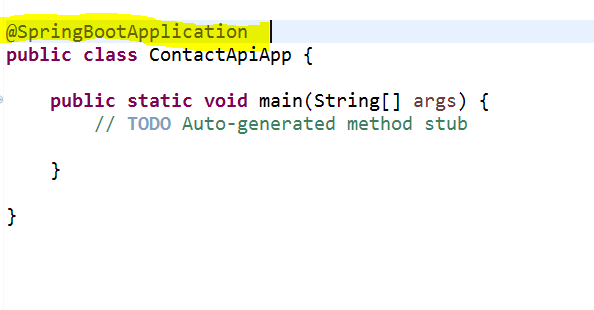
**Step 1:** First create a simple MAVEN project

**Step 2:** After creating theMAVEN project add the below entries in the pom.xml

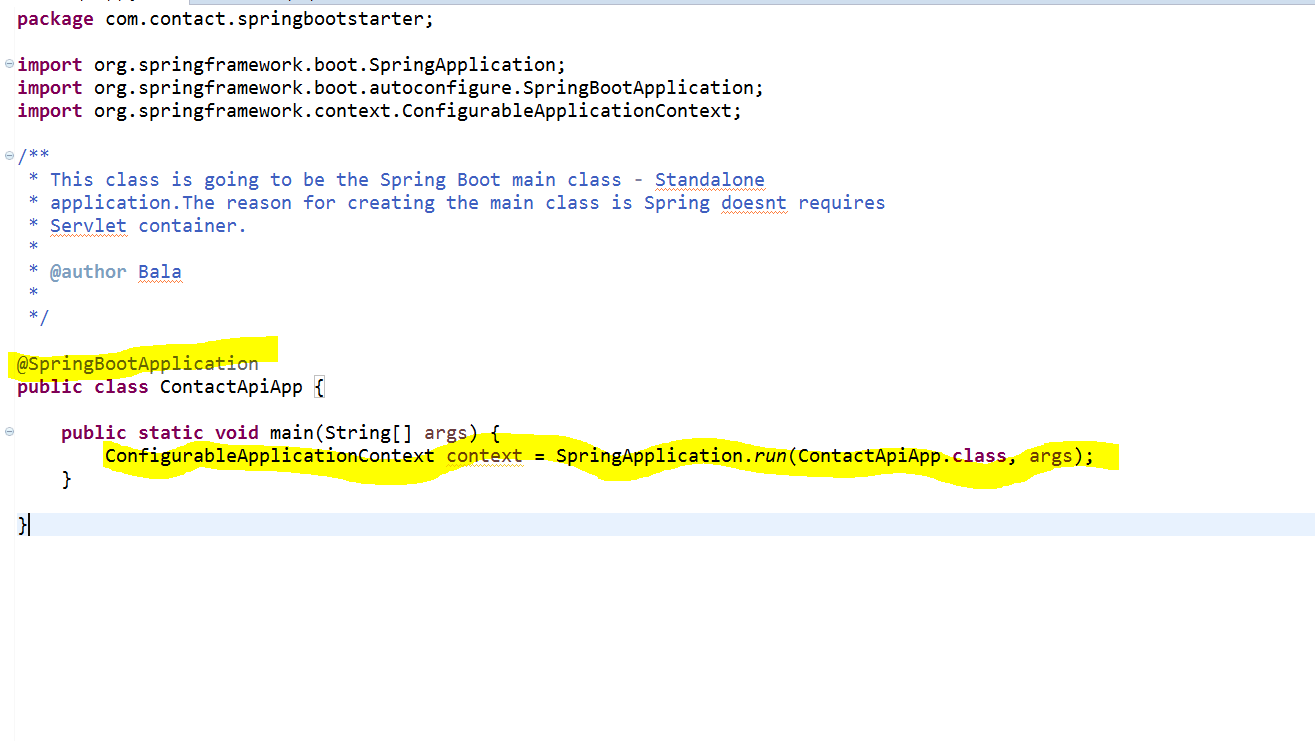


* **Starting a Spring Boot application**

**Step 1:** We need to have a main class which basically the boot strap for Spring / We already mentioned that Spring boot doesn’t requires servlet container to deploy. So the main class will be the Boot strap standalone application. We need a way to tell that the defining main class is the spring boot standalone application, for that we need to use the annotation ‘@SpringBootApplication’ (tells that it’s the starting point of the Spring boot application)

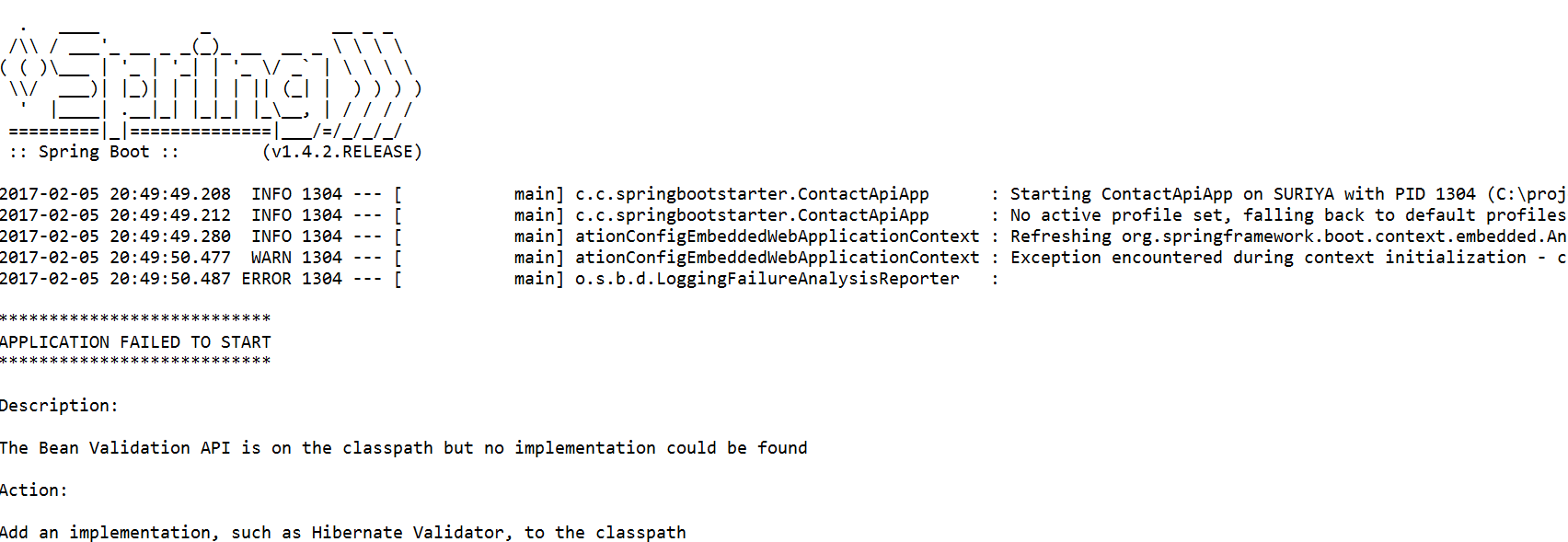


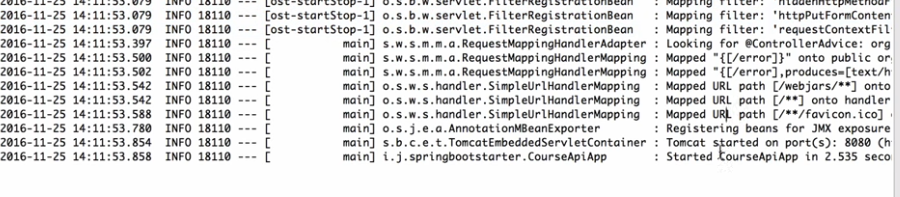
**Step2 :** Next is we need a way to tell the Spring Boot to start the application / create a Servlet container and deploy the app in the created servlet container and host it to handle the request …this seems lots of code to achieve…but the spring boot easily achieve this by



ConfigurableApplicationContext context = SpringApplication.run(ContactApiApp.class, args);

That’s it the Spring Boot application is ready , when you run this the Spring Boot gets started





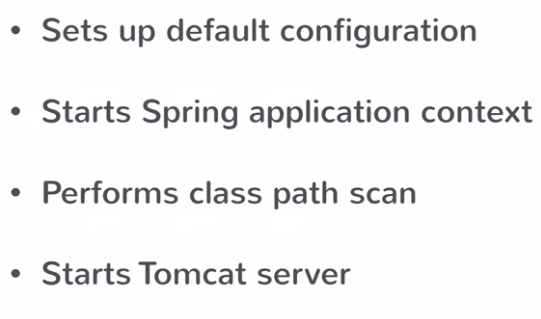
See when you started the application , embedded tomcat servlet container started @ port 8080

* **Spring Boot Startup Steps:**

The magic line is

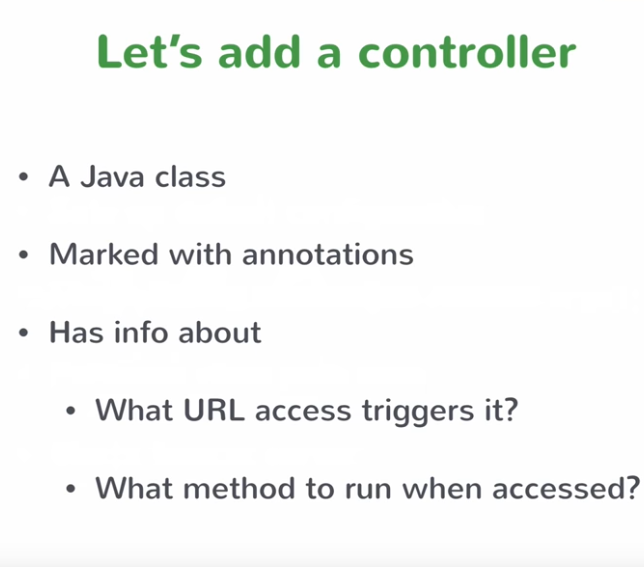


It addresses the 80% use case, completes the default configurations. It starts the application context (Spring Container is called as application context, every spring application has application context) so the Spring boot starts the application context. It also performs the class path scan – the way to plugin your custom classes based on the annotation (if its service annotation treat that class as Service class / if it’s a controller class then treat it as a Controller Class) so the spring boot will scan / infers based on the annotations. Finally it starts the Tomcat Server (This came with Spring boot that’s the reason is called as Standalone).



* **Adding a REST controller :**

We are going to create a Controller and maps to a URL.



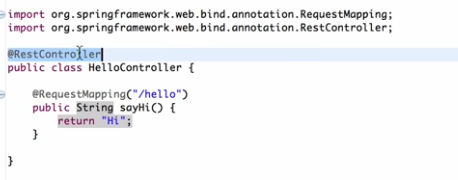
Let’s say you have a URL localhost:8080/hello.So you need to map the ‘/hello’

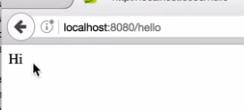
We are going to create a simple MVC controller. (It’s going to be a REST controller in Spring)

It means you are going to map the methods based on the URL request.

@RequestMapping methods maps to all HTTP URL request to methods

@RestController helps to identify the REST controller



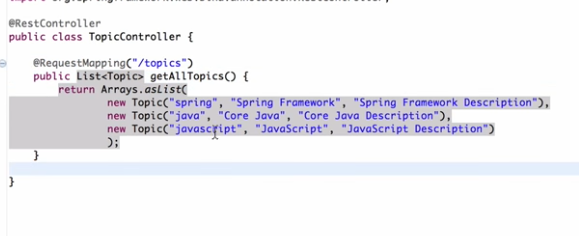


* **Returning objects from Controller : (More controller Mappings)**

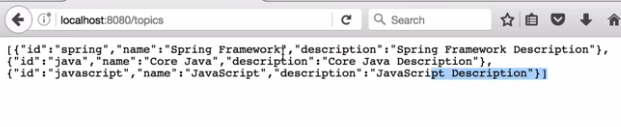
Going to add a ‘TopicController’ request with ‘/topics’ comes in then the controller going to list all the list of topics

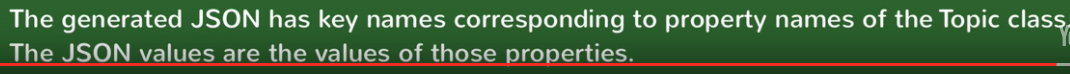
The Controller going to interact with the database to get the list of the topics. At this point lets mock the data’s and return the list of ‘Topic’ objects , this automatically converts to JSON and returns





It automatically converts to JSON response.

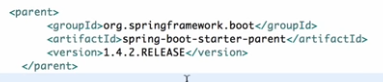




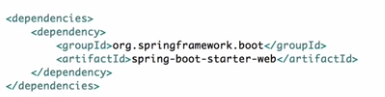
* **Understanding Bill of Materials**

What’s happening in the background, spring does everything for us in the background? Let’s try to understand in details

Let’s start from pom.xml



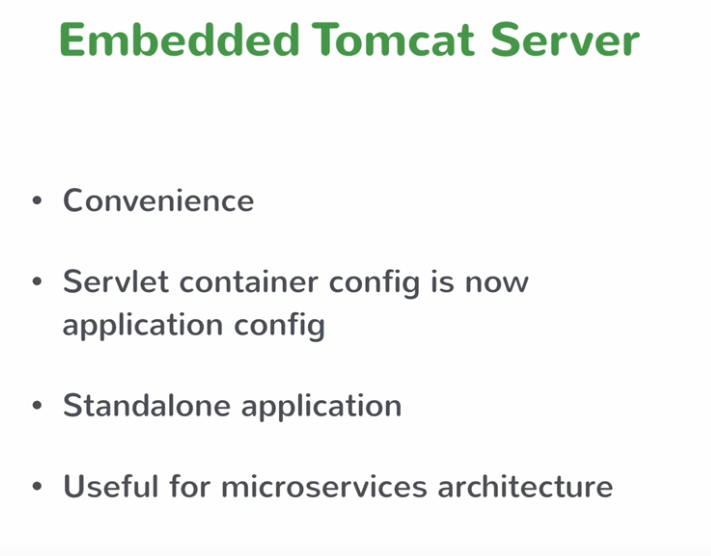
The spring boot starter parent imports only the configurations / whereas the ‘spring-boot-starter-web’ dependency imports all the necessary spring related jars (spring-core / spring-aop / spring-beans/spring-context/spring-webmvc/embedded tomcat servers….)



That’s how spring boot provided pre-sets of possible combination jars called as ‘Bill of materials’. Based on the version it picks certain combination of jars so pick the right version of dependency to work with those set of features.

* **What’s happening here : Embedded Servlet Container**

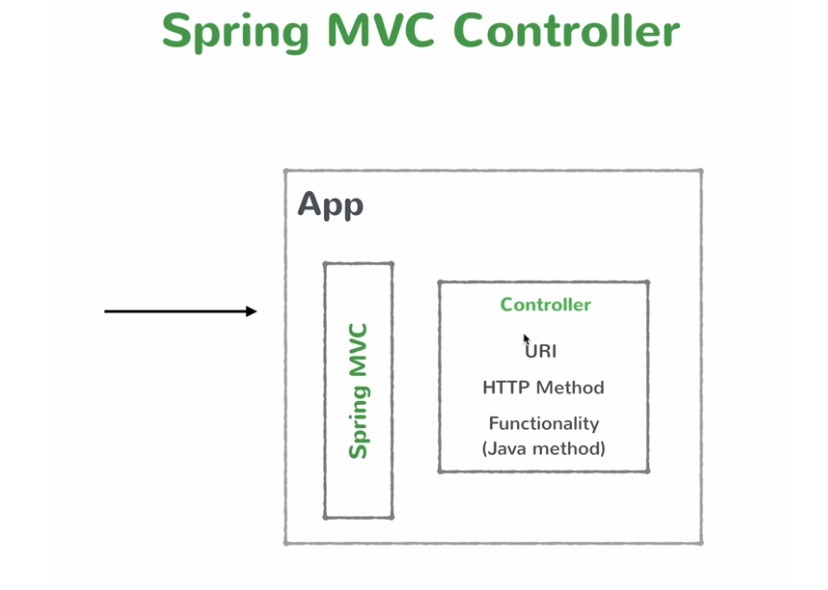
Understanding Embedded Servlet container here: We don’t need to download the tom cat servlet container to deploy our application / Spring boot comes with embedded servlet tom cat container mainly because of convenience / Servlet container configuration steps involved in Tomcat ..Now the servlet configurations should be done as part of application configuration (just like configuring as part of your application source code ...you need to include servlet configurations also. This really helps to achieve the standalone applications. It’s useful for micro services architecture. If you have one big module of service, you can take a pain to deploy the services, if you are using micro services 10 small services then embedded servlet container really helps to deploy it for you. These are the following reasons Spring boot comes with servlet container, if you want to change the servlet container then you need to include the dependency in your code base.



**PART 2: SPRING MVC: The View Tier:**

* **How Spring MVC works: Spring MVC Controller**

View tier is handled by Spring MVC module.



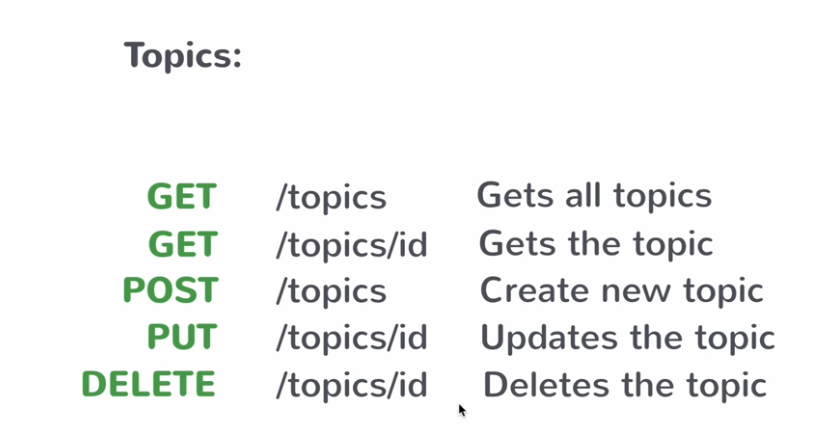
The flow of controller is as follows :

* Any HTTP requests (GET / POST / PUT / DELETE /..) intercepted by the Spring MVC based on the request URI (/contacts , /comments….) forwards to the corresponding java method.
* After the method process (database call/ service call) finally the Controller returns the response object to the Spring MVC
* The Spring MVC converts the response objects to JSON response, you don’t need to worry about converting the response object to JSON..Spring MVC automatically does this for you.
* **The REST API build :**

REST API mainly deals with nouns / You need to identify the resources / After identifying the resources you can then look at the ways how the consumer access those Resource URI’s this is the standard REST.



First let’s cover Topic / let’s use the hardcoded data’s. (Course API’s)

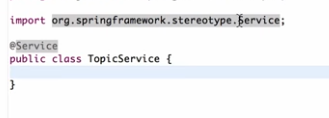


* **Creating Business Service :**

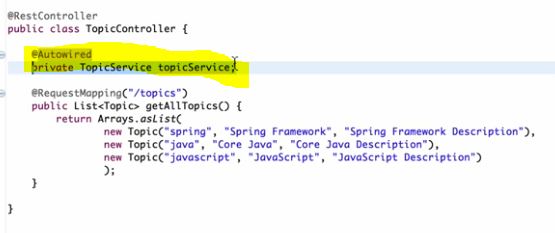


Now let’s see how to build the business service, you start with the simple java class called ‘TopicService’ Now we need this class to be business service. In spring the business service are Singleton classes, spring injects to different controllers.

We will use **@Service** to mark the class as business service.

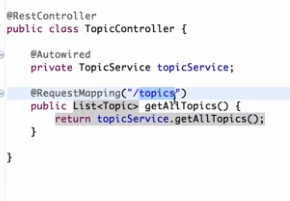


Now let’s see how you inject the service in your controller

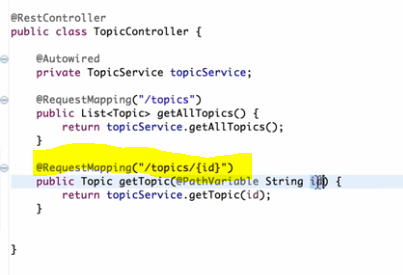


Just declare as private service variable, in order to inject use @Autowired injection. Spring scans the package and injects the service class in controller

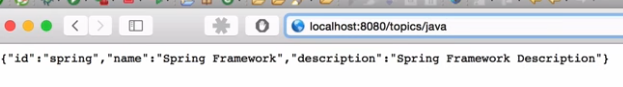




* **Getting a Single Resource :** Here lets implement the logic to return just requested topic based on the ID

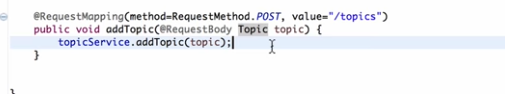




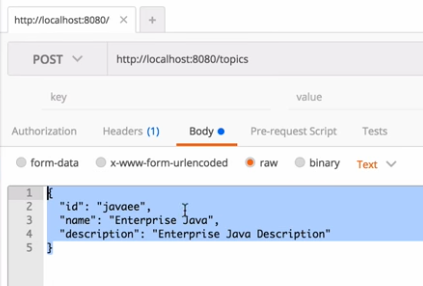


**Creating new resource with POST:**





Constructing the body

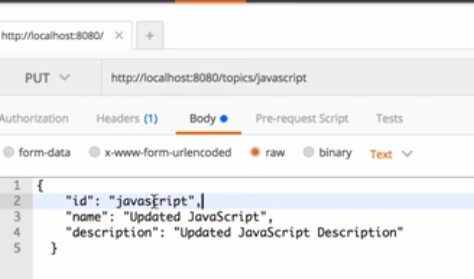


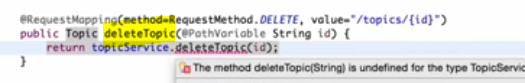
If you pass the JSON request body, Spring MVC automatically converts it for you as Topic object.

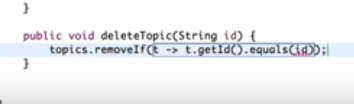
* **Implementing Update and Delete :**

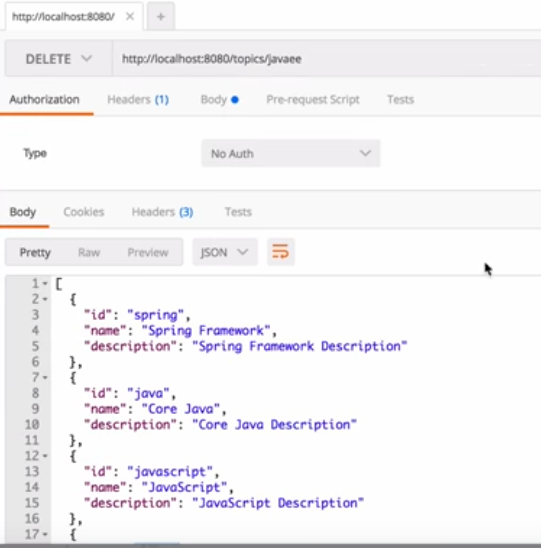


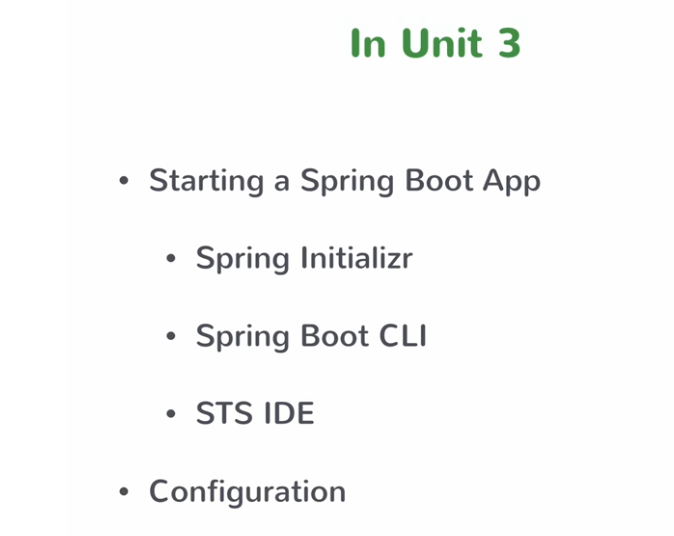










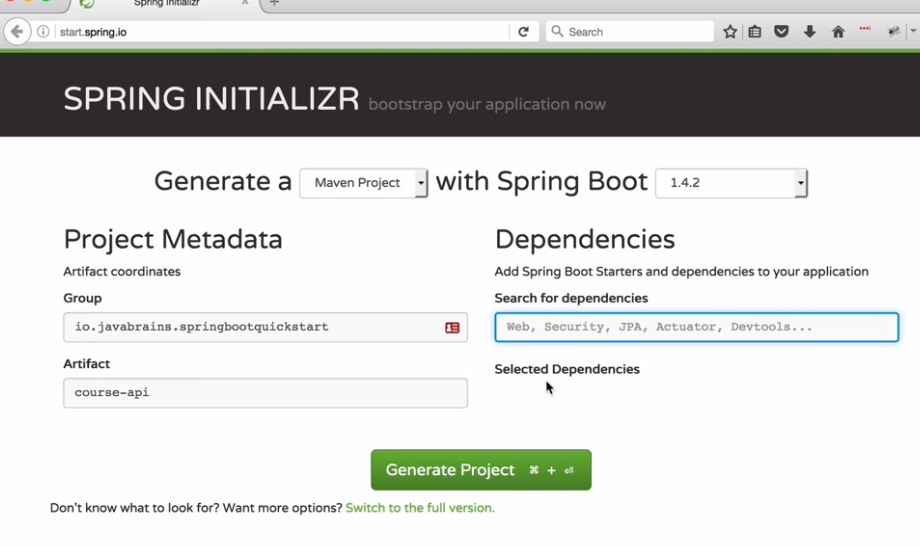


**Unit 3: Booting Spring Boot**

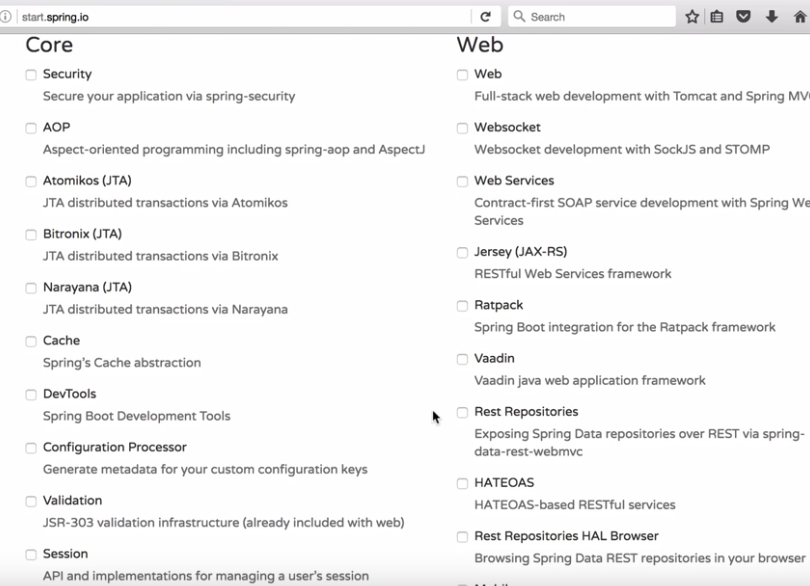
We saw one way of booting spring application, there are lot of ways to implement. Let’s see the couple of other ways, we can pick based on our requirement / choice

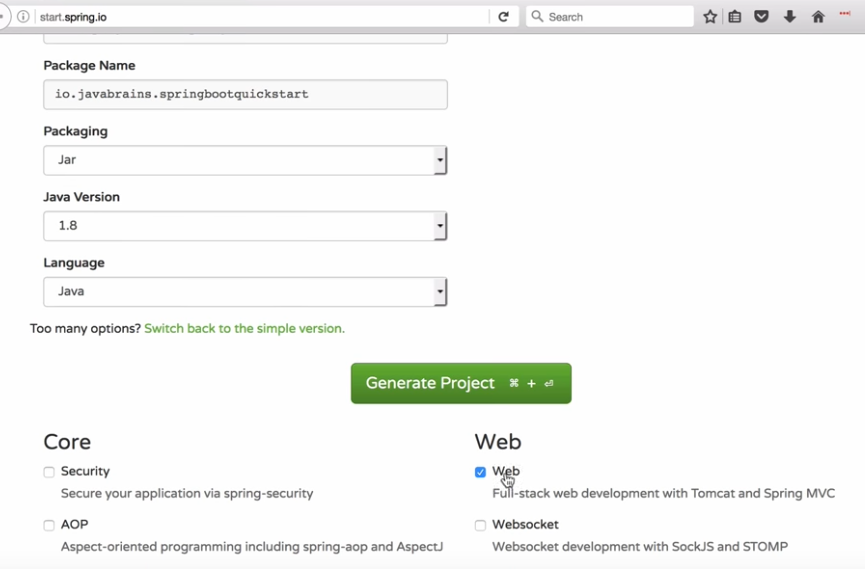
* **Using Spring Initializr**

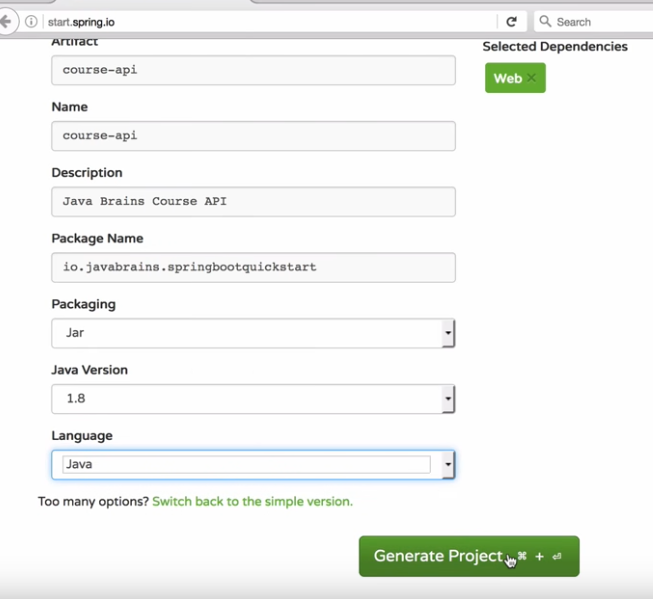
In start.spring.io you can specify the details to boot strap the Spring Initializer like below, you can choose the list of dependencies that you want to configure for your project



**The list of dependencies that you want to configure for your project**

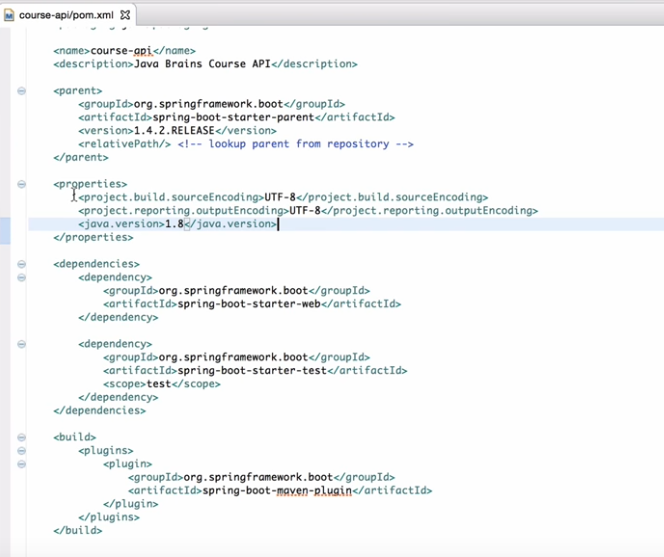






After download, you need to import as ‘Maven’ project . (There is also option to generate as ‘Gradle’ project).

After you download, when you check the pom.xml, you will see those dependencies



It also comes up with the main class (Boot class written)

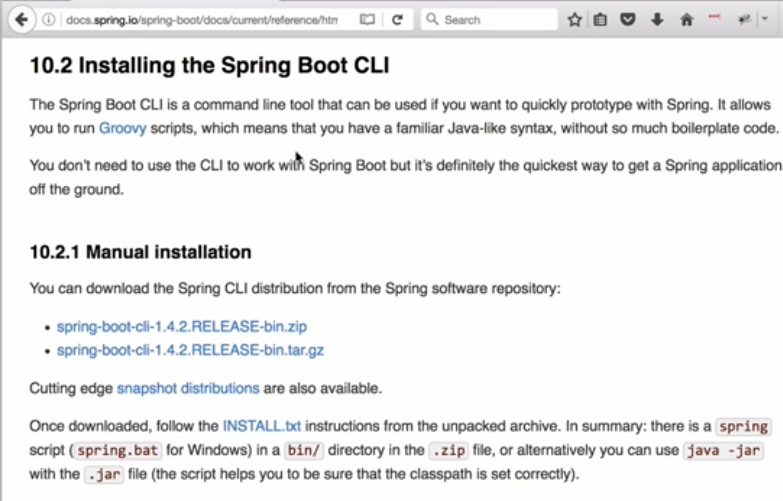


The name of the application helps to initialize the boot strap code is ‘Spring Initializr’ available in the URL start.spring.io

* **Using Spring Boot CLI**

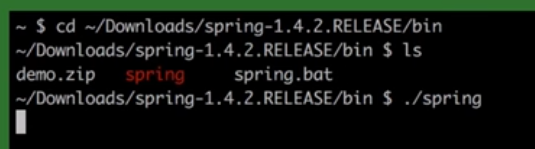
Command line tool used to quickly configure the Spring Boot Strap for your application.

Got to the below URL to download the spring-boot-cli. This also helps to run GROOVY scripts



You need to install and manually install Spring Boot CLI

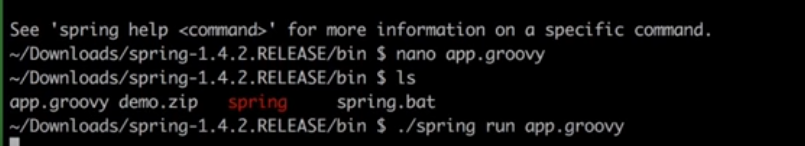


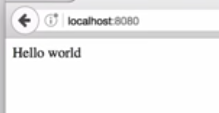


What is Spring Groovy script : It will create a complete Spring Project. You just need to create the Controller class to start without SpringBoot main class here thru Groovy script



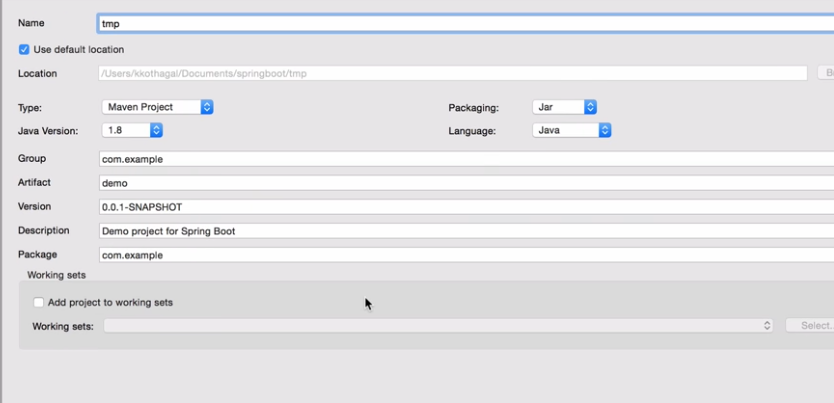
You can directly run the script



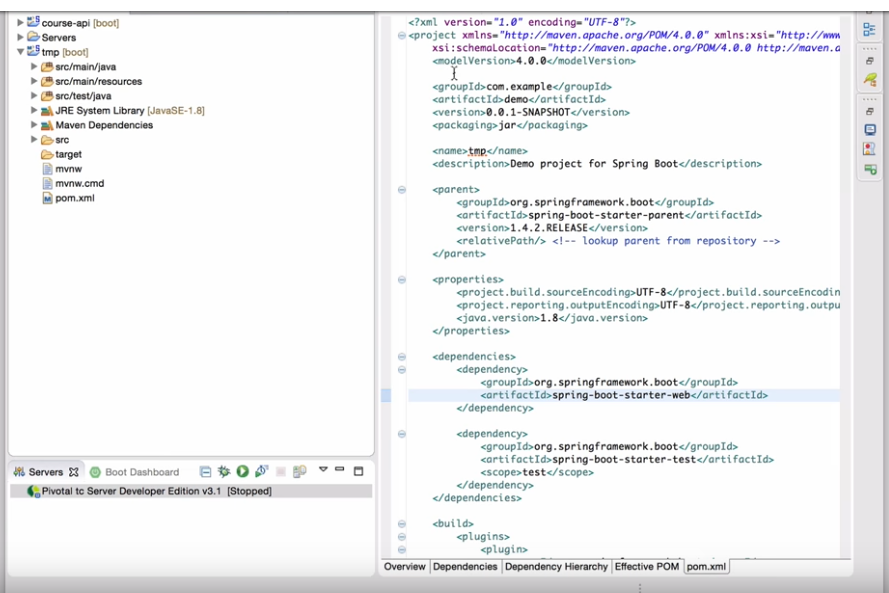


* **Using STS IDE : (Most convenient way)**

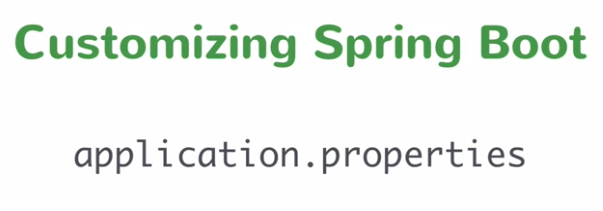
File >> New >> Spring Starter Project (From Spring Tools Suite)



It creates the entire project for you

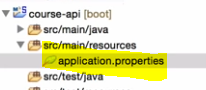


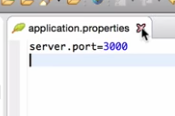
* **Using application properties :**



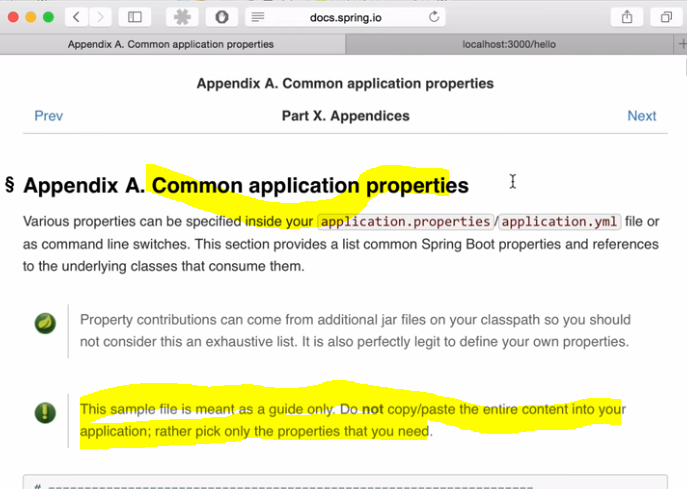
Easy way to do is application. properties. In spring application, common way to specify the property files under resource folder.

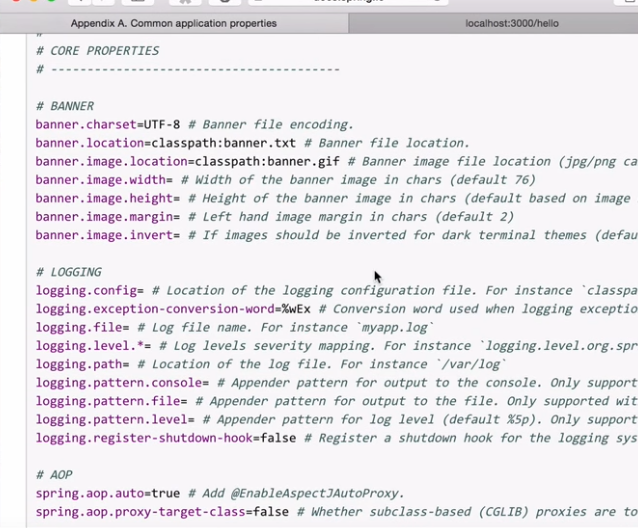
Place a property file in your project and change the default servlet container other than 8080





Now run the application. You need to know what are all the properties to be added in the application.properties to spring boot your application

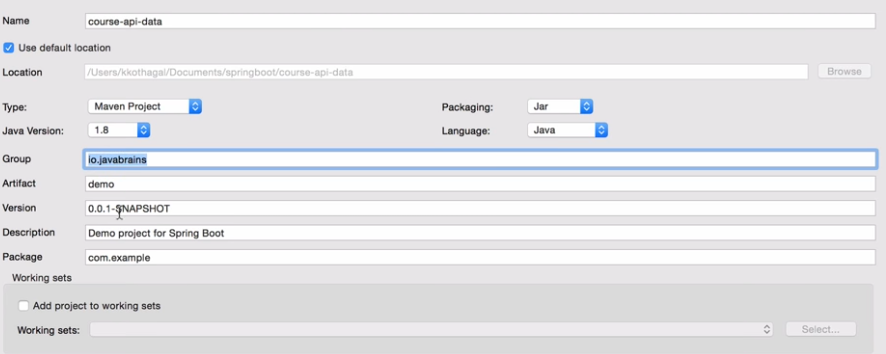




**Unit 4: What is JPA (Data Tier)**

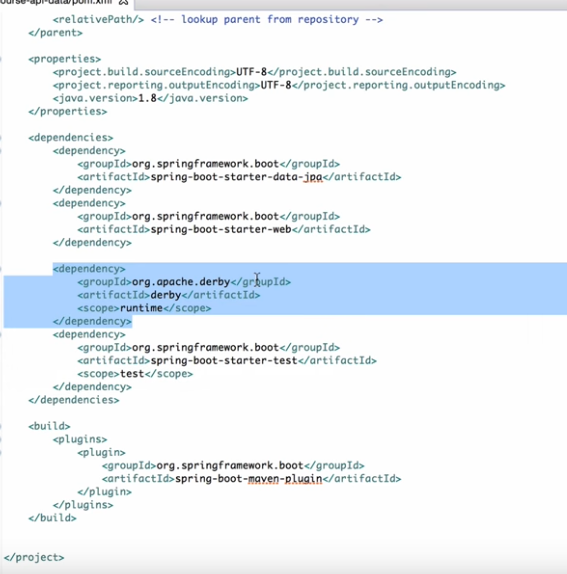
* **Spring JPA : Java persistence API :** basically specification that let you do ORM connect to Relational database. SQL database is the relational database , when you need to connect from application you need to use JDBC , once you connect you can run queries. In ORM maps the objects and tables thru metadata information . (Class and Table mapping) JPA specification way to use the ORM. This is JPA only, now let’s look into Spring JPA.
* **Adding Spring Data JPA :**

Add few dependencies in your spring boot web project to support database. File >> New Spring Starter Project





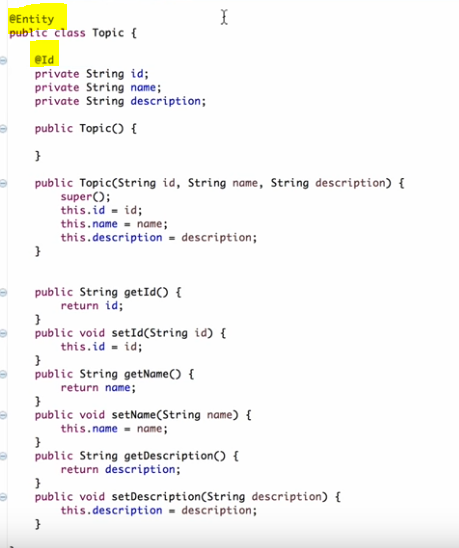
It supports the embedded database, not for production environment. For development it really helps



* **Creating JPA Repository**

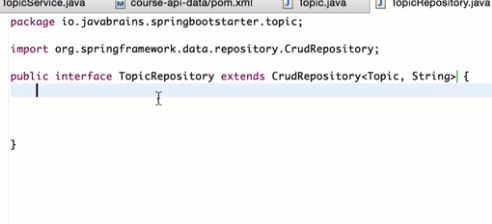
We are going to remove the hardcoded data in the service class and going to connect the JPA / database connectivity to bring real data’s

Identify the Entity class (Identify the object to be saved to database)



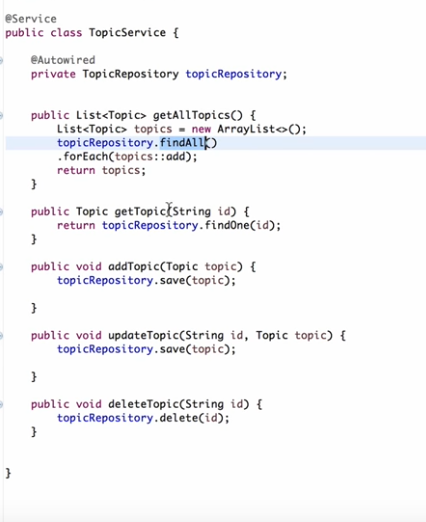
The framework already builds the database standard methods (get / getall / update / delete / insert..)

So what you need to do is create a interface in your application that uses the implementation provided by the Spring Boot (CrudRepository provided by Spring Boot out of box)



* **Crud operations with Repository :** All the CRUD operations are handled in CrudRepository



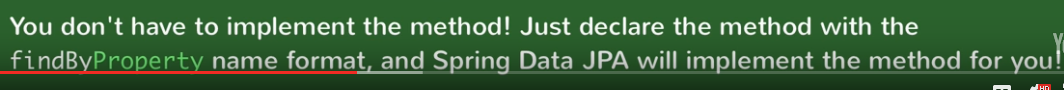


* **Extending Repository**

How to we let the spring data JPA know ‘Topic’ is basically foreign key to the primary key of the Topic table. How to get the Course Service based on Topic



Hibernate reads the above annotations and know what to do



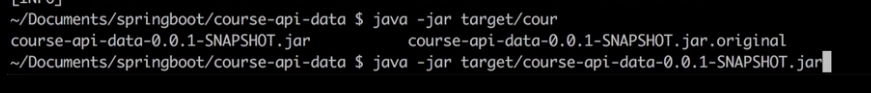
**Unit 5: Packaging and Running**



How to deploy in production environment. We don’t need servlet container / makes it very flexible. Just we need JRE / application bundles the servlet container

Once you done with all the changes – build the JAR using MAVEN build

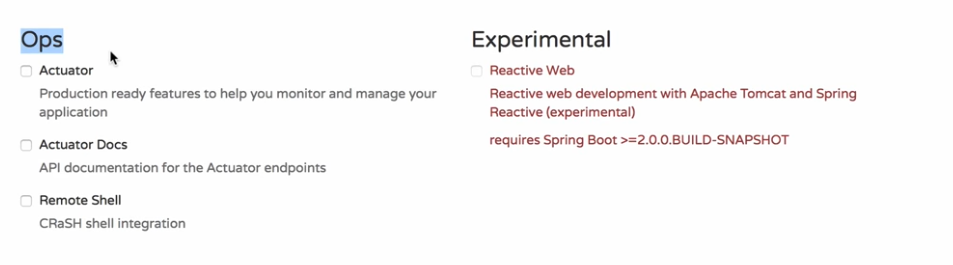
Run usiong below comment

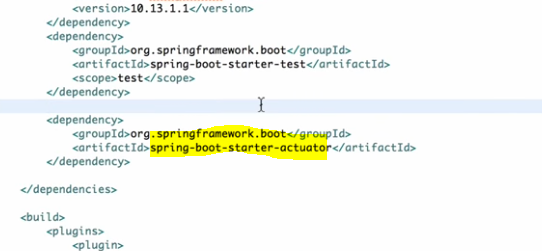


Easy to deploy the Micro services. It also supports WAR.

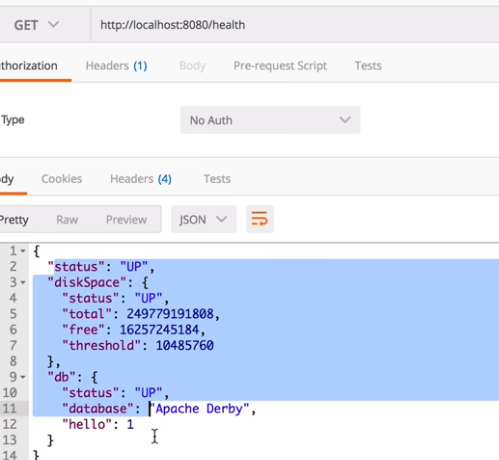
* **Spring Boot Actuator**

Bunch of monitors helps to monintor your spring boot application. Go to ‘start.spring.io’





Actuator group of binaries in your class path added



JSON file you can check the bunch of methods gives us the entire status / new end point added to our application to check the healthiness state of our application.

* **Real Examples:**

Pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.4.2.RELEASE</version>

</parent>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<version>1.5.2.RELEASE</version>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

<configuration>

<classifier>exec</classifier>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

<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>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.2.0.Final</version>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.8.0</version>

</dependency>

<dependency>

<groupId>org.jvnet.ws.wadl</groupId>

<artifactId>wadl-core</artifactId>

<version>1.1.6</version>

</dependency>

<dependency>

<groupId>org.jvnet.ws.wadl</groupId>

<artifactId>wadl-client-plugin</artifactId>

<version>1.1.6</version>

</dependency>

**Controller**

@RestController

@RequestMapping(value = "/api/users")

@EnableAsync

@CrossOrigin

**public** **class** RegistrationController {

/\*\*

\* Profile Service Object auto injected

\*/

@Autowired

**private** ProfileService profileServiceObj;

/\*\*

\* ProfileUtil object auto injected

\*/

@Autowired

**private** ProfileUtil profileUtil;

/\*\*

\* GeneralUtil object auto injected

\*/

@Autowired

**private** GeneralUtil generalUtil;

/\*\*

\* Logger object to log the application related messages

\*/

**private** **final** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());

/\*\*

\* variable to hold the password for Truststore.Property value injected from

\* application property

\*/

@Value("${cas.activate.url}")

**private** String casActivationUrl;

/\*\*

\* Default empty constructor.

\*/

**public** RegistrationController() {

}

@CrossOrigin

@RequestMapping(value = "/v1/profile/register", method = RequestMethod.***POST***, consumes = {

"application/JSON" }, produces = { "application/JSON" })

**public** RegisterResponse registerUser(@RequestBody RegisterRequest registerRequest) {

**IMPL Class:**

@Component

**public** **class** ProfileServiceImpl **extends** BaseService **implements** ProfileService {

/\*\*

\* Logger object to log the application related messages

\*/

**private** **final** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());

/\*\*

\* ProfileUtil object auto injected

\*/

@Autowired

**private** ProfileUtil profileUtil;

@Autowired

**private** ProfileDao profileDaoImpl;

@Autowired

**private** CrmDao crmDaoImpl;

**DAO Class:**

@Component

**public** **class** ProfileDaoImpl **extends** BaseDao **implements** ProfileDao {

/\*\*

\* Logger object to log the application related messages

\*/

**private** **final** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());

@Autowired

**private** GeneralUtil generalUtil;

@Autowired

**private** EnvironmentConfig environmentConfig;

/\*\*

\* This method invokes the SOA call to get the profile details

\*

\* **@param** getProfileByIdRequest

\* **@return**

\*/

**public** GetProfileResponse getProfileById(GetProfileByIdRequest getProfileByIdRequest,String tenantId) {

GetProfileResponse response = **null**;

**Property / Environment Related (Reads from application.properties)**

@Component

**public** **class** EnvironmentConfig {

@Value("${soa.endPointUrl}")

**private** String endPointUrl;

@Value("${soa.insertProfileUrl}")

**private** String insertProfileUrl;

@Value("${soa.updateProfileUrl}")

**private** String updateProfileUrl;

<http://blog.arungupta.me/microservice-design-patterns/>