**Java basic training guidelines**

Author: Nikolay Kandev

# **Purpose**

The sole purpose of this document is to provide guidelines for establishing basic understanding of Java development process and few of the most widely used frameworks. The training process is designed to be self-executed, but whenever there are questions or problems that cannot be resolved through googling please refer to the nearest Java developer for help.

# **Preparation**

This chapter will cover the software that will be needed during the training.

* Download and install the latest Java platform JDK <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
* Download and install the latest Eclipse IDE for Java EE developers

<https://eclipse.org/downloads/>

* Download and install the latest Apache Tomcat Server

<http://tomcat.apache.org/>

* Download and install MySQL community server and MySQL community workbench

<http://dev.mysql.com/downloads/>

* Open eclipse and configure the Apache Tomcat Server – just add it to the Server view
* Download and install SoapUI – for testing of web services

<https://www.soapui.org/downloads/latest-release.html>

* Install Postman Chrome plugin – for testing rest services

# **Theory**

The first half of the training is all about gaining knowledge about Java 8 and SQL as both languages are widely used in the enterprise projects.

* **Java language**

[**OCA Oracle Certified Associate Java SE 8 Programmer I Study Guide Exam 1Z0-808.pdf**](file:///\\10.0.0.42\Volume_1\Books\Java\OCA%20Oracle%20Certified%20Associate%20Java%20SE%208%20Programmer%20I%20Study%20Guide%20Exam%201Z0-808.pdf)

See also the Java 8 Streaming API:

<http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

* **SQL**

<http://www.w3schools.com/sql/>

# **Practice**

This chapter is designed to develop knowledge about some of the basic concepts and most commonly used frameworks by applying them to practice through developing a simple application project. Detailed guidelines will be provided for each component of the application.

## **Requirements:**

A client needs an application that manages a client database. A RDBMS (relational database management system) should be used as a storage engine. Five basic operations need to be supported

* Create a client
* Update a client by id
* Retrieve a client by id or email
* Delete a client by id
* Retrieve all clients

These operations should be exposed through two different interfaces

* REST over HTTP
* WebService over HTTP

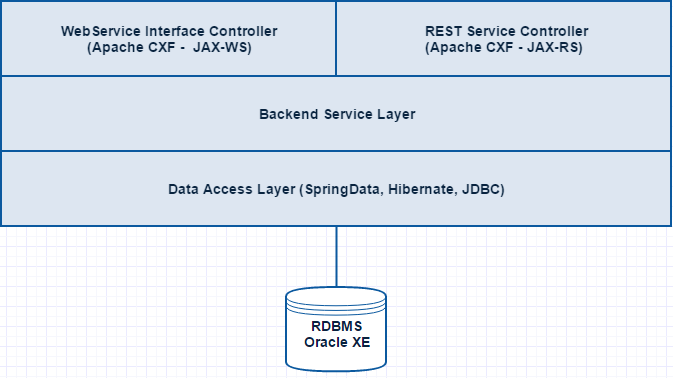
The client object contains the following fields

* Id (system generated guid)
* Email
* First Name
* Last Name
* Addresses – list of address objects. Each address has: Country, City, ZipCode, AddressLine

## **Design and project skeleton**

**Step 1**: Design and create a database structure that can hold the client information and create it in the MySQL database by using MySQL Workbench (or other db client).

**Step 2:** Review the project’s high level architecture



**Step 3:**  Create initial java project.

* Open Eclipse
* Create a new Maven project

Uncheck “Create a simple project” checkbox in the wizard and click “Next”

Group id: com.scalefocus.edu

Artefact id: ClientStore

Packaging: war

* Create the following package structure in your project:

com.scalefocus.edu.config – keeps the configuration classes

com.scalefocus.edu.db.dao – keeps the DAO(Data Access Object) classes

com.scalefocus.edu.db.model – keeps database bean classes

com.scalefocus.edu.api – keeps the api common interfaces

com.scalefocus.edu.api.model – keeps interface bean classes

com.scalefocus.edu.api.ws – keeps web service controller classes

com.scalefocus.edu.api.rs – keeps rest service controller classes

com.scalefocus.edu.service – keeps the backend service classes

* Get familiar with Maven’s basics

<http://www.javaworld.com/article/2072203/build-ci-sdlc/an-introduction-to-maven-2.html>

* Get familiar with Spring dependency injection concept <http://www.vogella.com/tutorials/SpringDependencyInjection/article.html>

**Step 4:**  Configure java based application initializer with spring (avoid XML configurations).

We want to avoid having web.xml file and whole bunch of other configuration xml files so we are going to use Java configurations for everything, which makes the solution much cleaner. To achieve that add the following snippet to your *pom* file.



Now we have everything we need to initialize our web application without web.xml file.

Get familiar with the concept here

<http://docs.spring.io/autorepo/docs/spring-framework/3.1.x/javadoc-api/org/springframework/web/WebApplicationInitializer.html>

So, let’s create our initializer classes:



And the second one containing the Spring configuration annotating which packages should be scanned by the spring container.



In our case we have told to the container to load all annotated beans (having annotation @Service, @Controller, @Component, etc.) in the context so they can be @Autowired.

**Step 6:** Backend service

Basically – the API classes calls the backend service which calls the dao service. Now we are going to create a dummy backend service just for demonstration.



**Step 7:** XML & Web service interface

Get familiar with the concept using the following links

XML – <http://www.w3schools.com/xml/>

XSD - <http://www.w3schools.com/xml/schema_intro.asp>

XSLT - <http://www.w3schools.com/xsl/>

Web Service - [https://en.wikipedia.org/wiki /Web\_service](https://en.wikipedia.org/wiki%09%09/Web_service)

WSDL - <http://www.w3schools.com/xml/xml_wsdl.asp>

JAX-WS - <https://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services>

Alright, now as we know how the Web Services work we are going to build one.

Apache-CXF is a framework that implements the WebService concept and we are going to integrate it in our simple application.

First, add CXF dependencies to our pom file:



Now let’s create our API interface and implementation class:



**

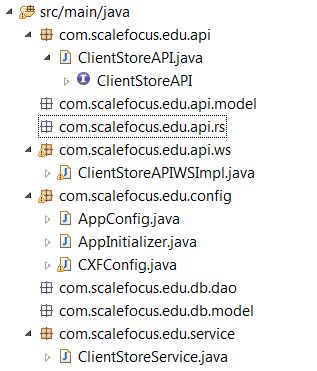
**Pay attention to the annotations that were used.**

Now we have to configure the CXF context to load our Web Service API correctly.

Create and carefully review the following configuration class:



If you look at your project it should look similar to the picture bellow:



Now deploy the application. Our web service should be listed at

[http://localhost:8080/ClientStore](http://localhost:8080/ClientStore/)

The exact path to the wsdl should be:

<http://localhost:8080/ClientStore/ws/clientStore?wsdl>

**Step 8:** JSON and REST interface

Get familiar with the concept using the following links

JSON - <http://www.w3schools.com/json/>

Restful Services - <http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069>

JAX-RS - <https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services>

Alright, let’s start building the REST interface. First we’ll need to add two set of libraries to our pom file – jax-rs and Jackson. Jackson libraries will take care about serialization and deserialization of JSON objects.

Here are the dependencies



Now, after we have imported the libraries we can create our API class:



We have created our API class, so now we need to add them in the CXF Rest context:

Open CXFConfig class and autowire our rest service:

**

And add the following code to the init() method

**

Deploy the app and test the rest service. It should be available at:

<http://localhost:8080/ClientStore/rs/hi>

**Step 9:** Setting up the data access layer

Assuming that MySQL server is correctly installed and already running. Go and create a database schema called ‘cstore’

So, now we need a JDBC driver in order to enable our application’s database connectivity. The driver contains classes needed for the communication between a java application and a MySQL database.

Just google ‘mysql jdbc download’, download the jar file and place it in the lib folder located in Tomcat’s directory.

Now we have to define our JDBC data source in Tomcat’s server context. Go to your eclipse press ctrl + shift + r and type ‘context.xml’ open the file and paste the following xml fragment in it:

**

Make sure that you have adjusted the properties accordingly to your local MySQL setup.

This basically defines the data source in the server’s context and makes it available to the applications deployed in that server through JNDI lookup.

**The Java Naming and Directory Interface (JNDI) is a Java API for a directory service that allows Java software clients to discover and look up data and objects via a name.**

For the data access layer we are going to use Spring Data which steps on JPA. Review the following resources to gain familiarity with the concepts:

<http://www.tutorialspoint.com/jpa/index.htm>

<http://projects.spring.io/spring-data/>

<http://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

Now after we know what Spring Data is and how does it work we are going to integrate it in our project.

First we need to add two dependencies to our pom file – spring data and hibernate. Hibernate is JPA implementation. JPA is a specification that has many implementations. Hibernate is the most widely used one.

Go ahead and add the following two dependencies:



The next thing is to configure the needed resources. Create the following class and carefully review the methods in it:



**Step 10:** Entity transformations.

In reality the objects exposed by the API usually defer from the db objects. This is due to the fact that the API objects are being designed for API perspective, while the db objects reflect the database tables’ structure.

This is why we have created a package for the API objects:

com.scalefocus.edu.api.model

And a package for the db objects

com.scalefocus.edu.db.model

The mapping between the two types of objects can be easily done using Dozer.

<http://dozer.sourceforge.net/>

Add the following maven dependency:



And the following two classes:





If we want to copy the values from API object to DB object that has the same fields the Dozer will do it without the need of configuring an explicit mapping. If we need to transform non-matching objects we have to tell dozer how to do the transformation in *ClientStoreMappingBuilder*

<http://dozer.sourceforge.net/documentation/apimappings.html>

In DozerConfig we initialized the DozerBeanMapper as a spring Bean, so whenever we want to map two entities we just need to autowire it and then we can use it



The right place in our case to transform API entities to DB ones is in the backend service layer.(ClientStoreService class)

## **Implementation**

Now after you have the project skeleton and all the frameworks fully integrated. Go ahead and implement the requirements. The steps you should take are:

* Create database schema
* Create entities corresponding to the tables you just created (place them in com.scalefocus.edu.db.model package)
* Create the Spring Data repositories
* Create implement all operations from the requirements

The service call chain should be like this:

Client 🡪 API Class 🡪 Backend Service 🡪 Spring Data Repository 🡪 Database

Have a fun.