(http://baeldung.com)

Una guía para JavaLite: creación de una aplicación RESTful CRUD

Última modificación: 20 de enero de 2018

por baeldung (http://www.baeldung.com/author/baeldung/) (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) + DESCANSO (http://www.baeldung.com/category/rest/)

1. Introducción

JavaLite (http://javalite.io/) es una colección de marcos para simplificar las tareas comunes que todo desarrollador debe enfrentar al construir aplicaciones.

En este tutorial, vamos a echar un vistazo a las características de JavaLite enfocadas en construir una API simple.

2. Configuración

Throughout this tutorial, we'll create a simple RESTful CRUD application. In order to do that, **we'll use ActiveWeb and ActiveJDBC** – two of the frameworks that JavaLite integrates with.

So, let's get started and add the first dependency that we need:

```
<dependency>
    <groupId>org.javalite</groupId>
    <artifactId>activeweb</artifactId>
    <version>1.15</version>
</dependency>
```

ActiveWeb artifact includes ActiveJDBC, so there's no need to add it separately. Please note that the latest activeweb (https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.javalite%22%20AND%20a%3A%22activeweb%22) version can be found in Maven Central.

The second dependency we need is **a database connector**. For this example, we're going to use MySQL so we need to add:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.45</version>
</dependency>
```

Again, latest mysql-connector-java (https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22mysql%22%20AND%20a%3A%22mysql-connector-java%22) dependency can be found over on Maven Central.

The last dependency that we have to add is something specific to JavaLite:

```
<plugin>
    <groupId>org.javalite</groupId>
    <artifactId>activejdbc-instrumentation</artifactId>
    <version>1.4.13</version>
    <executions>
        <execution>
            <phase>process-classes</phase>
            <goals>
                <goal>instrument</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

The latest activejdbc-instrumentation (https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.javalite%22%20AND%20a%3A%22activejdbc-instrumentation%22) plugin can also be found in Maven Central.

Having all this in place and before starting with entities, tables, and mappings, we'll make sure that **one of the supported databases (http://javalite.io/activejdbc#supported-databases) is up and running**. As we said before, we'll use MySQL.

Now we're ready to start with object-relational mapping.

3. Object-Relational Mapping

3.1. Mapping and Instrumentation

Let's get started by **creating a *Product* class that will be our main entity**:

```
public class Product {}
```

And, let's also **create the corresponding table for it**:

```
CREATE TABLE PRODUCTS (
    id int(11) DEFAULT NULL auto_increment PRIMARY KEY,
    name VARCHAR(128)
);
```

Finally, we can **modify our *Product* class to do the mapping**:

```
public class Product extends Model {}
```

We only need to extend *org.javalite.activejdbc.Model* class. **ActiveJDBC infers DB schema parameters from the database**. Thanks to this capability, **there's no need to add getters and setters or any annotation**.

Furthermore, ActiveJDBC automatically recognizes that *Product* class needs to be mapped to *PRODUCTS* table. It makes use of English inflections to convert singular form of a model to a plural form of a table. And yes, it works with exceptions as well.

There's one final thing that we will need to make our mapping work: instrumentation. **Instrumentation is an extra step required by ActiveJDBC** that will allow us to play with our *Product* class as if it had getters, setters, and DAO-like methods.

After running instrumentation, we'll be able to do things like:

```
Product p = new Product();
p.set("name","Bread");
p.saveIt();
```

or:

```
List<Product> products = Product.findAll();
```

This is where *activejdbc-instrumentation* plugin comes in. As we already have the dependency in our pom, we should see classes being instrumented during build:

```
...
[INFO] --- activejdbc-instrumentation:1.4.11:instrument (default) @ javalite ---
************************** START INSTRUMENTATION **************************
Directory: ...\tutorials\java-lite\target\classes
Instrumented class: .../tutorials/java-lite/target/classes/app/models/Product.class
************************** END INSTRUMENTATION **************************
...
```

Next, we'll create a simple test to make sure this is working.

3.2. Testing

Finally, to test our mapping, we'll follow three simple steps: open a connection to the database, save a new product and retrieve it:

```
@Test
public void givenSavedProduct_WhenFindFirst_ThenSavedProductIsReturned() {

    Base.open(
      "com.mysql.jdbc.Driver",
      "jdbc:mysql://localhost/dbname (mysql://localhost/dbname)",
      "user",
      "password");

    Product toSaveProduct = new Product();
    toSaveProduct.set("name", "Bread");
    toSaveProduct.saveIt();

    Product savedProduct = Product.findFirst("name = ?", "Bread");

    assertEquals(
      toSaveProduct.get("name"),
      savedProduct.get("name"));
}
```

Note that all this (and more) is possible by only having an empty model and instrumentation.

4. Controllers

Now that our mapping is ready, we can start thinking about our application and its CRUD methods.

For that, we're going to make use of controllers which process HTTP requests.

Let's create our *ProductsController*:

```
@RESTful
public class ProductsController extends AppController {

    public void index() {
        // ...
    }

}
```

With this implementation, ActiveWeb will automatically map *index()* method to the following URI:

```
http://<host>:<port (http://<host>:<port)>/products
```

Controllers annotated with *@RESTful*, **provide a fixed set of methods automatically mapped to different URIs.** Let's see the ones that will be useful for our CRUD example:

Controller method	HTTP method	URI	
CREATE	*create()*	POST	*http://host:port/products*
READ ONE	*show()*	GET	*http://host:port/products/[id]*
READ ALL	*index()*	GET	*http://host:port/products*
UPDATE	*update()*	PUT	*http://host:port/products/[id]*
DELETE	*destroy()*	DELETE	*http://host:port/products/[id]*

And if we add this set of methods to our *ProductsController*:

```
@RESTful
public class ProductsController extends AppController {

    public void index() {
        // code to get all products
    }

    public void create() {
        // code to create a new product
    }

    public void update() {
        // code to update an existing product
    }

    public void show() {
        // code to find one product
    }

    public void destroy() {
        // code to remove an existing product
    }
}
```

Before moving on to our logic implementation, we'll take a quick look at few things that we need to configure.

5. Configuration

ActiveWeb is based mostly on conventions, project structure is an example of that. **ActiveWeb projects need to follow a predefined package layout**:

```
src
 |----main
       |----java.app
       |     |----config
       |     |----controllers
       |     |----models
       |----resources
       |----webapp
             |----WEB-INF
             |----views
```

There's one specific package that we need to take a look at – a*pp.config*.

Inside that package we're going to create three classes:

```
public class DbConfig extends AbstractDBConfig {
    @Override
    public void init(AppContext appContext) {
        this.configFile("/database.properties");
    }
}
```

This class configures database connections using a properties file in the project's root directory containing the required parameters:

```
development.driver=com.mysql.jdbc.Driver
development.username=user
development.password=password
development.url=jdbc:mysql://localhost/dbname (mysql://localhost/dbname)
```

This will create the connection automatically replacing what we did in the first line of our mapping test.

The second class that we need to include inside *app.config* package is:

```
public class AppControllerConfig extends AbstractControllerConfig {

    @Override
    public void init(AppContext appContext) {
        add(new DBConnectionFilter()).to(ProductsController.class);
    }
}
```

This code will bind the connection that we just configured to our controller.

The third class **will configure our app's context**:

```
public class AppBootstrap extends Bootstrap {
    public void init(AppContext context) {}
}
```

After creating the three classes, the last thing regarding configuration is **creating our *web.xml* file** under *webapp/WEB-INF* directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns=...>

    <filter>
        <filter-name>dispatcher</filter-name>
        <filter-class>org.javalite.activeweb.RequestDispatcher</filter-class>
        <init-param>
            <param-name>exclusions</param-name>
            <param-value>css,images,js,ico</param-value>
        </init-param>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>dispatcher</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

Now that configuration is done, we can go ahead and add our logic.

6. Implementing CRUD Logic

With the DAO-like capabilities provided by our *Product* class, it's super simple to **add basic CRUD functionality**:

```
@RESTful
public class ProductsController extends AppController {

    private ObjectMapper mapper = new ObjectMapper();

    public void index() {
        List<Product> products = Product.findAll();
        // ...
    }

    public void create() {
        Map payload = mapper.readValue(getRequestString(), Map.class);
        Product p = new Product();
        p.fromMap(payload);
        p.saveIt();
        // ...
    }

    public void update() {
        Map payload = mapper.readValue(getRequestString(), Map.class);
        String id = getId();
        Product p = Product.findById(id);
        p.fromMap(payload);
        p.saveIt();
        // ...
    }

    public void show() {
        String id = getId();
        Product p = Product.findById(id);
        // ...
    }

    public void destroy() {
        String id = getId();
        Product p = Product.findById(id);
        p.delete();
        // ...
    }
}
```

Easy, right? However, this isn't returning anything yet. In order to do that, we have to create some views.

7. Views

ActiveWeb uses FreeMarker (http://freemarker.org/) as a templating engine, and all its templates should be located under *src/main/webapp/WEB-INF/views*.

Inside that directory, we will place our views in a folder called *products* (same as our controller). Let's create our first template called *_product.ftl*:

```
{
    "id" : ${product.id},
    "name" : "${product.name}"
}
```

It's pretty clear at this point that this is a JSON response. Of course, this will only work for one product, so let's go ahead and create another template called *index.ftl*:

```
[<@render partial="product" collection=products/>]
```

Esto básicamente generará una colección llamada *productos* , con cada uno formateado por *_product.ftl* .

Finalmente, **debemos vincular el resultado de nuestro controlador a la vista correspondiente** :

```
@RESTful
public class ProductsController extends AppController {

    public void index() {
        List<Product> products = Product.findAll();
        view("products", products);
        render();
    }

    public void show() {
        String id = getId();
        Product p = Product.findById(id);
        view("product", p);
        render("_product");
    }
}
```

En el primer caso, estamos asignando una lista de *productos* a nuestra colección de plantillas llamada también *productos* .

Entonces, como no estamos especificando ninguna vista, se *usará index.ftl* .

En el segundo método, asignamos el producto *p* al *producto del* elemento en la vista y estamos diciendo explícitamente qué vista mostrar.

También podríamos crear un view *message.ftl* :

```
{
    "message" : "${message}",
    "code" : ${code}
}
```

Y luego llámelo desde cualquiera de los métodos de nuestro *productoController* :

```
view("message", "There was an error.", "code", 200);
render("message");
```

Veamos ahora nuestro *productoProgramador* final :

```
@RESTful
public class ProductsController extends AppController {

    private ObjectMapper mapper = new ObjectMapper();

    public void index() {
        view("products", Product.findAll());
        render().contentType("application/json");
    }

    public void create() {
        Map payload = mapper.readValue(getRequestString(), Map.class);
        Product p = new Product();
        p.fromMap(payload);
        p.saveIt();
        view("message", "Successfully saved product id " + p.get("id"), "code", 200);
        render("message");
    }

    public void update() {
        Map payload = mapper.readValue(getRequestString(), Map.class);
        String id = getId();
        Product p = Product.findById(id);
        if (p == null) {
            view("message", "Product id " + id + " not found.", "code", 200);
            render("message");
            return;
        }
        p.fromMap(payload);
        p.saveIt();
        view("message", "Successfully updated product id " + id, "code", 200);
        render("message");
    }

    public void show() {
        String id = getId();
        Product p = Product.findById(id);
        if (p == null) {
            view("message", "Product id " + id + " not found.", "code", 200);
            render("message");
            return;
        }
        view("product", p);
        render("_product");
    }

    public void destroy() {
        String id = getId();
        Product p = Product.findById(id);
        if (p == null) {
            view("message", "Product id " + id + " not found.", "code", 200);
            render("message");
            return;
        }
        p.delete();
        view("message", "Successfully deleted product id " + id, "code", 200);
        render("message");
    }

    @Override
    protected String getContentType() {
        return "application/json";
    }

    @Override
    protected String getLayout() {
        return null;
    }
}
```

En este punto, nuestra aplicación está lista y estamos listos para ejecutarla.

8. Ejecución de la aplicación

Usaremos el plugin Jetty:

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.4.8.v20171121</version>
</plugin>
```

Encuentra el último jetty-maven-plugin (https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.eclipse.jetty%22%20AND%20a%3A%22jetty-maven-plugin%22) en Maven Central.

Y estamos listos, **podemos ejecutar nuestra aplicación** :

```
mvn jetty:run
```

Vamos a crear un par de productos:

```
$ curl -X POST http://localhost:8080/products
  -H 'content-type: application/json'
  -d '{"name":"Water"}'
{
    "message" : "Successfully saved product id 1",
    "code" : 200
}
```

```
$ curl -X POST http://localhost:8080/products
  -H 'content-type: application/json'
  -d '{"name":"Bread"}'
{
    "message" : "Successfully saved product id 2",
    "code" : 200
}
```

.. léelos:

```
$ curl -X GET http://localhost:8080/products
[
    {
        "id" : 1,
        "name" : "Water"
    },
    {
        "id" : 2,
        "name" : "Bread"
    }
]
```

.. actualizar uno de ellos:

```
$ curl -X PUT http://localhost:8080/products/1
  -H 'content-type: application/json'
  -d '{"name":"Juice"}'
{
    "message" : "Successfully updated product id 1",
    "code" : 200
}
```

... lee el que acabamos de actualizar:

```
$ curl -X GET http://localhost:8080/products/1
{
    "id" : 1,
    "name" : "Juice"
}
```

Finalmente, podemos eliminar uno:

```
$ curl -X DELETE http://localhost:8080/products/2
{
    "message" : "Successfully deleted product id 2",
    "code" : 200
}
```

9. Conclusión

JavaLite tiene muchas herramientas para ayudar a los desarrolladores a **poner en marcha una aplicación en minutos** . Sin embargo, aunque basar las cosas en las convenciones resulta en un código más simple y más limpio, lleva un tiempo entender la denominación y ubicación de clases, paquetes y archivos.

Esta fue solo una introducción a ActiveWeb y ActiveJDBC, encuentra más documentación en su sitio web (http://javalite.io) y busca nuestra aplicación de productos en el proyecto Github (https://github.com/eugenp/tutorials/tree/master/java-lite) .

Descargar el libro electrónico

¿Construir una API REST con Spring 4?

Email Address

Descargar

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORES (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)