Getting Started with Dropwizard: First Steps

(/users/1370705/javaeeeee.html) by Dmitry Noranovich (/users/1370705/javaeeeee.html) · Jan. 30, 15 · Web Dev Zone (/web-development-programming-tutorials-tools-news)

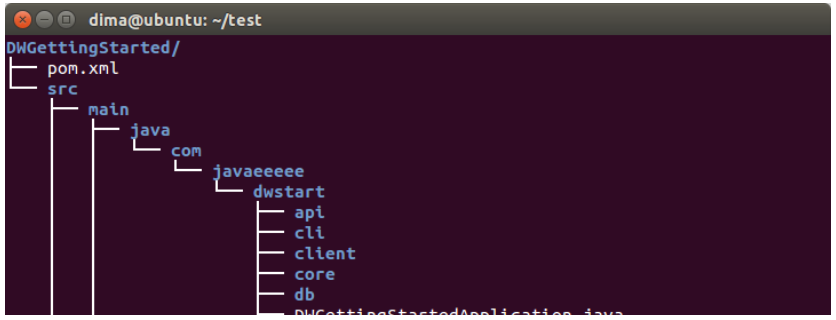
Easily build powerful user management, authentication, and authorization into your web and mobile applications. Download this Forrester report (/go?i=124040&u=http%3A%2F%2Fgo.stormpath.com%2Fforrester-customer-identity-access-management-stormpath%3Futm_source%3Ddzone%26utm_medium%3Dpre-text%26utm_content%3Dforrester%26utm_campaign%3Dweb-dev-2016) on the new landscape of Customer Identity and Access Management, brought to you in partnership with Stormpath (/go?i=124040&u=http%3A%2F%2Fgo.stormpath.com%2Fforrester-customer-identity-access-management-stormpath%3Futm_source%3Ddzone%26utm_medium%3Dpre-text%26utm_content%3Dforrester%26utm_campaign%3Dweb-dev-2016).

Dropwizard (http://dropwizard.io/) is a bunch of superb frameworks glued together to provide a fast way of building web applications including REST APIs. We'll talk about great frameworks which are parts of Dropwizard as we go over our examples. The full code for the examples can be obtained here (https://bitbucket.org/dnoranovich/dropwizard-getting-started) .

The simplest way to create a Dropwizard project is to use the Maven archetype called java-simple which is a part of Dropwizard. This can be accomplished either via the command line or using your favorite IDE. Let's stick to the former approach. The command below is formatted for clarity but it should be pasted as a single line to the terminal.

```
mvn archetype:generate
  -DgroupId=com.javaeeeee
  -DartifactId=DWGettingStarted
  -Dname=DWGettingStarted
  -Dpackage=com.javaeeeee.dwstart
  -DarchetypeGroupId=io.dropwizard.archetypes
  -DarchetypeArtifactId=java-simple
  -DinteractiveMode=false
```

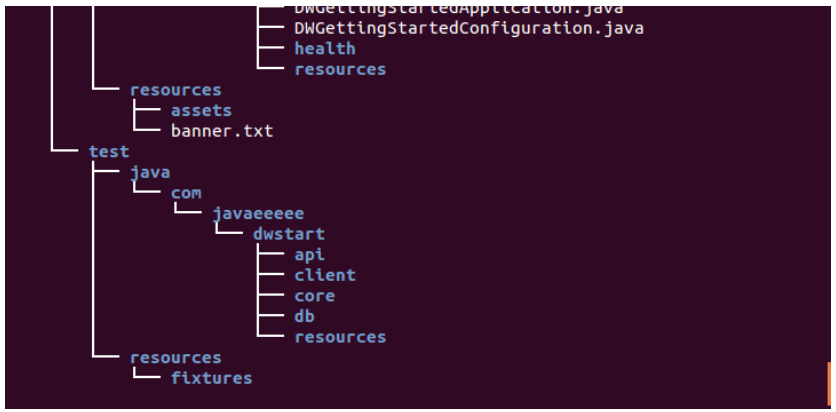
After the project was created it can be modified using an editor or opened in an IDE. All the three IntelliJ, Eclipse and Netbeans have Maven support. An important parameter to note in the pom.xml file of the newly created project is dropwizard.version, which at the time of writing was automatically set to 0.8.0-rc2-SNAPSHOT and the latest stable version was 0.7.1. We'll discuss later the important consequences of the version for project development. The directory structure of the project is shown below.



There are multiple sub-packages and two classes that were created for us in a package `com.javaeeeee.dwstart`. Now let's create a simple resource class that produces a string "Hello world!", start the application and test the results using both browser and cURL. The snippet below shows the resource class.

```
@Path("/hello")
public class HelloResource {

    @GET
```




```
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getGreeting() {
        return "Hello world!";
    }
}
```

There is a special package, `com.javaeeeee.dwstart.recources` for placing resource classes or an appropriate folder can be found from the screenshot above if you use a text editor and not an IDE.

One of the main tenets of the REST architecture style is that each resource is assigned a URL. In our case the URL of our resource would be http://localhost:8080/hello, that is we access it from our local machine, the default port which is used by Dropwizard is 8080 and the final part of the URL is enshrined in @Path annotation in our class, in other words, the aforementioned annotation helps to describe the URL used to access a resource.

Moving on to method getGreeting() it can be seen that it is a simple method which returns the desired string “Hello world!”, although it is marked with two annotations: first, @GET prescribes that this method is called when the aforementioned URL is accessed using HTTP GET method, second, annotation @Produces specifies what media types can be produced when the method is accessed by the client, such as browser, cURL or some other program. In our case it is plain text. These annotations are part of Java API for RESTful Web Services (http://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services) (JAX-RS), and its reference implementation Jersey (https://jersey.java.net) is the cornerstone of Dropwizard.

Other annotations include @POST for HTTP POST method, @DELETE for DELETE and so on and media types include MediaType.APPLICATION_JSON and not so popular MediaType.APPLICATION_XML among others. Now, to see the result of our coding, we should do some configuration. Let's open one of the created by Maven files, DWGettingStartedApplication.java, and register our resource.

```
    public void run(final DWGettingStartedConfiguration configuration,
            final Environment environment) {
        environment.jersey().register(new HelloResource());
    }
```

We added a single line environment.jersey().register(new HelloResource()); to help Dropwizard find our resource class. Now we have to create a jar-file which contains an embedded Jersey web server to serve the incoming requests, as well as all the necessary libraries. Dropwizard applications are packaged as executable jar-files rather than war-files which are deployed to an application server. The kind of jar files used by Dropwizard applications are also called “fat” because they include all the .class files necessary to run the application as this leads to a situation that the versions of all libraries are the same in both development and production environments which leads to less problems. One shouldn't worry about creating such jar-files as Maven creates them for us using instructions in the generated pom.xml file of the project. The jar-file can be created using command-line command issued from the project's folder

```
mvn package
```

or using an IDE. After that we should start the application, which could be accomplished either from the command line by issuing a command

```
java -jar target/DWGettingStarted-1.0-SNAPSHOT.jar server
```

or from the IDE which can be instructed to pass a “server” argument to the executable jar file. To stop the application it is sufficient to press Ctrl+C in the terminal.

Now it is time to check if it works. The simplest way to achieve this is to navigate your browser to the URL mentioned earlier. You should see the greeting string in the main window of a browser. If you cannot see the greeting, there is probably some problems and the program’s

the greeting string in the main window of a browser. If you cannot see the greeting, there is probably some problems and the program's output is the place to look for clues for what went wrong.

The same greeting can be seen in the terminal window with help of cURL.

```
curl -w "\n" localhost:8080/hello
```

Protocol HTTP and GET method are defaults, so it is not necessary that they be included explicitly in the command. As the result a message "Hello world!" without quotes should be printed. The w-option is used to add a trailing newline to prettify the output.

It is a good idea to start using tests as early as possible, so let's write a test that checks the work of our resource class using in-memory Jersey. We'll create the test in test packages and as we are testing a resource an appropriate package to place our test is `com.javaeeeee.dwstart.resources` (or src/test/com/javaeeeee/dwstart/resources folder).

It is necessary that the following dependency be added to the pom-file of the project. (It works without explicit jUnit dependency.)

```
        <dependency>
            <groupId>io.dropwizard</groupId>
            <artifactId>dropwizard-testing</artifactId>
            <version>${dropwizard.version}</version>
            <scope>test</scope>
        </dependency>
```

An example test for our greeting resource is shown below.

```
public class HelloResourceTest {

    @Rule
    public ResourceTestRule resource = ResourceTestRule.builder()
            .addResource(new HelloResource()).build();

    @Test
    public void testGetGreeting() {
        String expected = "Hello world!";
        //Obtain client from @Rule.
        Client client = resource.client();
        //Get WebTarget from client using URI of root resource.
        WebTarget helloTarget = client.target("http://localhost:8080/hello");
        //To invoke response we use Invocation.Builder
        //and specify the media type of representation asked from resource.
        Invocation.Builder builder = helloTarget.request(MediaType.TEXT_PLAIN);
        //Obtain response.
        Response response = builder.get();

        //Do assertions.
        assertEquals(Response.Status.OK, response.getStatusInfo());
        String actual = response.readEntity(String.class);
        assertEquals(expected, actual);

    }
}
```

The annotation @Rule is from jUnit realm and could be placed on public non-static fields and methods that return a value. This annotation may perform some initial setup and clean-up like @Before and @After methods, but it is more powerful as it allows to share functionality between classes and even projects. This rule is needed to allow us to access our resources from the code of the test using in-memory Jersey.

At this point of time we should pause and discuss the version of our product. There is a pom.xml file in the dropwizard folder on the GitHub (https://github.com/dropwizard/dropwizard), which can readily be accessed by cloning the project, and this file specifies the version of constituent products among other things. It can be seen from it that the version of Jersey in the snapshot version of Dropwizard is 2.13 and in the version 0.7.1 it is 1.18.1. The version of Jersey influences the code of the tests as 1.x and 2.x versions of Jersey have different client APIs, that is the code to access the service programmatically from Java depends on the version of Jersey framework. Let's stick to the latest 2.x version of Jersey as it is liable to be included in following 0.8.x releases of Dropwizard and if you are interested how to test using the older

version of Jersey client, please check the project repository (https://bitbucket.org/dnoranovich/dropwizard-getting-started) , there is a special branch v0.7.x. Another observation concerning testing is that Fest matchers were superseded by AssertJ counterparts.

First, we obtain an instance of Jersey client to make requests to our service. Second, we create an instance of a class that implements WebTarget interface for the resource using its URL. After that we use Invocation.Builder that helps to build request and send it to the server. The Invocation.Builder interface extends SynchInvoker interface which defines a bunch of get(...) methods used to invoke HTTP GET method synchronously. Finally, we check the results, namely that the HTTP status code was 200 and the desired string was returned.

The same result could be obtained by chaining the methods as in the snippet below.

```
    actual = resource.client()
             .target("http://localhost:8080/hello")
             .request(MediaType.TEXT_PLAIN)
             .get(String.class);
     assertEquals(expected, actual);
```

The replacement of the two assertions by a single is possible due to the fact that the get(...) method we used throws a WebApplicationException if the HTTP status code is not in the 2xx range, in other words if there is a problem.

Now let's improve our resource to take a name and return a bespoke greeting. It could be done in a couple of ways using Jersey's annotations. The first is to use so-called path parameters, that is you pass the name using a URL like http://localhost:8080/hello/path_param/your_name. The application should return “Hello your_name”. Let's see a code snippet.

```
@Path("/path_param/{name}")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getNamedGreeting(@PathParam(value = "name") String name) {
    return "Hello " + name;
}
```

Now we see a @Path annotation on our method. It adds its content to those produced by class level annotation as seen from the URL. Methods marked with this annotation are called sub-resource methods. The name of the parameter, name, is in curly braces. Something interesting happens inside the parenthesis of the getNamedGreeting method. There is an annotation which prescribes to extract the content of the url after “/path_param/” to the method as a value. It should be noted that if you find yourself marking each method with the same annotation @Produces(MediaType.TEXT_PLAIN), the latter can be removed from methods and used to mark the class, then all the methods produce the desired media types, although this can be overruled by an annotations with a different media type for some of the methods.

The second way to pass a name is to use query parameters and the URL could look like http://localhost:8080/hello/query_param?name=your_name. The output should be “Hello your_name” without quotes. The snippet is shown below.

```
@Path("/query_param")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getNamedStringWithParam(@DefaultValue("world") @QueryParam("name") String name) {
    return "Hello " + name;
}
```

Once again, the @Path annotation adds something to our URL, then the default value of the parameter is set in case there is no question mark and following it symbols in the URL using @DefaultValue annotation. In other words, if the parameter is omitted in the URL and it looks like http://localhost:8080/hello/query_param, a phrase “Hello world” is printed. The last annotation injects a parameter from the URL to the method's parameter. By the way, all these cases can be tested as was shown above. To test a method using query parameters one can use a queryParam(...) method of WebTarget interface.

There is a couple of points to pay attention to. Firstly, there are other @*Param annotations available and secondly, the @DefaultValue annotation will not work with path parameters and to process the absence of the parameter a special sub-resource method should be introduced.

You were patient enough to bear all this plain text stuff, but it is not what REST APIs are about. Let's create another sub-resource method which is preprogrammed to return JSON representation. The first step is to create a representation class, which should be placed to com.javaeeeee.dwstart.core package. Let's limit ourselves to extremely simple example which is shown below.

```
public class Greeting {

    @JsonProperty
    private String greeting;

    public Greeting() {
    }

    public Greeting(String greeting) {
        this.greeting = greeting;
    }

    public String getGreeting() {
        return greeting;
    }
}
```

The class above relies on one more important part of the Dropwizard stack namely Jackson (http://wiki.fasterxml.com/JacksonHome) , which can turn Java objects to JSON and vice versa. A @JsonProperty annotation is used to do the job. The sub-resource method looks like pretty much the same as our previously discussed methods.

```
    @Path("/hello_json")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Greeting getJSONGreeting() {
        return new Greeting("Hello world!");
    }
```

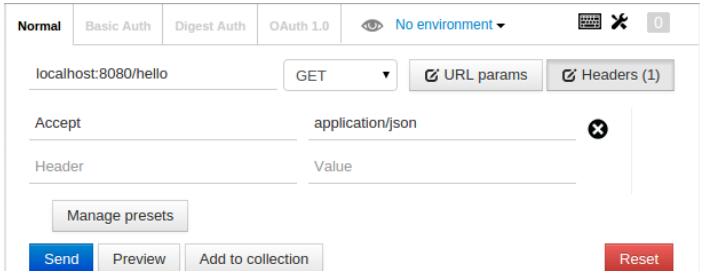
The difference is that we changed the media type and the method returns an instance of the greeting class. If you navigate your browser to to a URL http://localhost:8080/hello/hello_json or use cURL, you should see a JSON representation of the object {"greeting":"Hello world!"}.

What if we remove the @Path annotation from the aforementioned method altogether? Well, if you try to access the localhost:8080/hello resource you'll see the text representation. How one could access the JSON-producing method? There is a special HTTP header called Accept which is used to instruct the server what representation to return, the process called content negotiation. The Jersey inside the Dropwizard will choose for us what method to use based on the content of the Accept header. Let's try cURL to do this.

```
curl -w "\n" -H 'Accept: application/json' localhost:8080/hello/
```

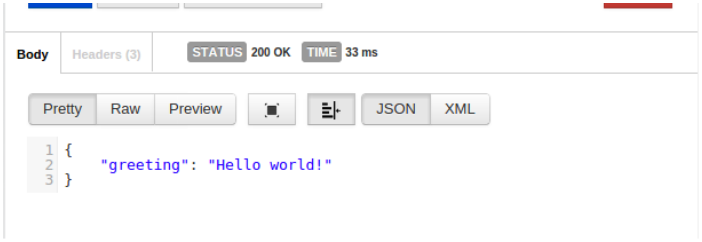
The H-option is used to send headers and we instructed cURL to ask the JSON response. Other possible types to try could include text/plain and application/xml. The former should return the plain-text representation and the letter should result in error as there is no method in our resource class that can produce the latter media type response.

Another way to engage in content negotiation is to use a REST client for your browser. There are several clients for each of major browsers but we will use Postman extension for Chrome browser. There are special fields to input headers as the screenshot below shows.



One can enter the resource URL, headers and press Send button and the response will show up in the lower part of the window.

To sum up we created a simple REST-like API using Dropwizard, which can greet its users. We learned how to create a Dropwizard project using Maven archetype and created a simple resource class to accomplish the task of greeting and a representation class to produce JSON response. An important ingredient of REST



produce JSON response. An important ingredient of REST architecture style, HATEOAS, was omitted for now for simplicity reasons. Also a problem of testing resource classes was touched as well as the ways to access resources both from browser and command line. There are some other ideas for creating sub-resources such as returning the current date and adding two numbers using query parameters which can be easily implemented using the information presented in this article.

References

1. Dropwizard on GitHub (https://github.com/dropwizard/dropwizard)
2. Maven Archetypes (http://maven.apache.org/guides/introduction/introduction-to-archetypes.html)
3. Dropwizard Getting Started (http://dropwizard.io/getting-started.html)
4. JAX-RS Resources and Subresources (https://jersey.java.net/documentation/latest/user-guide.html#jaxrs-resources)
5. Dropwizard Testing (http://dropwizard.io/manual/testing.html)
6. Jersey Client API 2.x (https://jersey.java.net/documentation/latest/client.html)
7. Jersey Client API 1.x (https://jersey.java.net/documentation/1.18/client-api.html)
8. Jackson annotations (https://github.com/FasterXML/jackson-annotations)

Topics: JAVA, TESTING, API, TUTORIAL, JSON, REST, JETTY, RESTFUL, JAX-RS, JACKSON

From ng-controller to Components With Angular 1.5+

For those who still haven't migrated their Angular 1 apps past Angular 1.5, it's time to play catch-up. Review the syntax and functionality changes and improvements to upgrade your app towards Angular 2.

(/users/999973/juristr.html) by Juri Strumpflohner (/users/999973/juristr.html) MVB · Jul 14, 16 · Web Dev Zone (/web-development-programming-tutorials-tools-news)

The web has moved fast and so should you. Learn how to upgrade your Angular 1 app from a more MV* architecture to a cleaner, more component oriented approach. We will learn about how to refactor your code properly and about the new features introduced in Angular 1.5+ that will help you succeed along this path.

The App

I've created a series of Plunks which you can use to play around with the code by yourself. So here's our initial app, a very simple one, just enough to showcase some concepts we're going to explore.

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8" />
    <title>AngularJS Plunker</title>
```