Spring Boot RESTful Web Service Example

The main purpose of this sample project is to demonstrate the capabilities of spring boot. But additionally, I want to show challenging problems that can occur during the development while using the Spring Boot.

First goal is to show how it is easy to start a web service with embedded tomcat and embedded H2 database. This is the main goal of the project.

Secondly, we are using Spring and I have used dependency injection. But what is a challenging problem about dependency injection. Assume that you have two implementations ready for one implementation, how are you going to select the implementation? I'll explain several ways but also I'll demonstrate how we can select our implementation via external configuration so that we can update our configuration and don't need to touch the code, restart our jar file and that's all.

Thirdly, I also have demonstrated how to use Java Application Configuration within the double implementation for the single interface scenario I've explained above.

Lastly, I will explain all the deployment details, the main configuration of the whole project including H2 database configuration.

Moreover, I will also demonstrate how you are going to test your RESTful application with Postman tool.

TOC

0 Prerequisite And Demo App

To use this project, you are going to need;

- Java JDK 8 (1.8)
- Maven compatibile with JDK 8
- Any Java IDE
- Postman tool (optional, will be used for testing web service)

We are going to build a demo app named as consultant-api. This will be a simple web service with basic CRUD operations. I'm going to demonstrate default and external configuration, how to use multiple implementation and autowire them within the code and outside the code with an external configuration file. Our app will be a standalone application that we can use independently, and we are going to use an embedded tomcat, an embedded H2 database.

Go back to TOC

1 About Spring Boot

Whenever there is a new framework on the town, you must think two thinks. One, why should I use this framework which means "what are the benefits of this framework", also can be interpreted like "what this framework solves?". Two, "When should I use this framework?", also can be interpreted as "on which specific scenarios this framework is useful" or can be simplified as "what is the problem domain of this framework?".

When we make a web service with spring framework, we have to generate a war file, we need to configure web.xml, and also if we are going to use the connection pool, the configuration is costly. All of these increases the cost of time. So instead of writing your code, doing your development, you a lot of time is wasted during the configuration. This is where Spring Boot comes to the action. Spring Boot simplifies configuration, reduces boilerplate code that puts no any value to your software development.

So, what Spring Boot solves is the time lost for the configuration. For example, you can create a web service with Spring Boot that runs on an embedded Tomcat server which is automatically configured and you don't have to deal with the configuration. You can do all your configuration parameters via default application properties. Also you can connect to an H2 embedded database, same applies for the configuration here.

Secondly, you don't have to generate a war file. All Spring Boot applications run as a standalone java jar file. Where is it useful then? If you are using a microservice architecture which runs especially on a cloud (but not necessarily), then you can easily do your development via Spring Boot. In my opinion, Spring Boot is one of the best frameworks you should use on such a scenario and architecture. You can easily create simple web services, put them inside a Docker container (which is not a part of this tutorial) and run them on the Amazon Web Services or on any cloud environment.

Additionally, you don't have to track the versioning of your dependencies. Normally, if you are using a version of spring, then you have to use the appropriate versions of your other dependencies which are dependent to the main dependency. With Spring Boot, you don't have to check out what version you have to use for Jackson which is compatible with the version of Jersey. You don't even need to write the version of your dependencies. I'm going to demonstrate all of these benefits.

Go back to TOC

2 Create Spring Boot Project With Maven

What we need to setup a Spring Boot project. However there are other ways (like spring initializer), I'll go with setting up our project with maven.

Because that we are creating a web application here, we will first create a maven project with web application archetype, then we will add the spring boot dependencies;

You can use the following maven command to create a project. In this project, I've used this exact maven command to create our project;

```
mvn archetype:generate -DgroupId=com.levent.consultantapi -DartifactId=consultant-api  -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Go back to TOC

3 Spring Boot Dependencies

How we make our project a Spring Boot project. We simply define a parent project in our POM file just as below;

```
<!-- Spring Parent Project -->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
</parent>
```

Then our dependencies aware of our Spring Boot project. Then for example, if we are going to create a web application which is true, then we add the Spring Boot Starter dependencies. On this project, it is vital for us to add the dependency below;

```
<!-- Spring boot starter web: integrates and auto-configures  -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

See that we did not use the version attribute of the dependency block. That's what Spring Boot is going to handle but just adding this starter dependency, we will have all what we need like Jersey, Jackson, and the rest. Also the versioning is managed by Spring Boot, we don't have to check if the versions of our transitive dependencies are compatible with each other or not.

Moreover, in this project we will need an H2 embedded database. So we are going to add the following dependency to our dependencies block;

```
<!-- H2 Embedded Database -->
<dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
</dependency>
```

The last thing additional to our dependencies in the POM file is to use the @SpringBootApplication annotation on our EntryPoint class.

With this, Spring Boot configuration is complete.

Go back to TOC

4 Making Uber Jar

When we are developing a web service, let's say, using Jersey framework within the Spring context, we make a .war file and upload it to a container. However, Spring Boot is a containerless framework. So we do not need any web container, which means also we won't need to generate a .war file. What we have to do is pack all the libraries and frameworks we are using in our project into a big jar file, the Uber Jar (a.k.a. Fat Jar).

To do so, our build tool maven has a plugin named as Maven Shade Plugin. We are going to define it within the build block of our POM file. You can see the sample build block as below;

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <transformers>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                <resource>META-INF/spring.handlers</resource>
              </transformer>
              <transformer
implementation="org.springframework.boot.maven.PropertiesMergingResourceTransforme
r">
                <resource>META-INF/spring.factories</resource>
              </transformer>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                <resource>META-INF/spring.schemas</resource>
              </transformer>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransforme
r" />
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransforme
r">
                <mainClass>com.levent.consultantapi.EntryPoint</mainClass>
              </transformer>
            </transformers>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

As you can see, there are two plugins used in the plugins block of the build block above. I've used maven compiler plugin so that I can define the source and destination version. The other plugin is the Maven Shade Plugin, which we use in order to pack our Uber Jar.

In the Maven Shade Plugin block, we define our mainClass. Because our application is a standalone Java application, we have to define the Entry Point, the starter class of our appliacation. The name of this class is arbitrary.

You can check out the full POM file: Project Object Model

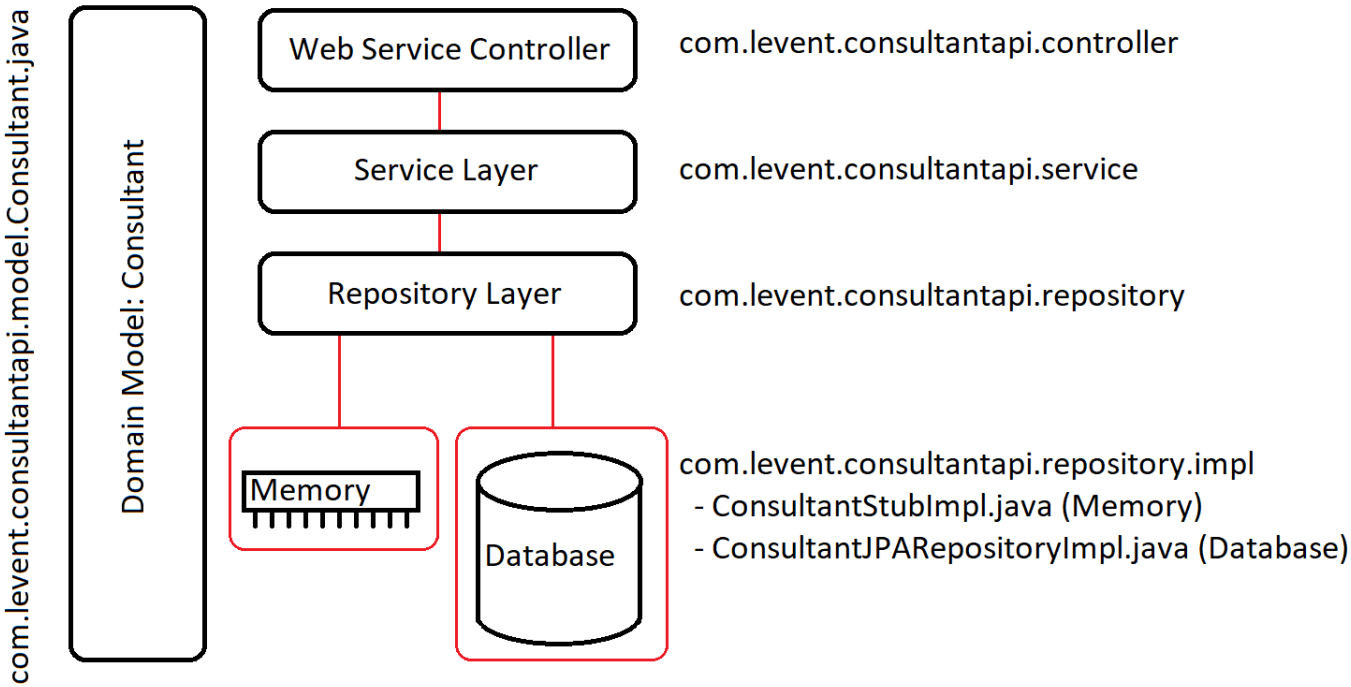
Go back to TOC

5 Project Overview

Our project consist of several layers. For the simplicity, if we exclude the EntryPoint class which is located at the top level package, we have the following packages;

- controller package
- service package
- repository package
- model package

You can see the logical representation below of these packages;



Controller package has one controller class which is ConsultantController.
With ConsultantController, you can define all the RESTful methods. This class need to use a Service Layer implementation, thus ConsultantController has a ConsultantServiceinterface and we are using the @Autowired annotation so that Spring context is going to find the appropriate implementation. In our case, we have only one Service implementation which is ConsultantServiceImpl.

You can also see the InfoService interface wrapped inside the ConsultantController class. It has also marked with the @Autowired annotation. I'll explain it later for the simplicity but our first focus in this project overview is to explain the main structure of this simple application.

At service layer, we have the interface ConsultantService and one implementation that fits with this interface: ConsultantServiceImpl. You will see that ConsultantServiceImpl has a ConsultantRepository interface and that interface also marked with the @Autowired annotation.

At repository layer, we have a different situation. There are two implementation fits with this ConsultantRepository interface;

- ConsultantStubImpl
- ConsultantJPARepositoryImpl

So, how Spring handles if there are two implementation classes those implements one interface with @Autowired annotation? How spring is going to select the implementation candidates? Normally, Spring is going to get confused, throw Exceptions and you will find the following message inside the stack trace;

```
Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No
qualifying bean of type [com.levent.consultantapi.repository.ConsultantRepository]
is defined: expected single matching bean but found 2:
consultantJPARepositoryImpl,consultantStubImpl
```

Because that there are two candidate implementation for one single interface, autowiring functionality of Spring Context will fail. The solution is to use @Primary annotation. You can see it inside the ConsultantJPARepositoryImpl. When there are multiple implementation for the same interface marked with @Autowired annotation, you have to use @Primary on one of the implementations.

But what if we want to have two different implementations and we want to change which implementation to use, without changing the code ? Then we can use an external configuration, which I'm going to explain it on next chapter.

Go back to TOC

6 External Configuration Example

Normally, Spring Boot configuration is defined with application.properties. It is default location can be either under src/main/resources folder, or under a subfolder of current directoy named with "config". I'm using the second way, created a config folder under the project, and put the main application.properties file there.

However, I also want to have the common properties on application.properties file, and in addition to it, for specific purposes, I want to use a secondary properties folder. In this chapter, I'm going to demonstrate how to use externalized custom properties file.

But first, let's go back to our previous problem that which I've talked about. The scenario is as follows: I've two or multiple implementations for a single interface with @Autowired annotation, and I don't want to do any code change when I switch between the implementations. Spring's solution for that was using the @Primary annotation so that Spring Context is not going to throw an exception because that it does not know which implementation to use.

Here is my solution;

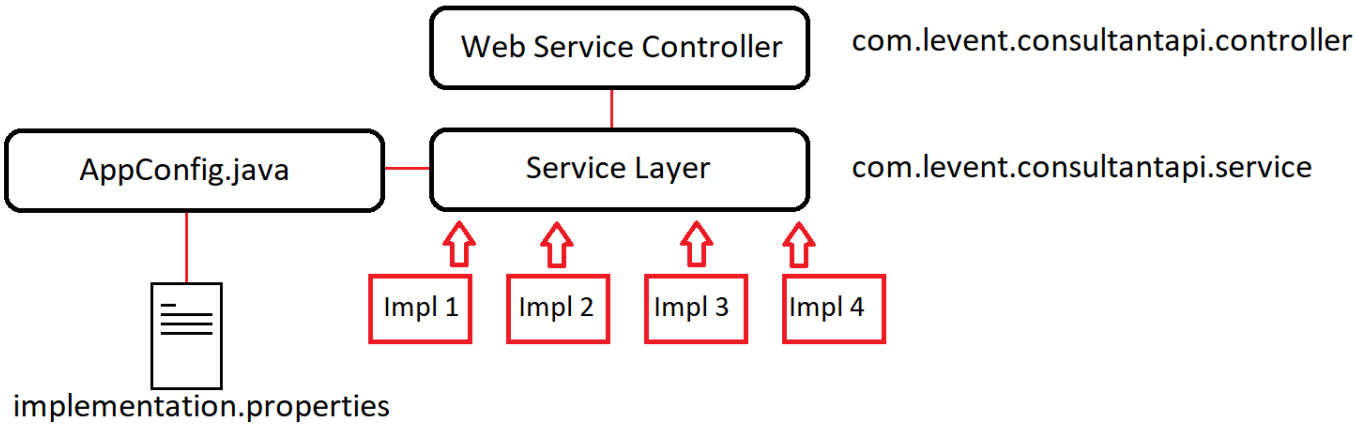
- I create an externalized configuration file named as implementation.properties
- I've created AppConfig class to read the custom configuration file

If you look at to AppConfig class under the config package, you will see that the class file is marked with two annotations. First it is marked with @Configuration annotation because it needs to be done before the injection of the autowired variable. And secondly, it is marked with @PropertySource, so that we are going to point out which specific property file we are going to use.

The process is as follows;

- Spring context searchs for the @Configuration annotation, finds the AppConfig
- Via AppConfig, reads the implementation.properties file
- The greeter.implementation property's value is loaded to the impl variable in the AppConfig
- Via the #getImplementationFromPropertiesFile method, the implementation bean is created based on the config file
- The context has the right implementation based on the configuration
- During the creation of the ConsultantController class, Spring Context finds the autowired field: greeter which is an interface: InfoService
- The implementation bean is autowired to this greeter field

You can see the overview of this process via the diagram below;



Go back to TOC

7 Application Properties

Spring Boot solves our problem with automatic configuration as we use an embedded Tomcat and an embedded H2 database but how are we going to specify the running port of the Tomcat container, the target database, connection pool parameters and so on?

Spring Boot provides a default configuration properties file called as application.properties. Within this file there are hundreds of configuration parameters we can use. You can see the detailed parameter list via following link;

Spring Boot Application Properties Reference

The default locations of the application.properties file is either somewhere within the classpath, for example under src/main/resources in a maven project, or a inside config folder under current working directory. It is better to put the file under config folder which will make it easy to deploy inside a docker container, but the choice is yours. I place application.properties under config folder.

Because that we are using a server, H2 database, a datasource, a db connection pool and lastly, hibernate, we should define parameters in this application.properties file, based on the documented reference list.

Let's take a detailed look;

- Server Configuration The default configuration port is 8080, however we may want to change this. Thus I add the port configuration as below;

```
#server port
server.port=8080
```

- H2 Database Configuration We also need to specify whether the console is activated, so that we can use H2 database via the console, create our tables and initialize our db entries.

```
#H2 configuration
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

- DataSource Configuration Instead of writing a connection string, we are defining the parameters via our properties file as below; Notify that we defined our database as a file and the name of the database is "consultantapi". We are going to use it when we need to connect the database via the console;

```
#Data source configuration
spring.datasource.url=jdbc:h2:file:~/consultantapi
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
```

- Connection Pool Configuration Here we define the connection pool parameters;

```
#DB Pool conf
spring.datasource.max-active=10
spring.datasource.max-idle=8
spring.datasource.max-wait=10000
spring.datasource.min-evictable-idle-time-millis=1000
spring.datasource.min-idle=8
spring.datasource.time-between-eviction-runs-millis=1
```

- Hibernate Configuration We don't want Hibernate to delete our database entries on every restart of our server, so we need to configure as below;

```
#Hibernate Config
spring.jpa.hibernate.ddl-auto=false                #false for persistent database
```

You can check our project's application properties file via here: application.properties

Go back to TOC

8 H2 Database Preparation

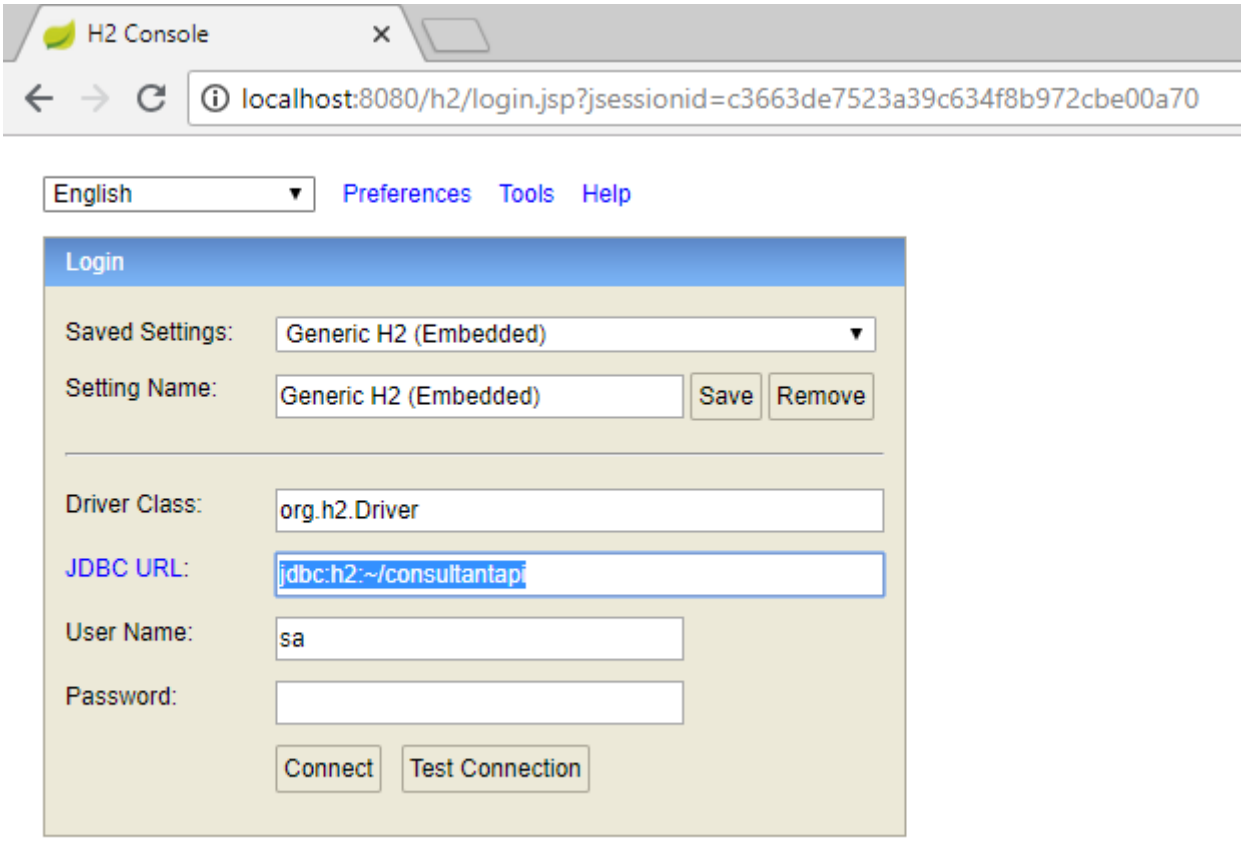
Before using our database implementation instead of using our stub implementation, we need to prepare our table and initial entries within the database. We have two things to do;

1. We need to create a CONSULTANT table to store our consultant model, defined in Consultant class.

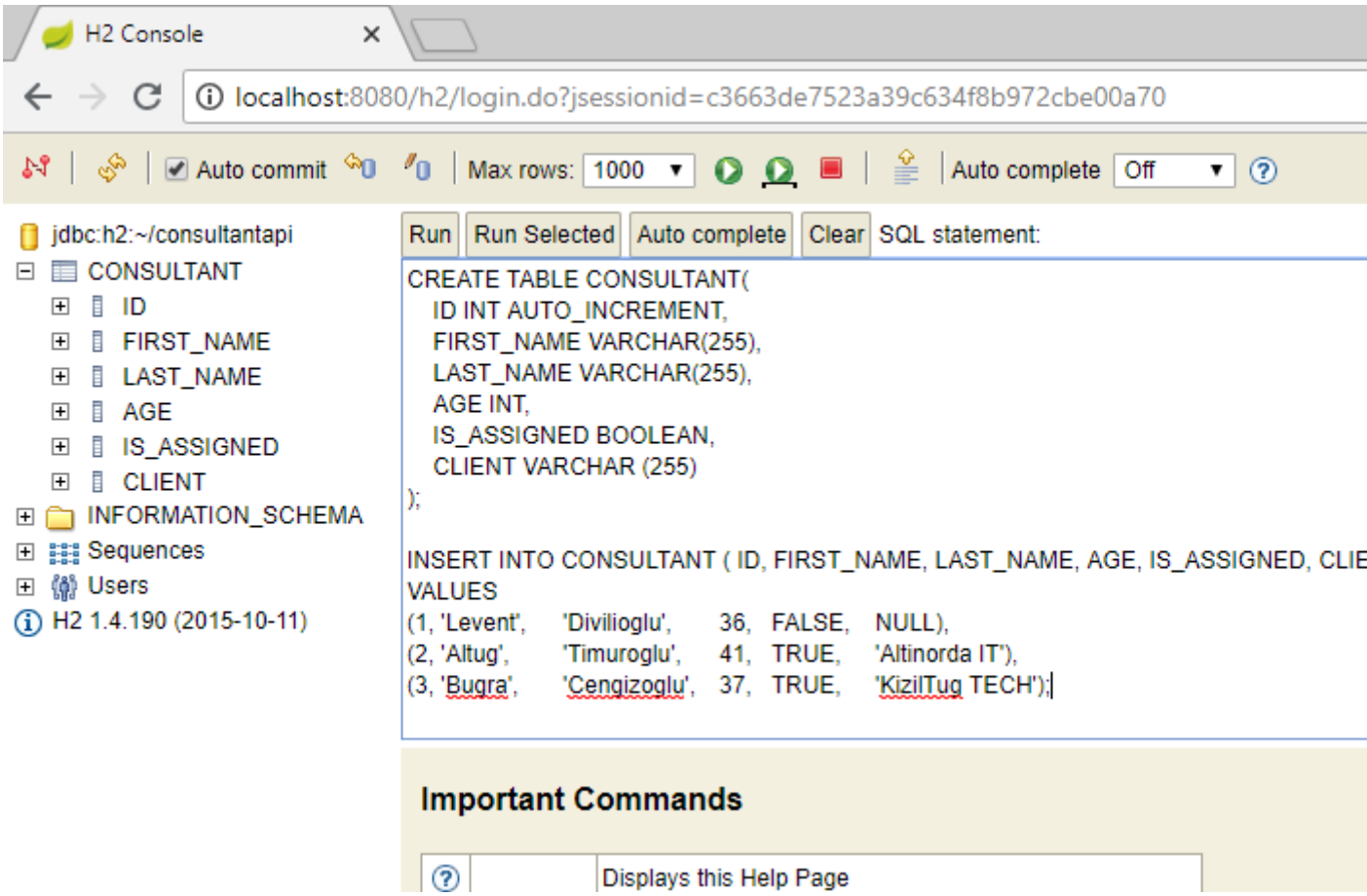
2. We need to insert some trivial entries to the table.

The related SQL commands are located under the consultant.sql

As it is written on previous chapter, we have defined our H2 console with "/h2" postfix. Also we have defined our server port as "8080", so connection url will be http://localhost:8080/h2/. We can connect our H2 database console as below;



After connecting to the database via the console, we can easily run our sql commands defined in consultant.sql as below;



Now our database is ready to go!

Go back to TOC

9 Sending And Receiving JSONs With Postman

In this part, I'm going to demonstrate all operations defined in our ConsultantController class and how to test them either with our web browser or with the Postman tool. We can test our GET methods with any web browser but for operations that use HTTP POST, PUT, DELETE methods, we cannot execute them with a simple web browser, so I'm going to use Postman toolfor that.

You can download Postman via this link

I've provided CRUD operations within postman, so that you can load all the prepared operations in Postman tool. You can find the content under misc directory;

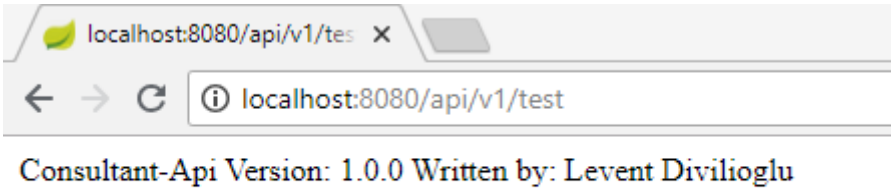
Postman Collection

9-a Test

```
Sub Path: /test
Full URL: http://localhost:8080/api/v1/test
Method:   GET
Sends:    N/A
Receives: Text
Sample Input: N/A
Sample Output;

Consultant-Api Version: 1.0.0 Written by: Levent Divilioglu
```

We can simply use our web browser and receive the text output as below;



But take into the consideration that this response based on which implementation we define on our custom configuration file which is: implementation.properties.

You can select one of the four different implementations via the configuration file and see the results. For more information, you can go back to the 6 External Configuration Example section.

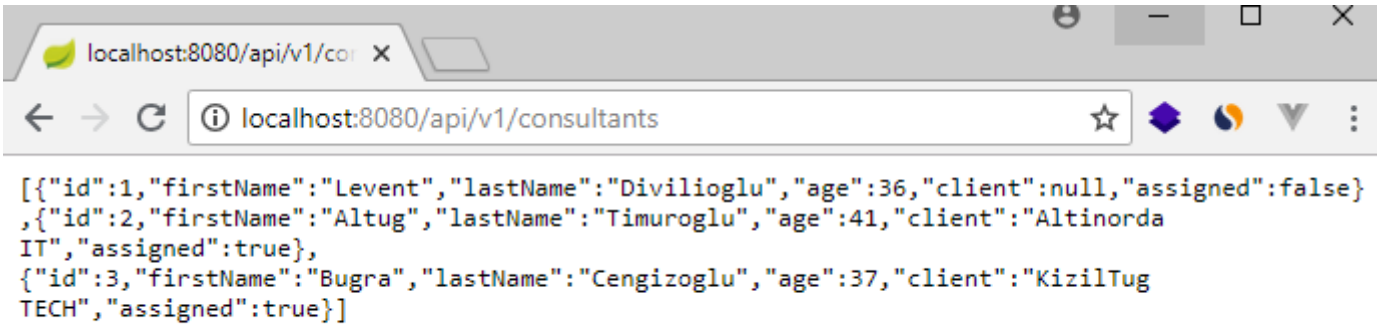
Go back to Sending And Receiving JSONs With Postman
Go back to TOC

9-b List

```
Sub Path: /consultants
Full URL: http://localhost:8080/api/v1/consultants
Method:   GET
Sends:    N/A
Receives: JSON
Sample Input: N/A
Sample Output;
[{
        "id": 1,
        "firstName": "Levent",
        "lastName": "Divilioglu",
        "age": 36,
        "client": null,
        "assigned": false
},
{
        "id": 2,
        "firstName": "Altug",
        "lastName": "Timuroglu",
        "age": 41,
        "client": "Altinorda IT",
```

```
        "assigned": true
},
{
        "id": 3,
        "firstName": "Bugra",
        "lastName": "Cengizoglu",
        "age": 37,
        "client": "KizilTug TECH",
        "assigned": true
}]
```

Again we can use web browser to get the results as below;

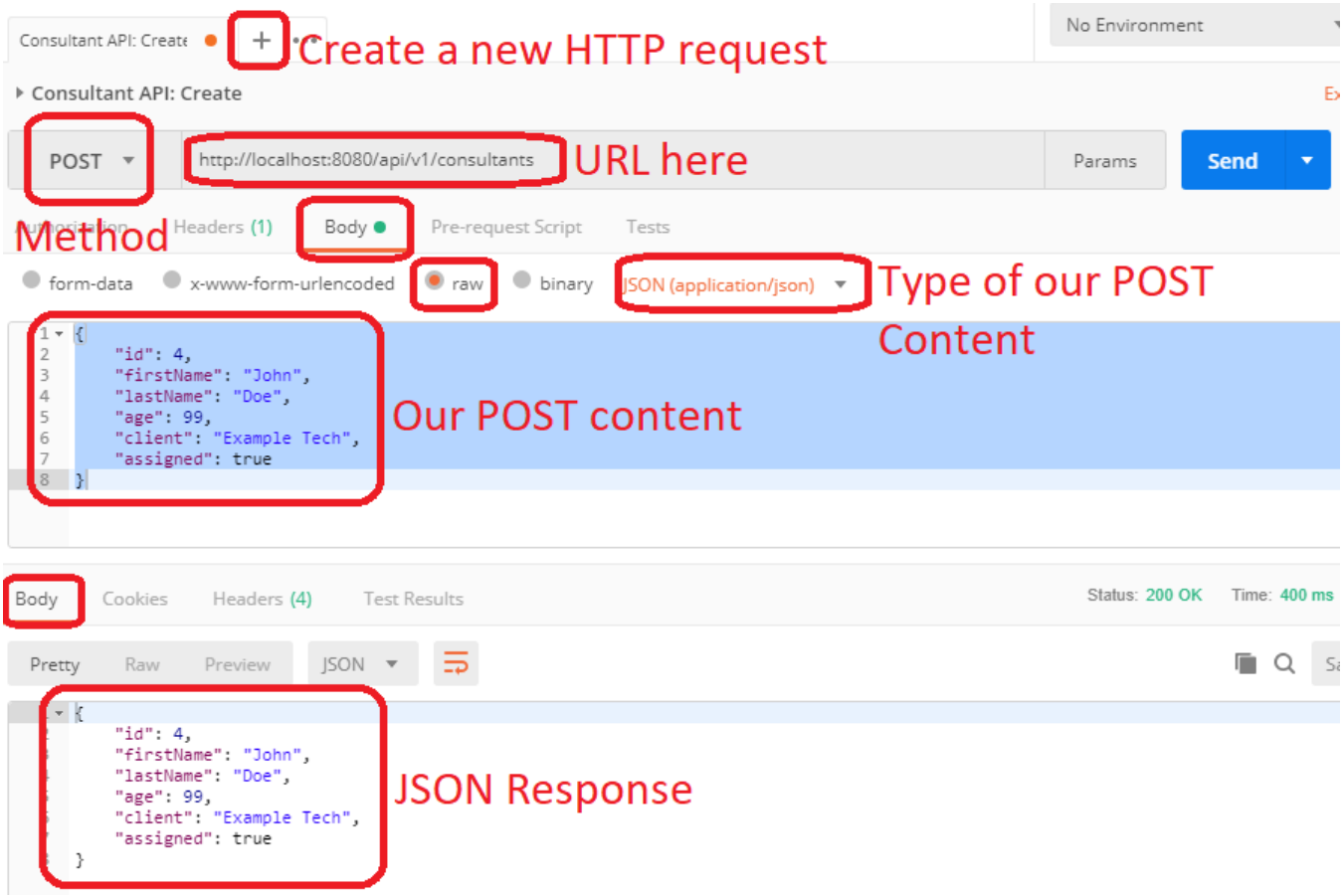


Go back to Sending And Receiving JSONs With Postman
Go back to TOC

9-c Create

```
Sub Path: /consultants
Full URL: http://localhost:8080/api/v1/consultants
Method:   POST
Sends:    JSON
Receives: JSON
Sample Input;
{
        "id": 4,
        "firstName": "John",
        "lastName": "Doe",
        "age": 99,
        "client": "Example Tech",
        "assigned": true
}
Sample Output;
{
    "id": 4,
    "firstName": "John",
    "lastName": "Doe",
    "age": 99,
    "client": "Example Tech",
    "assigned": true
}
```

This time, the operation we are using is POST, so we cannot do that with our browser, we have to use our tool Postman. Here is how I create my HTTP request;



With the '+' sign above on the Postman screen, we can create our HTTP request. Then we select HTTP Method as POST. We paste the URL, select "raw", and JSON for our input content. Then we select the "body" tag, and paste our content which we want to POST. Under the second half of the screen, on the "body" tag, we will retrieve our JSON response.

As you can see, the created content is returned. We can also test the result using our list service via postman (or web browser);

Go back to Sending And Receiving JSONs With Postman
Go back to TOC

9-d Retrieve

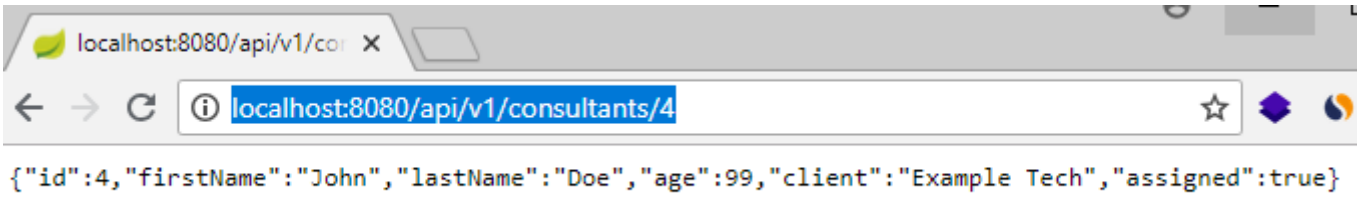
```
Sub Path: /consultants
Full URL: http://localhost:8080/api/v1/consultants/{id}
Method:   GET
Sends:    N/A
Receives: JSON
Sample Input: N/A
Sample Output;
{
    "id": 4,
    "firstName": "John",
    "lastName": "Doe",
    "age": 99,
    "client": "Example Tech",
```

```
    "assigned": true
}
```

It is a simple GET again, and let's use our browser for testing. We are going to use a path parameter which will be "4", the full path is as below;

```
http://localhost:8080/api/v1/consultants/4
```

The output will be as follows;



Go back to Sending And Receiving JSONs With Postman
Go back to TOC

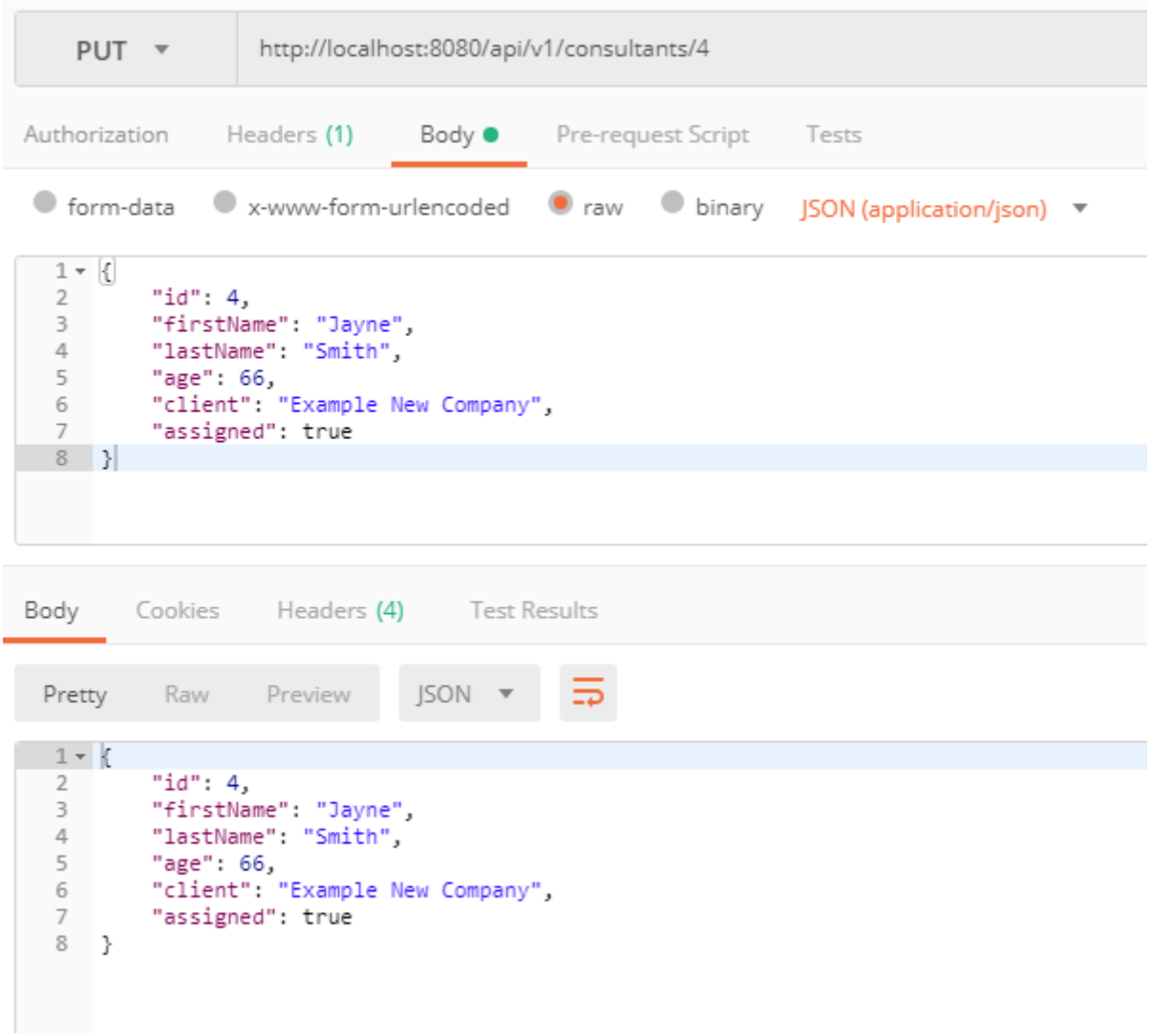
9-e Update

```
Sub Path: /consultants
Full URL: http://localhost:8080/api/v1/consultants/{id}
Method:   PUT
Sends:    JSON
Receives: JSON
Sample Input;
{
        "id": 4,
        "firstName": "Jayne",
        "lastName": "Smith",
        "age": 66,
        "client": "Example New Company",
        "assigned": true
}
Sample Output;
{
    "id": 4,
    "firstName": "Jayne",
    "lastName": "Smith",
    "age": 66,
    "client": "Example New Company",
    "assigned": true
}
```

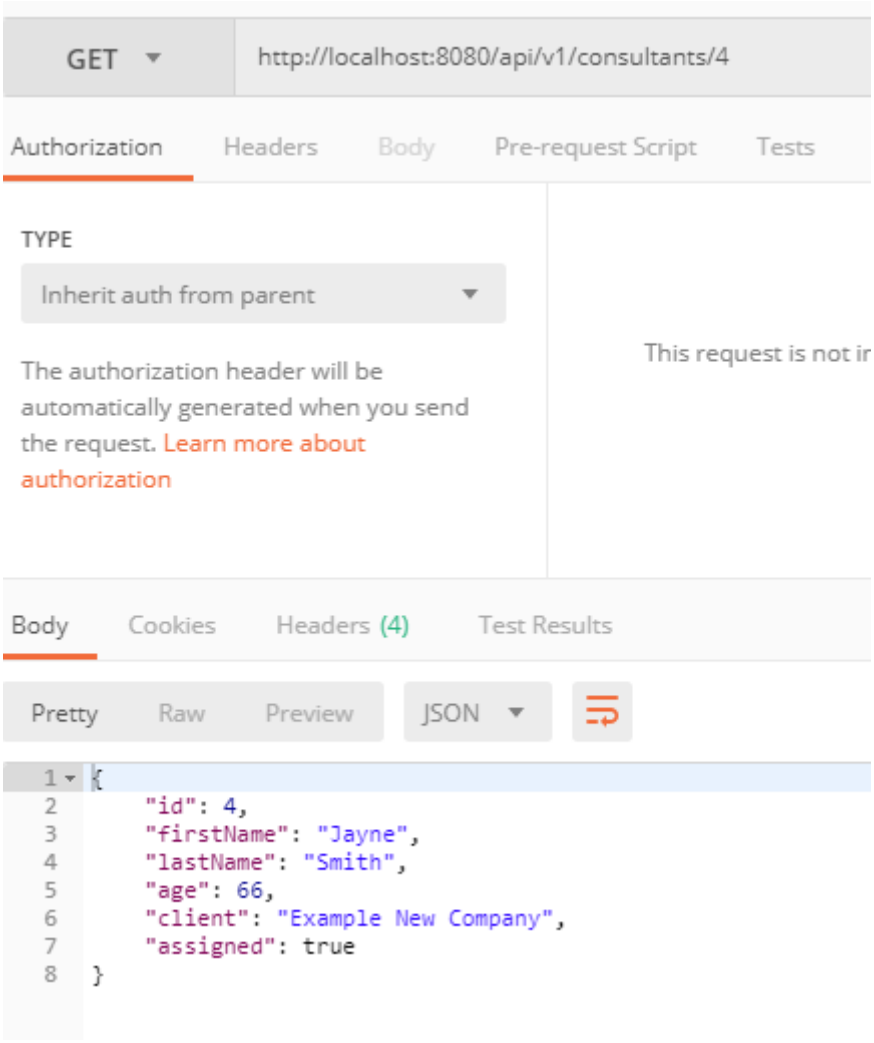
In order to update, we are going to use PUT method, so we will use Postman again.

For update method, we have to give the id in the URL as a path parameter, so URL will be as below;

```
http://localhost:8080/api/v1/consultants/4
```



As you can see, the updated content is returned. We can also test the result using our retrieve service via postman (or web browser);



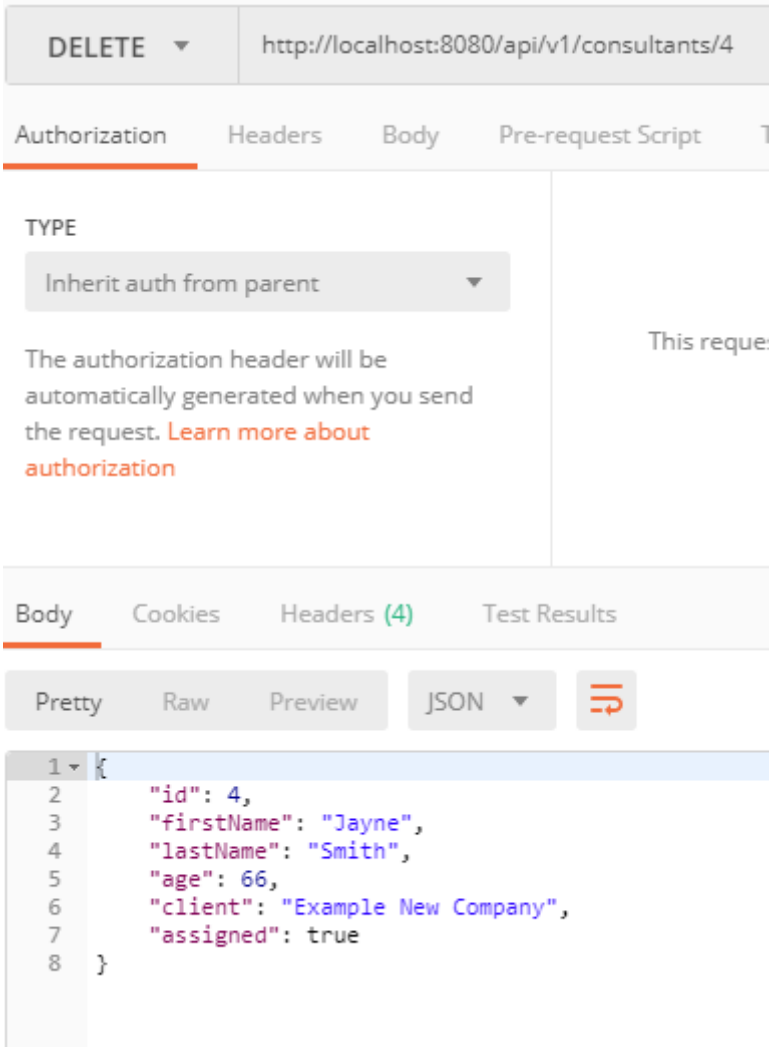
Go back to Sending And Receiving JSONs With Postman
Go back to TOC

9-f Delete

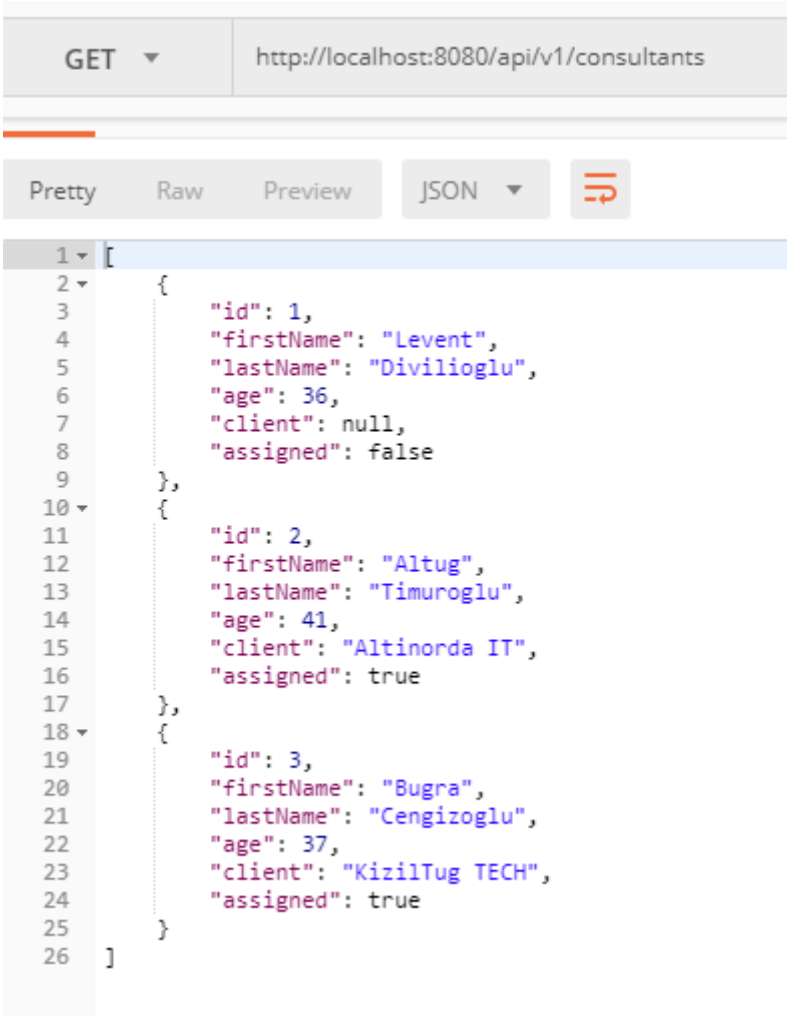
```
Sub Path: /consultants
Full URL: http://localhost:8080/api/v1/consultants/{id}
Method:   DELETE
Sends:    N/A
Receives: JSON
Sample Input: N/A
Sample Output;
{
    "id": 4,
    "firstName": "Jayne",
    "lastName": "Smith",
    "age": 66,
    "client": "Example New Company",
    "assigned": true
}
```

Again, we will use Postman for DELETE operation. As it is shown above, we don't have to provide a body for this operation, but we have to provide the id of the consultant to be deleted within the URL as below;

```
http://localhost:8080/api/v1/consultants/4
```



As you can see, the deleted content is returned. We can also test the result using our retrieve service via postman (or web browser);



Go back to Sending And Receiving JSONs With Postman
Go back to TOC

10 Building And Running The Standalone Application

Now we can demonstrate how to run our consultant-api as a standalone application. First we must build with the following maven command;

```
mvn clean package
```

This command is going to collect all the needed jars and pack them into an Uber Jar (a.k.a. fat jar). We can find this Uber Jar under the "target" folder. The name of the file will be: "consultant-api-1.0-SNAPSHOT.jar"

We are going to take this file and copy it to another arbitrary folder. Remember that we also need two configuration files, those that located under the config file. We will also copy those config files to our arbitrary folder.

I copied all the files I mentioned above to the folder "D:\consultant-api", the structure is as follows;

```
D:\consultant-api
       |
       |___ consultant-api-1.0-SNAPSHOT.jar
       |___ config
              |_______ application.properties
              |_______ implementation.properties
```

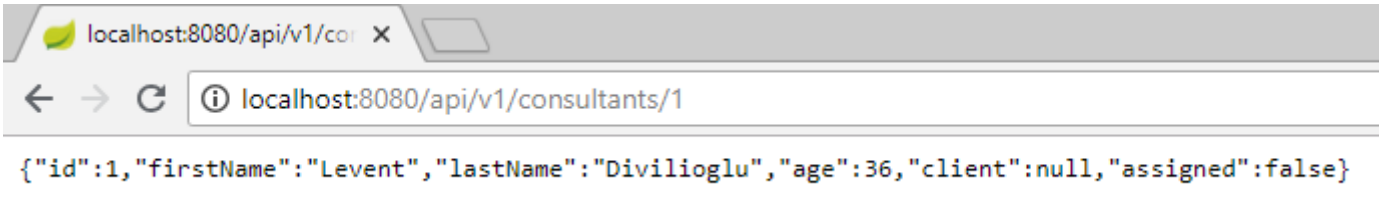
If the structure of the arbitrary folder (here it is 'consultant-api'), then we can try to run our standalone application to see if it is working. Here is a successful output. For the simplicity, I'm not going to paste all the log output;

```
D:\consultant-api>java -jar consultant-api-1.0-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::         (v1.0-SNAPSHOT)

2018-07-01 16:16:08.705  INFO 6616 --- [           main]
com.levent.consultantapi.EntryPoint      : Starting EntryPoint v1.0-SNAPSHOT on
LEVASUS with PID 6616 (D:\consultant-api\consultant-api-1.0-SNAPSHOT.jar started
by Levent in D:\consultant-api)
2018-07-01 16:16:08.711  INFO 6616 --- [           main]
com.levent.consultantapi.EntryPoint      : No active profile set, falling back to
default profiles: default
2018-07-01 16:16:08.785  INFO 6616 --- [           main]
ationConfigEmbeddedWebApplicationContext : Refreshing
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationCo
ntext@27808f31: startup date [Sun Jul 01 16:16:08 CEST 2018]; root of context
hierarchy
2018-07-01 16:16:10.487  INFO 6616 --- [           main]
o.s.b.f.s.DefaultListableBeanFactory     : Overriding bean definition for bean
'beanNameViewResolver' with a different definition: replacing [Root bean: class
[null]; scope=; abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0;
autowireCandidate=true; primary=false;
factoryBeanName=org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfigurati
on$WhitelabelErrorViewConfiguration; factoryMethodName=beanNameViewResolver;
initMet
```

Then we can test if our standalone application is working fine with our web browser;



Yes, our standalone application is working fine. We can easily deploy it in a Docker container, or just run it as it is.

Go back to TOC