Rest Assured API Testing

A Complete Road Map to REST API Automation

Using JAVA and TestNG

Category: Beginners to Intermediate
Pre-requisite: Java, OOPs and willingness to learn

Er. Raghubir Singh
CERTIFIED TEST PROFESSIONAL

Acknowledgement

Special thanks to my mentor Deepak Chanana
Who's great guidance cannot be expressed into words ...

And

Insaaf Awareness Movement – NGO®
Working for Men Rights in India – Men in Distress
Help line No: 8882 498 498

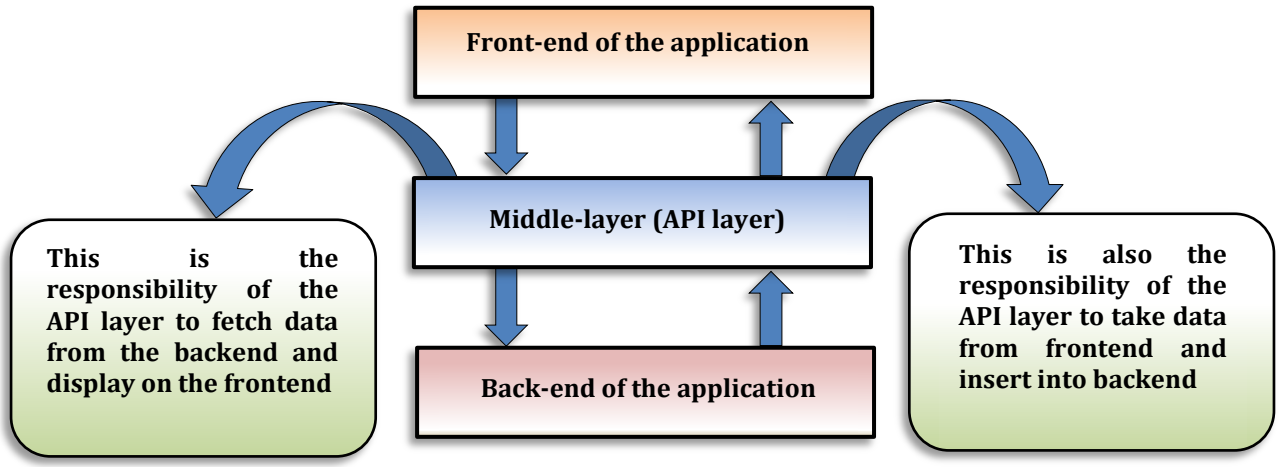
Dedicated to Dr.Tejinder from Canada who inspired me to restart my career in IT

Table of Contents

Chapter -1: Introduction to API Automation

Application Program Interface (API): Is the way of communication between two applications where applications may differ in their platforms or in terms of technology also.

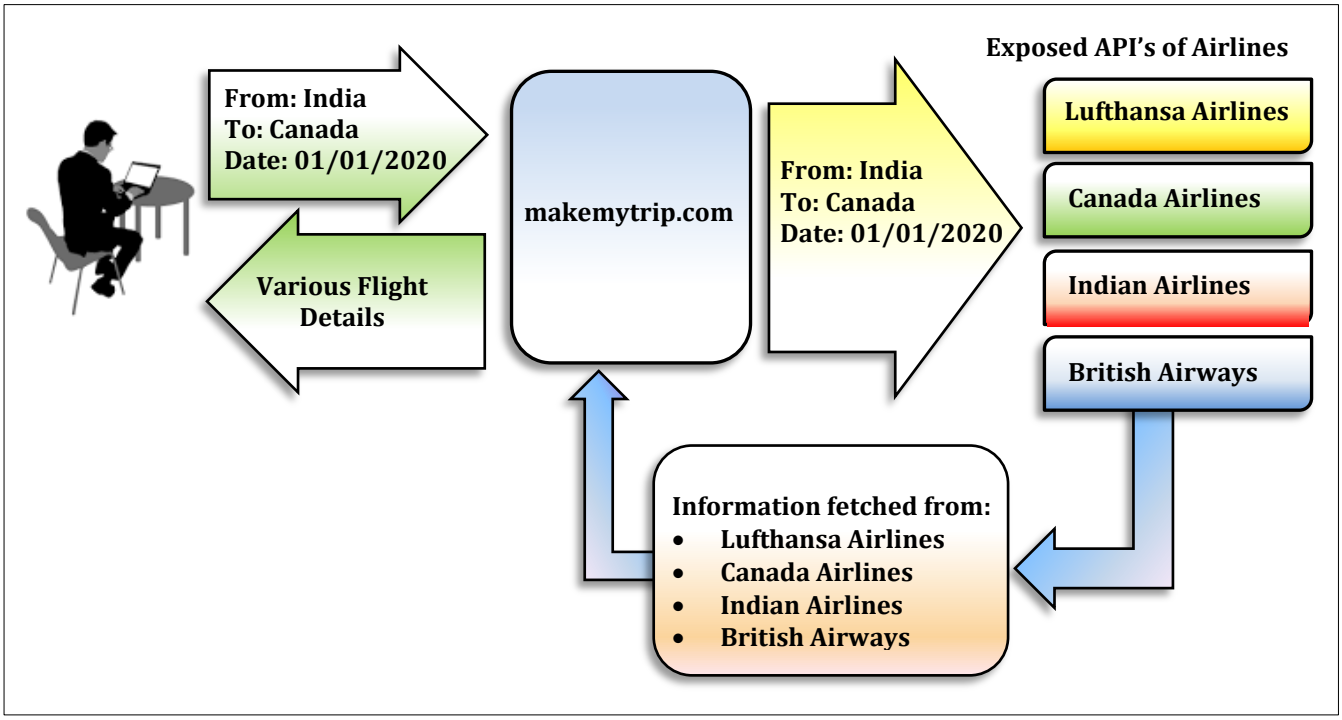
Diagrammatically Representing And Understanding What Exactly API's Are:



Chapter-1 Figure 1

So, from the above we can say that API is a way of communication between two layers.

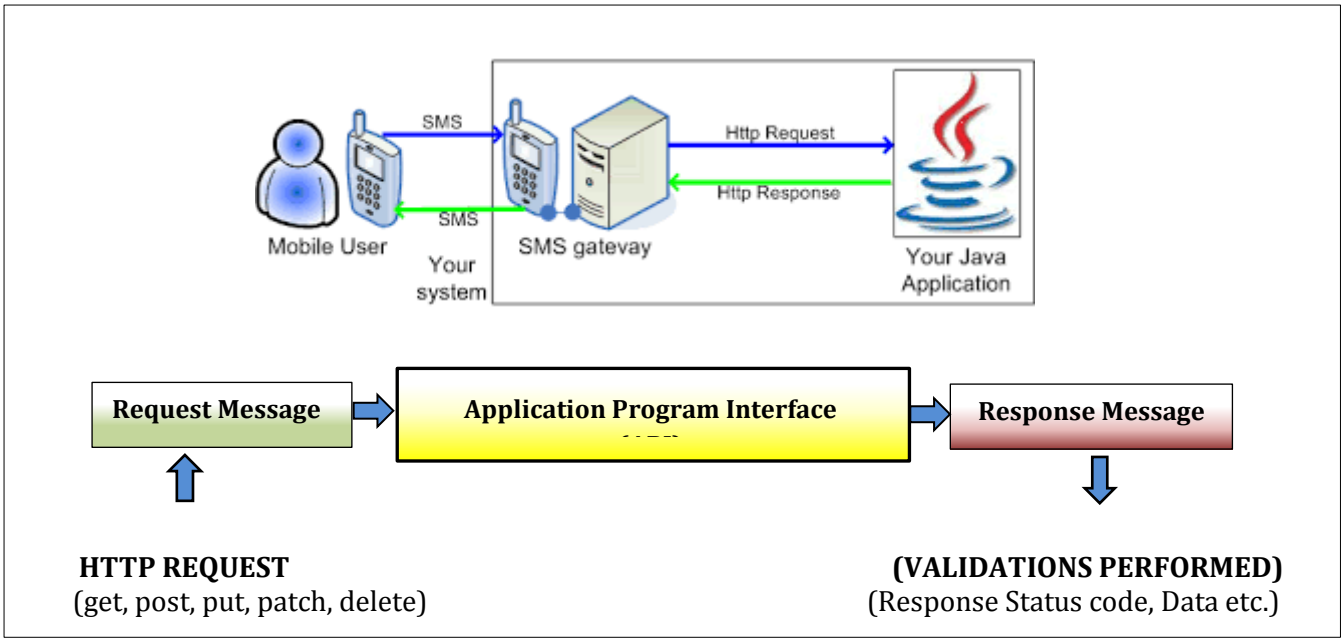
It is very obvious more the API layer is secure more it will be able to fetch data from backend and show on the frontend. It means if API layer has been tested correctly it will show the correct data on the frontend and ensures that the data insertion will also work fine.
For An Instance: Makemytrip.com is consuming API's exposed by various airlines and when we search for some flight it fetches the information from those exposed API's of various airlines.



Chapter-1 Figure 2

Another example is **whatsapp** application that uses locations feature of google map application. Google have exposed its map API and whatsapp is using that API for its own purpose. Please note that neither the map feature was developed by whatsapp nor it is in the same technology that was used to develop map by google but still whatsapp is able to use google maps feature just because of the API's of google map inspite of the additional fact that google is web-based application while whatsapp is mobile application.
Apart from that **bookmyshow.com** and **Paytm** is another example of API's.

Why API Automation Is Faster Than Frontend Application?
Let's understand this with example: To test tweet and retweet feature using selenium will roughly take around a minute (entering the url, sign in, clicking new post typing post, clicking post etc.) and when performed through exposed API of tweeter it will be done in 3-4 seconds as we just have to enter the data and validate the response.



Chapter-1 Figure 3

What Do We Mean By Features And Resources In Context of API Testing?
'Feature' is the term used in manual testing to test some functionality and similarly 'Resource' is the term used in API Automation testing referring some functionality.

Types Of API: There are two types of API Automation, Simple Object Access Protocol (SOAP) and REST Assured (Representational State Transfer). Both are the web services.

What Is The Different Between API And Web Services?
There are two types of API's- Soap and REST Assured. Web Services are services used by the applications communicating over the same network (generally internet) and API does not always require any such internet connection as said earlier, it is a communication layer between the applications which may are may not be using internet. All API's are not web services.

API v/s Web Service
Web services facilitate communication between two web parties in compliance with communication standards and protocols like Extensible Markup Language (XML), Simple Object Access protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration) UDDI and all of these collectively known as Web Services. API and Web service serve as a means of communication. The only difference is that a Web service facilitates interaction between two machines over a network. An API acts as an interface between two different applications so that they

can communicate with each other. An API is a method by which the third-party vendors can write programs that interface easily with other programs. A Web service is designed to have an interface that is depicted in a machine- executable format usually specified in Web Service Description Language (WSDL- generally is an API). Typically, "HTTP" is the most commonly used protocol for communication. Web service also uses SOAP, REST, and XML-RPC as a means of communication. API may use any means of communication to initiate interaction between applications. For example, the system calls are invoked using interrupts by the Linux kernel API.
An API exactly defines the methods for one software program to interact with the other. When this action involves sending data over a network, Web services come into the picture. An API generally involves calling functions from within a software program.

In case of Web applications, the API used is web based. Desktop applications such as spreadsheets and word documents use VBA and COM-based APIs which don't involve Web service. A server application such as Joomla may use a PHP-based API present within the server which doesn't require Web service.

A Web service is merely an API wrapped in HTTP. An API doesn't always need to be web based. API consists of a complete set of rules and specifications for a software program to follow in order to facilitate interaction. A Web service might not contain a complete set of specifications and sometimes might not be able to perform all the tasks that may be possible from a complete API.

The APIs can be exposed in a number of ways which include: COM objects, DLL and .H files in C/C++ programming language, JAR files or RMI in Java, XML over HTTP, JSON over HTTP, etc. The method used by Web service to expose the API is strictly through a network.

Summarizing:
1. All Web services are APIs but all APIs are not Web services.
2. Web services might not perform all the operations that an API would perform.
3. A Web service uses only three styles of use: SOAP, REST and XML-RPC for communication whereas API may use any style for communication.
4. A Web service always needs a network for its operation whereas an API doesn't need a network for its operation.
5. An API facilitates interfacing directly with an application whereas a Web service interacts with two machines over a network.

What Is The Difference Between SOAP And REST Assured?

	SOAP Services	REST Assured
1.	In case of soap services developer provides single WSDL (web services description language) file which all the operations/ features/functionality/resources are defined	Developer provides URI(uniform resource indicator) for testing each functionality/resource or WADL file may be provided
2.	It only supports xml format for communication	Supports xml, JSON, Plain text
3.	It is heavier version as compared to REST in the context that it contain all the operations in a single file	As every resource has its own URI therefore it is less complex and could be said lighter version.
4.	It is a protocol and therefore Soap cannot use REST services	REST is an architectural style and can use Soap web services as it is a concept and can use protocols like HTTP.
5.	Soap uses services interface to expose business logic and define strict xml standards	REST uses URI to expose business logic and does not defined too much standards

Chapter-1 Table -1

URI- Uniform Resource Indicator. Uniform resource indicator is a link at which one has to perform API testing. It is provided by the developer. It consist of two parts- the Base and the end-point. Precisely look at the URI:- "http://localhost:3000/friends"

http://localhost:3000 **-Base**	/friends **-End-point**
(Location where API is deployed)	(Generally resource- functionality)

Example: http://localhost:3000/friends. Here, "/friends" is the resource and rest of the portion "http://localhost:3000" is the part of Base and is the location where the API is deployed.

HTTP and HTTP Requests: HTTP stands for Hyper Text Transfer Protocol. World Wide Web (WWW) is all about communication between web clients and servers. Communication between client computers and web servers is done by sending HTTP Requests and receiving HTTP Responses.

The data travels between the request and response in other formats like JSON – JavaScript Object Notation which is one of the most commonly used and accepted format for data transmissions and we will also be following the same as discussed in subsequent chapters.

Communication between clients and servers is done by requests and responses:

- A client (a browser) sends an HTTP request to the web
- An web server receives the request
- The server runs an application to process the request
- The server returns an HTTP response (output) to the browser
- The client (the browser) receives the response

A typical HTTP request / response circle:

- The browser requests an HTML page. The server returns an HTML file.
- The browser requests a style sheet. The server returns a CSS file.
- The browser requests an JPG image. The server returns a JPG file.
- The browser requests JavaScript code. The server returns a JS file
- The browser requests data. The server returns data (in XML or JSON).

Below are the various http requests or methods we use in API automation testing

	Http Request	Description
1.	**Get**	No data is required for this request. it is used to fetch the data from the given resource location.
2.	**Post**	Data is required for this kind of request. It is used to insert the data at the given resource location.
3.	**Put**	Data is required for this request. Whole of the data is passed that is to be updated and the one that is not to be updated. Requires certain ID value too to update that particular record.
4.	**Patch**	Data is required to for this http request also. Only that data is passed which is required to be updated, Requires certain ID value too to update that particular record
5.	**Delete**	No data is required to this request. It deletes the record (data) from the given resource location. Requires certain ID value too to update that particular record.

Chapter-1 Table 2

Where the get, put and delete are Idempotent methods but the post is non-idempotent method that is repeation of the post request always created new resource.

(HTTP REQUEST)
(get, post, put, patch, delete)

(VALIDATIONS PERFORMED)
(Response Status code, Data etc.)

Basics Tasks We Have To Do In API Automation Testing:
We will be performing following steps in API Automation:

1. As we know we require data for the http request like post, put and patch so, we will be creating body data with various techniques.
2. As we will be sending various http requests and so will get some response also, we will be validating the response like status code.
3. We will also be validating response data as well.
4. We will finally be doing response data reusability (known as response data parsing).

JavaScript Object Notation (JSON)
JSON stands for JavaScript Object Notation. JSON is a lightweight format for storing and transporting data. Just remember two things at the moment that ***Json have data in the form of object(s) and Array(s).***

JSON is a ***data format/Notation*** that is used by the web services to transmit over the communication channel for an instance JSON is often used when data is sent from a server to a web page. JSON is "self-describing" and easy to understand.

JSON Example
This example defines an Student object: an array of 3 employee records (objects):

```
{
"Student":[
  {"firstName":"Johny", "lastName":"Doe"},
  {"firstName":"Ananiya", "lastName":"Smithy"},
  {"firstName":"Raghubir", "lastName":"Singh"}]
}
```

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

The JSON format is syntactically identical to the code for creating JavaScript objects. Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects. The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

JSON data (Simple JSON)
JSON data is written as name/value pairs, just like JavaScript object properties. A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
{"firstName":"Johny"}
```

JSON names require double quotes. JavaScript names do not.

JSON Objects (Complex JSON)
JSON objects are written inside curly braces. Just like in JavaScript, objects can contain multiple name/value pairs:

```
“Personal”:
{
{"firstName":"Johny",
"lastName":"Doe"
},
“Id”:”369”,
“Age”:”38”
}
```

JSON Arrays (JSON in the Form of Array)
JSON arrays are written inside square brackets.
Just like in JavaScript, an array can contain objects:

```
"Student":
[
  {"firstName":"Johny", "lastName":"Doe"},
  {"firstName":"Ananiya", "lastName":"Smithy"},
  {"firstName":"Raghubir", "lastName":"Singh"}
]
```

In the example above, the object "employees" is an array. It contains three objects. Each object is a record of a person (with a first name and a last name).

This is also to be remember that if Json data starts with ‘**{**‘ then it is said to be Json object and we have to use object of JSONObject class to handle it. If Json data starts with ‘**[**‘ then it is said to be Json Array and we will be using Object of JSONArray class to handle such data. This is discussed in details under Response data parsing chapter.

Important online tools: We can validate our Json file using online tool: https://jsonlint.com and can also view the structure of our Json file at https://jsonviewer.stack.hu

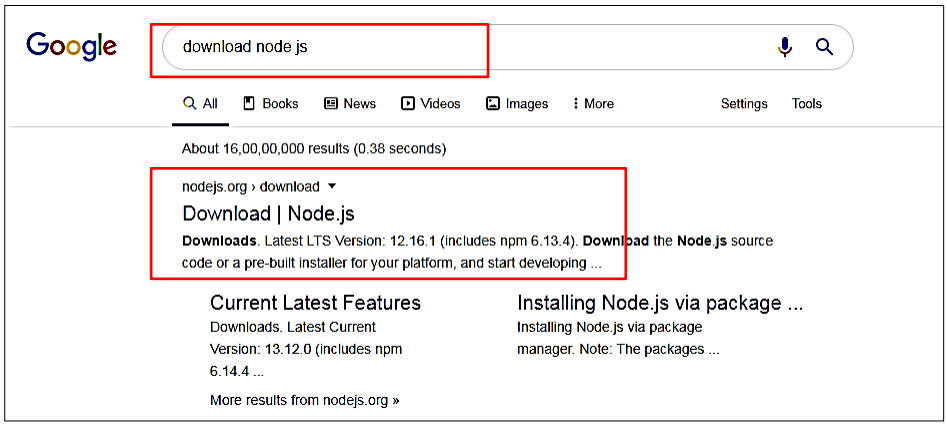
Chapter -2: Installing JSON Server for creating dummy API's

Creating Dummy API's for REST Assured API Automation Testing
At the moment, we do not have any real business exposed API with us for testing so, we will be creating dummy API's (where in the real time scenario they are provided by the developers) using NodeJS. ***npm package*** under Node JS will facilitates us further with installation of JSON Server where we can create dummy API's for our practice.

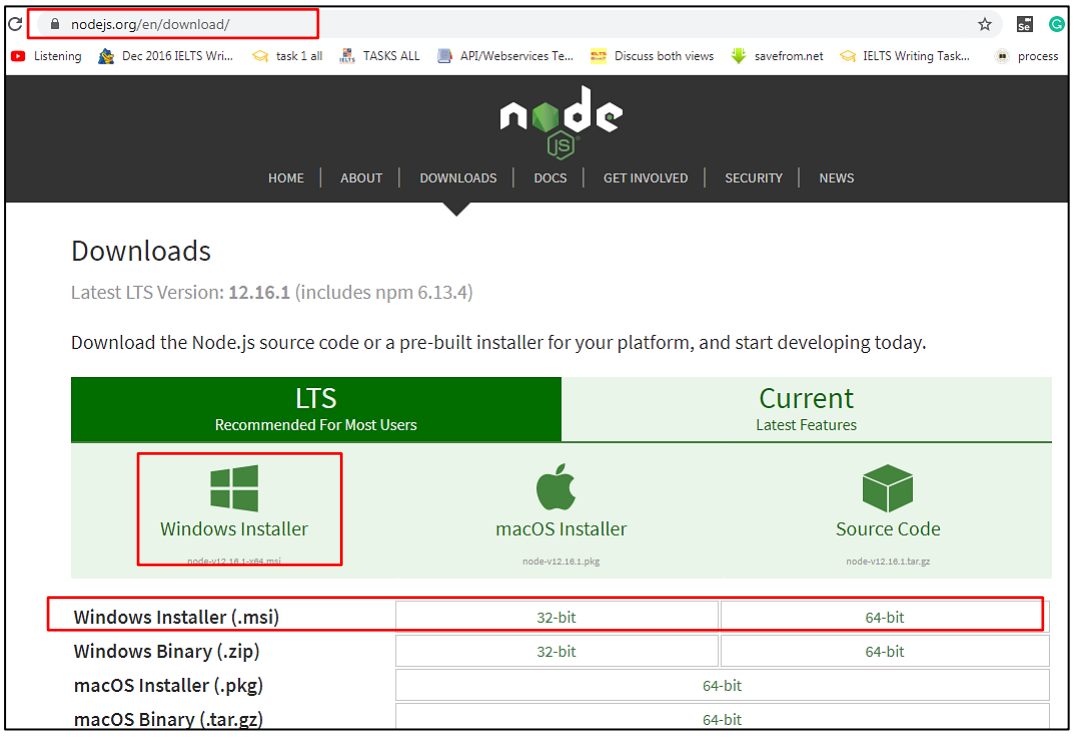
To install Json server first we need to install nodejs and npm (where npm is installed automatically when we install nodejs)

Installation and Configuration of NodeJS:

1. Goto google.com and search "download Node JS"

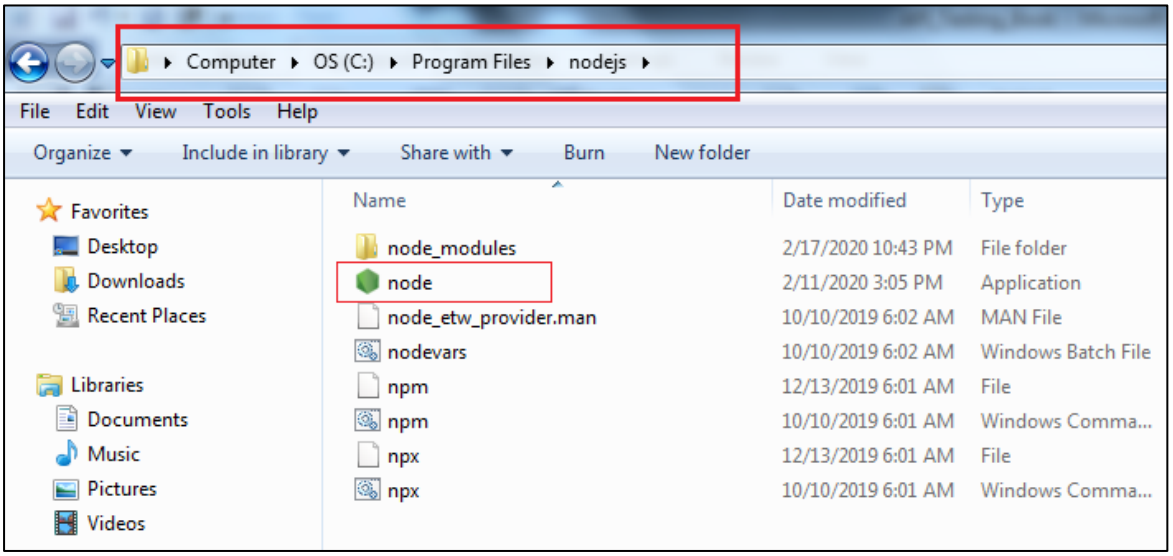


2. Download NodeJs from https://nodejs.org/en/download/



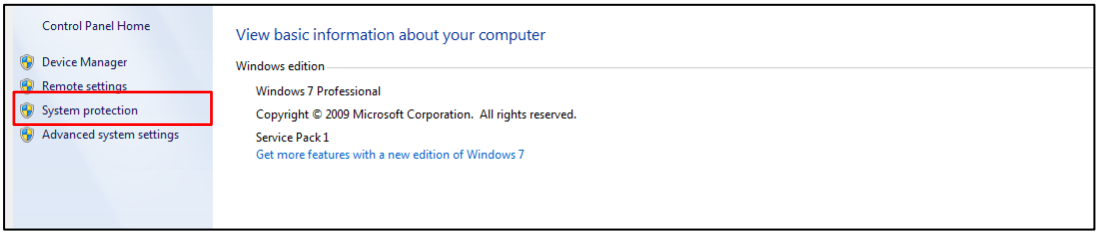
3. Select the installer according the your system and in my case windows installer has been selected

4. After downloading the same click next…next…next till finish.
5. You will find the nodejs setup under the nodejs folder generally create by default at C:\Program Files\nodejs. Run the setup. Click on nodejs folder and copy the path C:\Program Files\nodejs

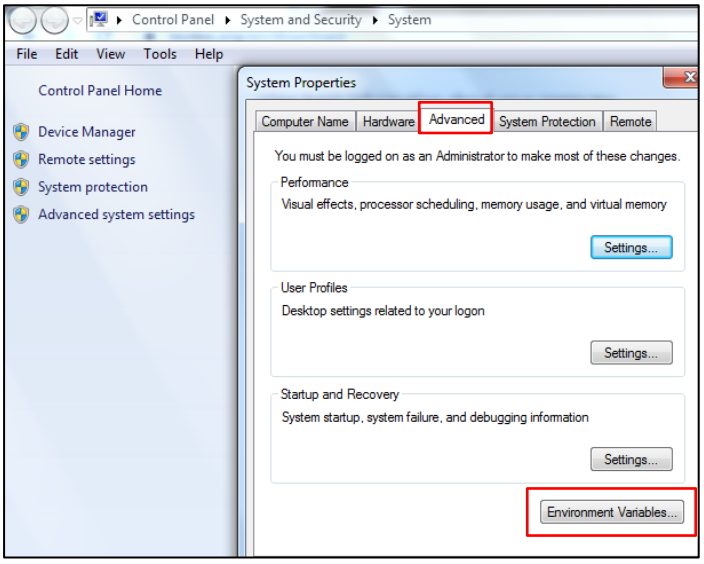


6. After setup we have to set its entry in the Path system variable for doing so follow the below steps:

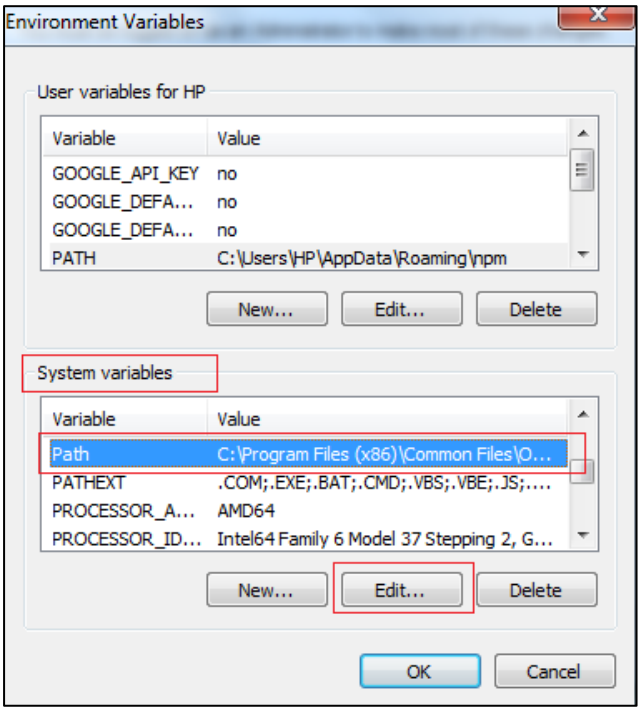
6.1 Right click on thisPC> Properties >Advance System Settings



6.2 Click on > Advance Tab and click Environment Variables



6.3 Select Path Variable under the System variables and click on Edit Button

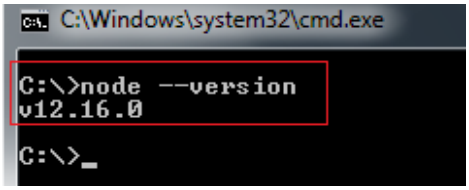


6.4 Put the semi colon and paste the path that was copied earlier in step 5 i.e. C:\Program Files\nodejs and click Ok…Ok and Ok.

6.5 Now again go to C:\Program Files\nodejs and double click on the folder node_modules and this time copy the path C:\Program Files\nodejs\node_modules.

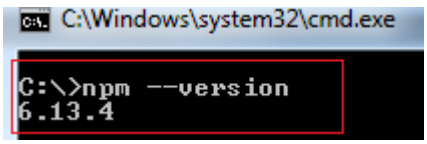
6.6 Repeat the steps 6.1 to 6.3 and put the semi colon and paste the path that was copied in step 6.5 C:\Program Files\nodejs\node_modules. Click Ok…Ok and Ok.

7. Now let's verify the nodejs setup done so far, go to the command prompt and type the command
C:\>node - -version



8. After installation of nodejs please notice that there is another file installed by the nodejs setup named as npm (node package manager) under the nodejs folder and this npm facilitates in installation of various packages of nodejs.

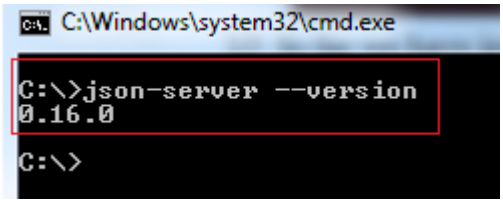
9. Go to command prompt and verify npm installation by typing the following command:
C:\>npm - -version



10. So far we have installed nodejs and npm and set the entries of nodejs and node_modules in our system variables which are the prerequisites to install the Json server. Why we want to

install the Json server because we want to create dummy API's for test automation. Let's install Json server by typing following command on the command prompt:
C:\> npm install –g json –server

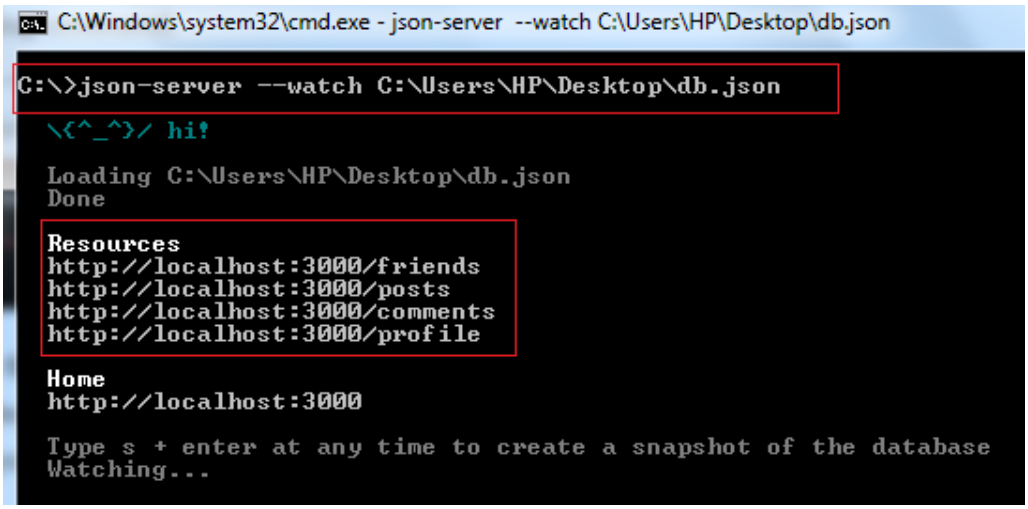
11. After that the json server will install and we will verify the same again on the command prompt by typing the following command:
C:\>json-server –version



So finally, we are all set with the starting point of API automation. Now we need some file that contains data and as we are doing API testing using JSON Server so we need a JSON data file. This data file is provided in the end in appendix-1 and you have to type it carefully in Notepad and save with valid name but extension must be .json only.

Now we will start Json server with the help of below command. We will also pass the path of above .json file containing our Json data:

C:\>json-server --watch <complete path where you have place .Json data file on your system)



The json server is up and it has created four dumpy API's for us ***as per the json data present in file:***

- http://localhost:3000/friends
- http://localhost:3000/posts
- http://localhost:3000/comments
- http://localhost:3000/profile

Now as the Json server is started and up at the moment, to terminate the server we have to type Ctrl+C command and then press Y as per confirmation message prompted below:

Chapter-3: Starting API Automation Testing Using REST Assured

REST Assured in Itself Is an API. It is an API designed to Test REST services and REST API's

Pre-requisites:

- Java and Eclipse IDE
- TestNG which is unit testing framework
- Maven (either we can add jar file manually to our project or we can create a maven project)

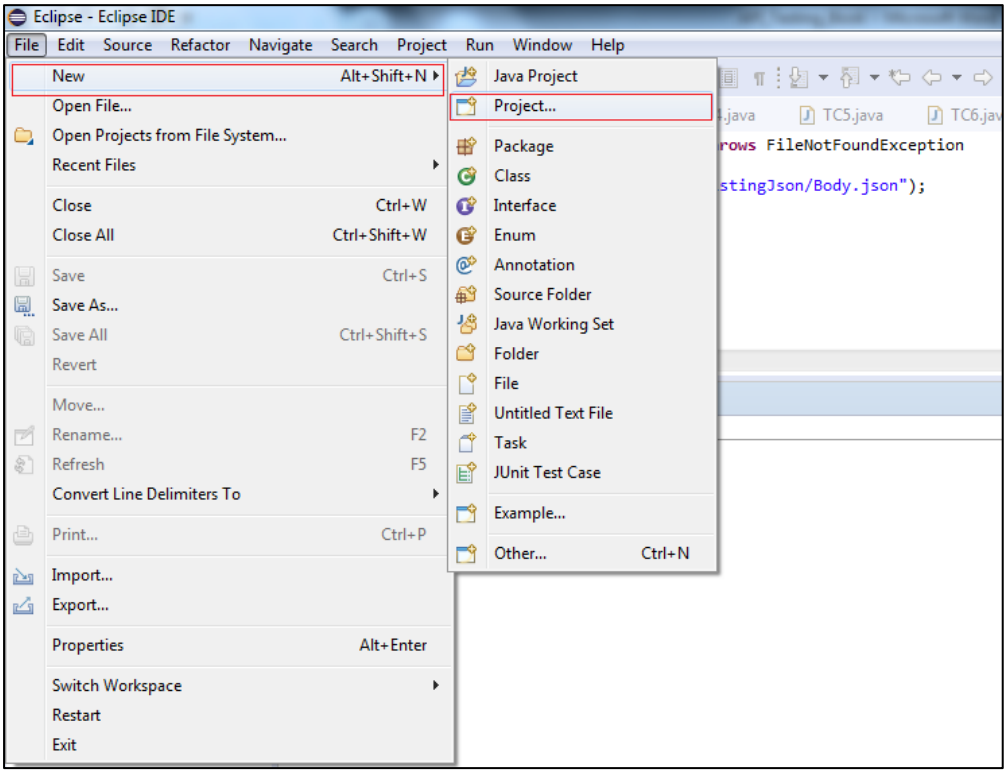
Rest Assured is a Java library to handle Rest API automation. There is a jar file (Java Archive) corresponding to each library and for each jar file there is a corresponding maven dependency.

We can either add the corresponding jar file with respect to the java library we want to use (say that of Rest Assured) or we can create the maven project, copy the maven dependency and paste the same under dependency tag in pom.xml file and let the system download the corresponding jar file automatically for us. We will opt the second option i.e. creating the maven project for REST Assured API testing and add the maven dependency in this book.

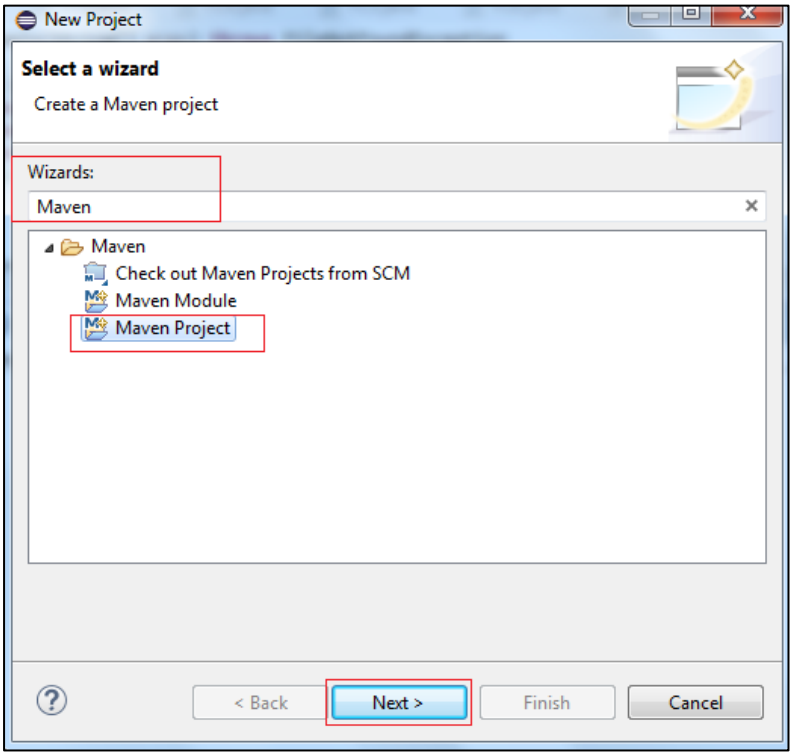
Maven is a build and project dependency management tool.

Creating First Maven Project

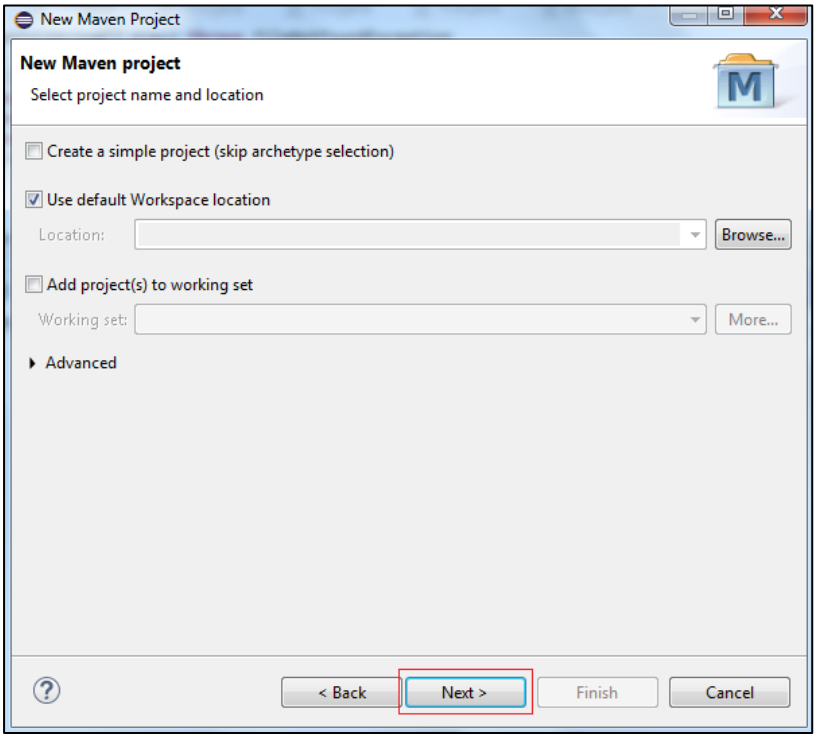
1. Open eclipse and click on File> New> Project



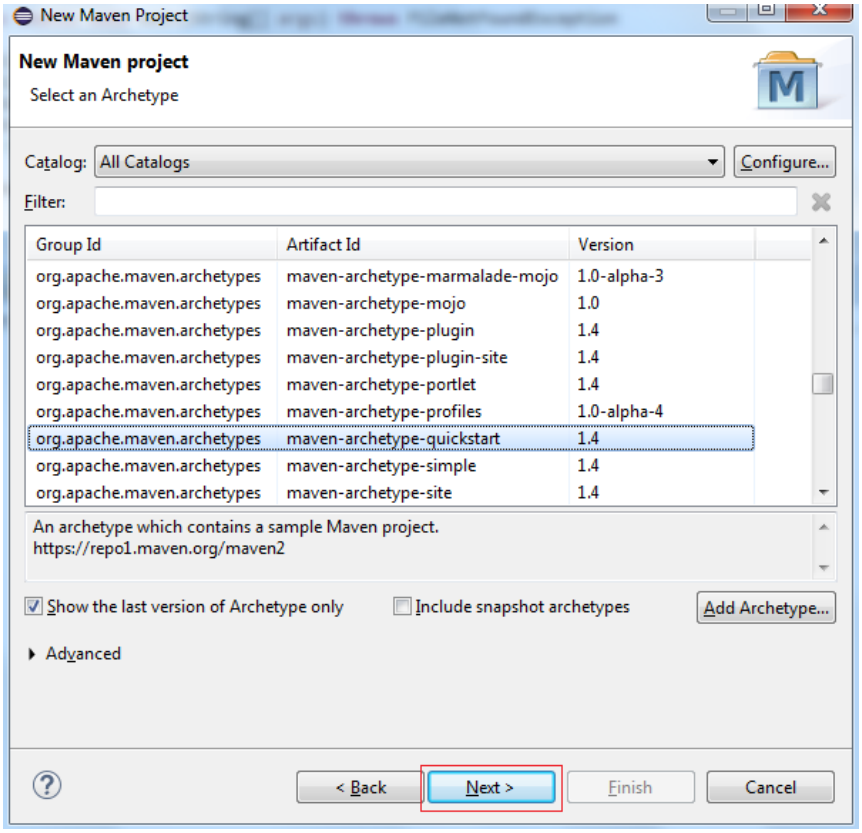
2. Write Maven in the wizard text box and select Maven from the list box as shown below. Click Next> button



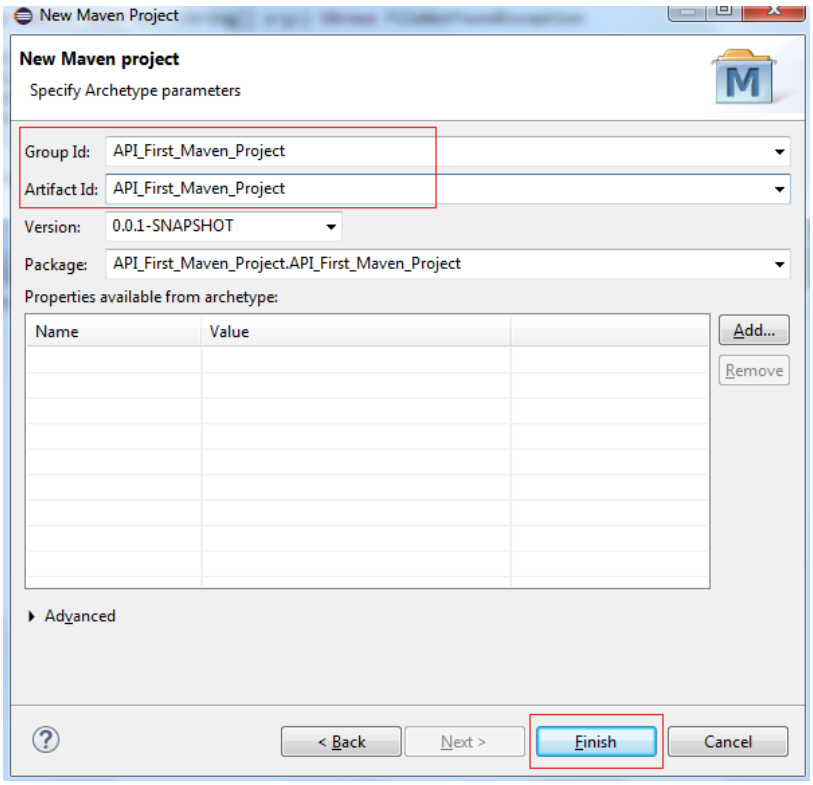
3. Simply click Next > button



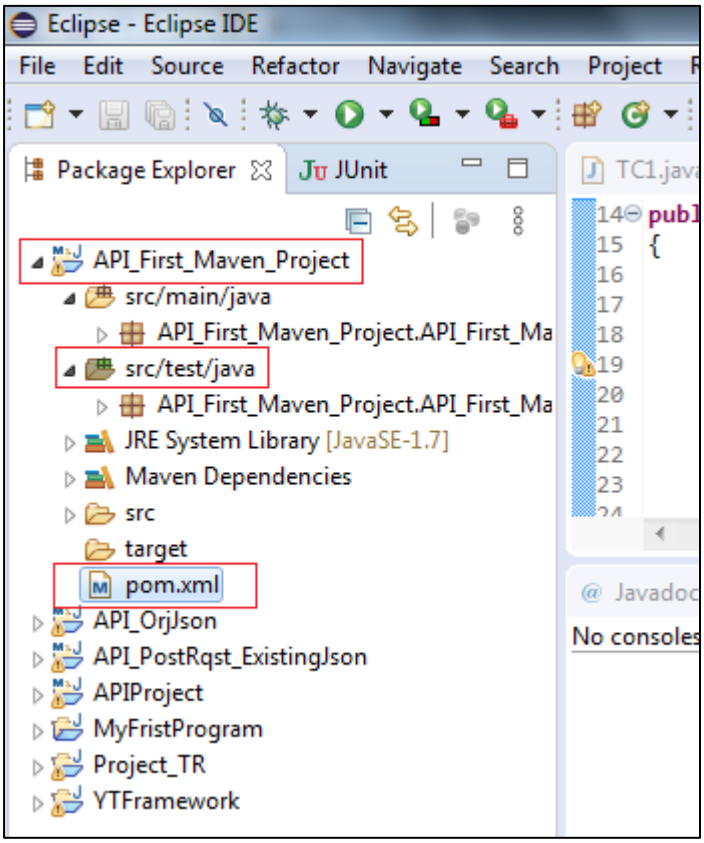
4. Click on Next button



5. Type in the Group Id and Artifact Id (You can give any name to that) and click Finish button



6. Finally Maven Project "API_First_Maven_Project " is created successfully. Please note that pom.xml is created by this Maven project as highlighted and shown below.



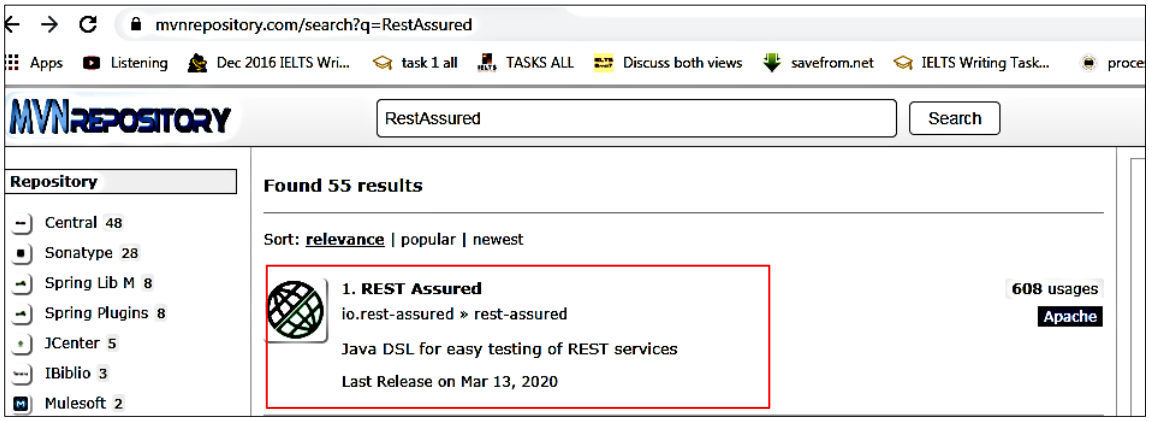
We have successfully created the Maven Project and now we will copy the various dependencies in the corresponding pom.xml file of the project so as to REST Assured API automation testing.

Including The Required REST Assured And Other Dependencies:

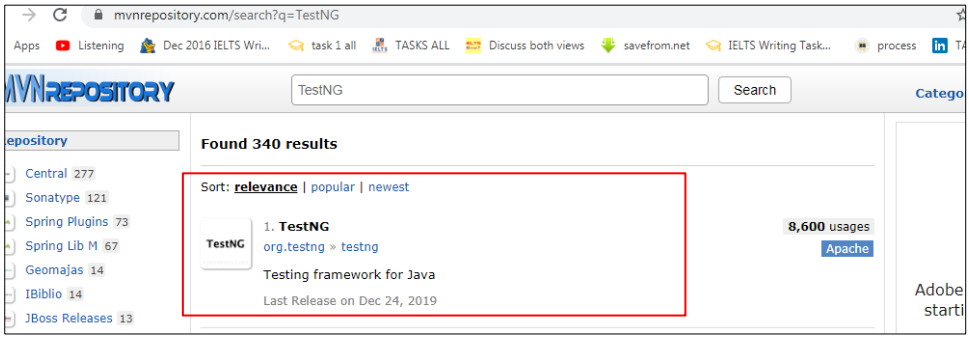
a. REST Assured- https://mvnrepository.com/artifact/io.rest-assured/rest-assured
b. TestNG- https://mvnrepository.com/artifact/org.testng/testng
c. JSON- Simple- https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple
d. Apache POI- https://mvnrepository.com/artifact/org.apache.poi/poi
e. Apache POI- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml

From Where To Get These Dependencies

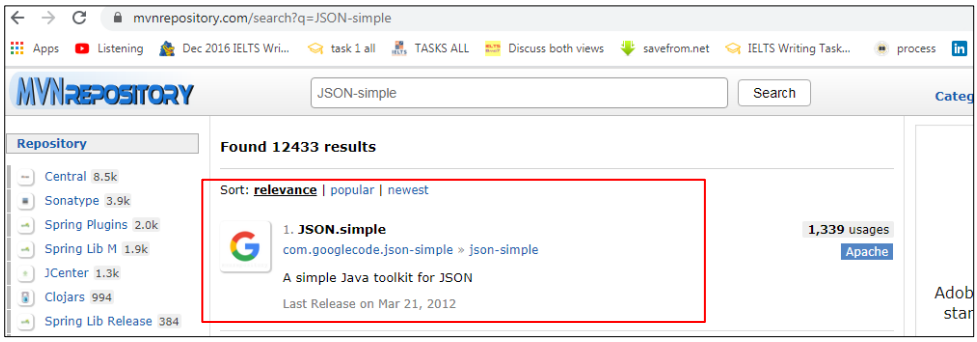
- Goto google.com and search for "maven repository"- https://mvnrepository.com/
- Search for "RestAssured for Java (click on io.rest-assured given by official rest assured



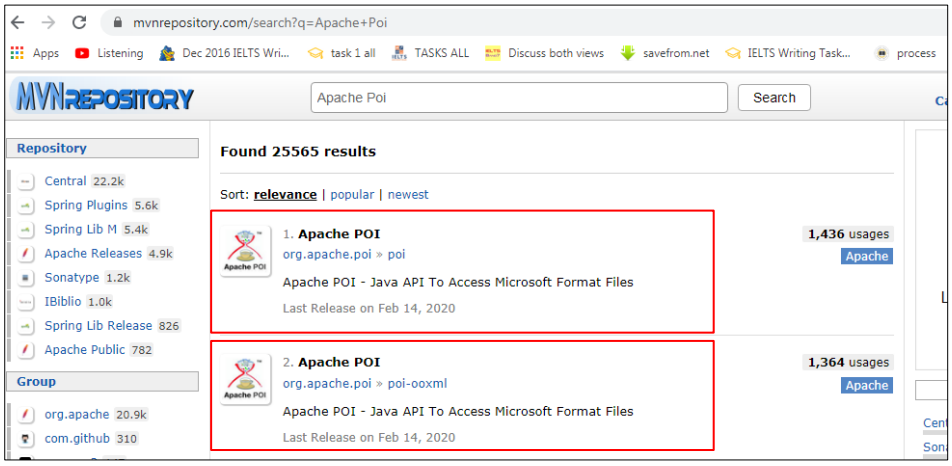
1. Click on searched REST Assured dependency as shown above
2. Copy the maven dependency of Rest Assured and paste in pom.xml under <dependencies> tag in respective pom.xml file and save file. (It will automatically download corresponding REST Assured jar files).
3. Now download the maven dependency for TestNG



4. Copy the maven dependency of TestNG and paste in pom.xml under <dependencies> tag in respective pom.xml file and save file. (It will automatically download corresponding TestNG jar files). Even if we have to include jar files of TestNG but still we need to install TestNG on our system.
5. Download the maven dependency for JSON-Simple

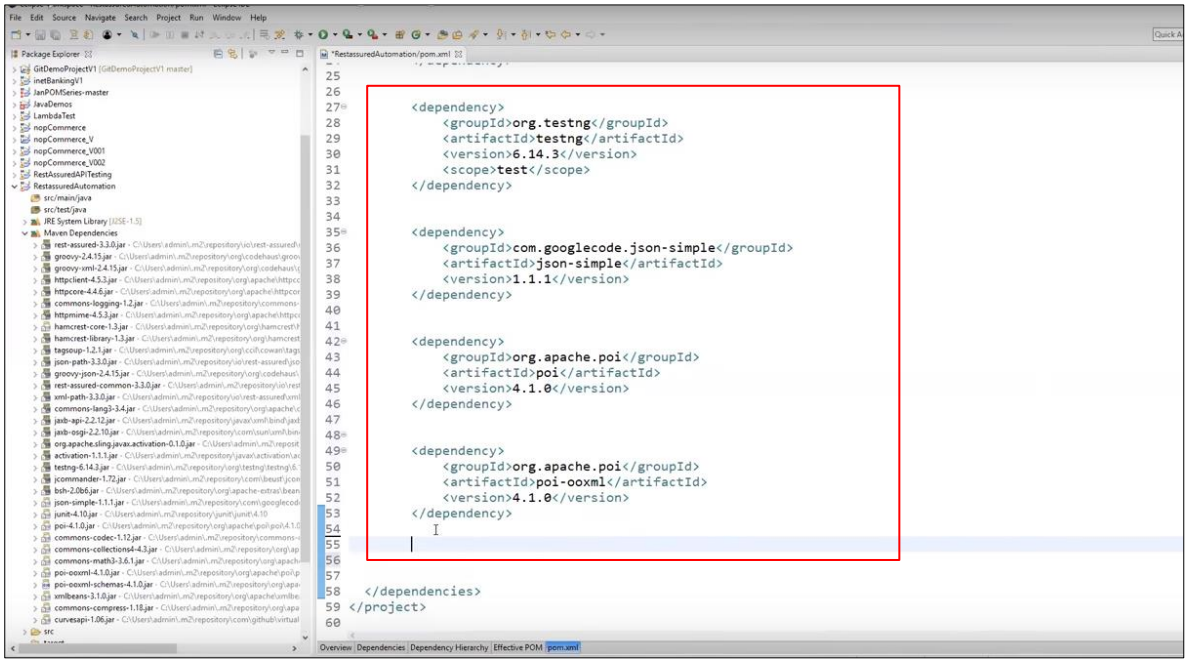


6. Copy the maven dependency of JSON-Simple and paste in pom.xml under <dependencies> tag in respective pom.xml file and save file. (It will automatically download corresponding JSON-Simple jar files).
7. Finally search for apache poi copy and paste the dependencies of apache poi (2 dependencies are there) in the dependency tag of respective pom.xml file.



(It will automatically download corresponding Apache POI jar files).

8. At last pom.xml will looks something like below:



Writing first REST Assured API Automation Program

For writing REST Assured programs we need some framework and we will be following **Behavior Driven Development (BDD) framework:**

We have following methods in BDD frame work with the help of which we will be creating our programs using REST Assured. We can broadly categorize the REST Assured program into three main section when we use BDD framework:

1. **Pre-requisite/Pre-condition section:** This section starts with **given()** method of RestAssured class and returns the object of ResquestSpecification class. Generally, we write all the pre-conditions like contentType, body etc. in this section before hitting the request.

2. **Http Method**: This section starts with the **.when()** method of RequestSpecification class and it returns the object of RequestSpecification class. It contains the http request method that we want to hit like get(), post(), put(), patch() and delete().

3. **The Ending section:** This section starts with the **.then()** method. This is the last section and we write all the validations that we want to do after the request is hit. For an instance to check whether the http request we hit in the 2nd section was success or not. We check the status of that particular hit request which are predefined and many of them are given in **appendix-2**

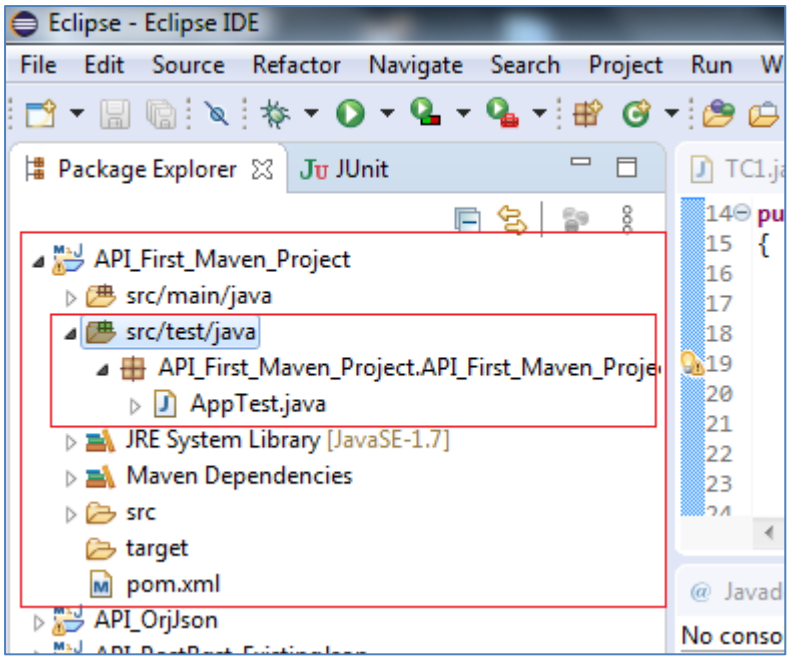
Let's start with one of the basic and simplest http request- the get() request for fetching the data. Just remember that as mentioned in the table: Chap1 Table1 we do not require any data to pass with this request and we use this request just to fetch data from the given data resource i.e. URI:

	Http Request	Description
1.	**Get**	No data is required for this request. it is used to fetch the data from the given resource location.

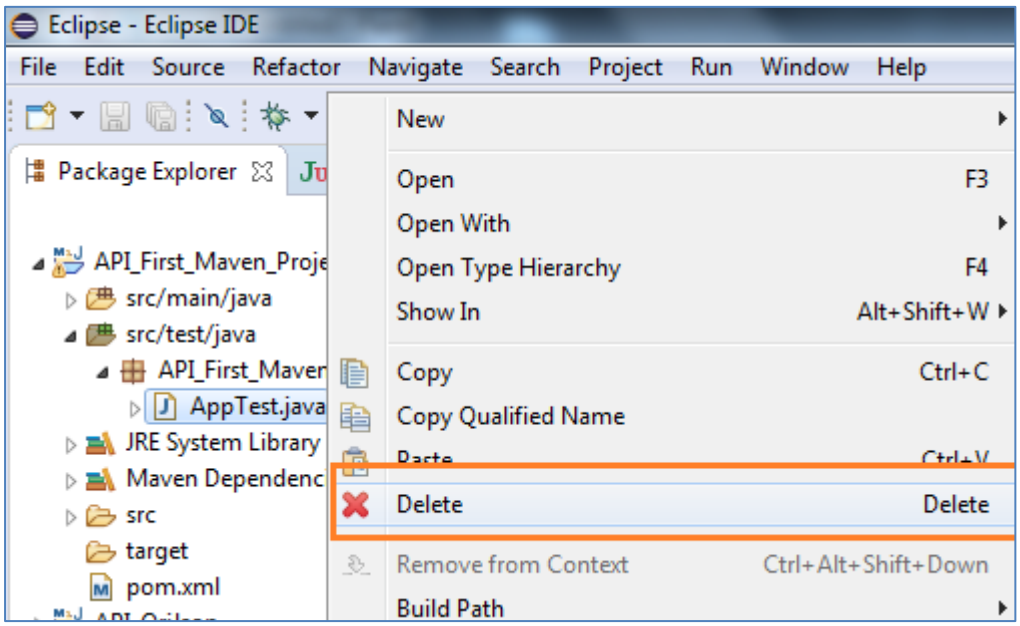
Hitting Our First http Request – The GET Request:

So far we have already created one maven project above and copied our required maven dependencies in our respective om.xml file and we have created data in a .json file (in real time scenario developers will provide this .json data file along with the URI). To create a first program follows the bellow steps:

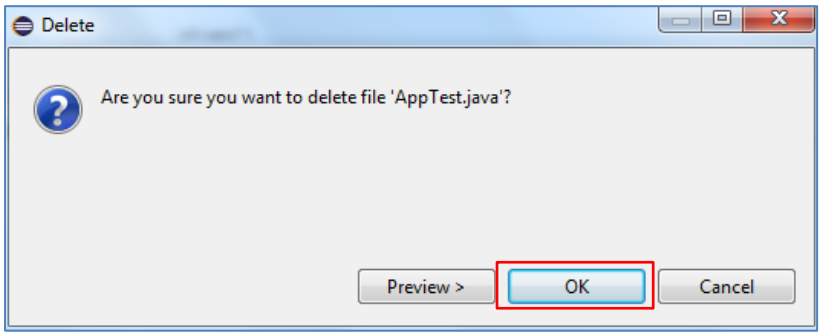
1. Click on the src/test/java package folder and click on the package API_First_Maven_Project. API_First_Maven_Project



2. Right click on AppTest.java class and delete the same. We just not to deal with this classes created by default during the creation of maven project

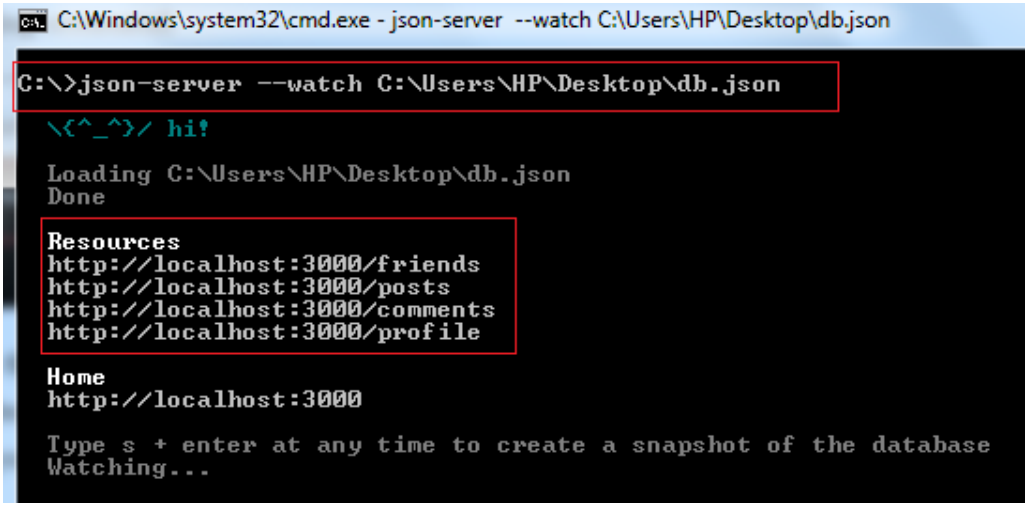


3. Simply click on Ok button



4. Write click on package API_First_Maven_Project. API_First_Maven_Project and create a new class say class1.java
5. Go to the command prompt and start the Json server by writing below command on command prompt and giving the complete path of your .json file

   C:\>json-server --watch <complete path where you have place .json data file on your system)



   and as we have seen earlier four dummy API's URI will be created by the Json server

   Let the Json server remain running and do not terminate it as we have to hit the get() http request and it will fetch the Json data from the given .json file (the one we have passed to the Json server on command prompt while starting the Json server) only and only if the Json server is up.

6. Write the following code in as shown below:
   - Import the all the ***static*** methods of RestAssured. To do this just write the first import statement as it is as shown in line no.4
   - Import com.jayway.restassured.http.ContentType as shown in line no.5
   - Import com.jayway.restassured.response.Response as shown in line no.6

As we know every java program is started with a class and hence the class name class1. We write the main method in the class and under this main method block the entire program resides and

```
public static void main(String[] args)
```

is the first statement from where the java program basically starts. We can say that it is the entry point of the program. Every program in Java is written with in a class only.

class1.java

```
1
2  // First REST Assured program using BDD framework
3  package APIProject.APIProject;
4  import static com.jayway.restassured.RestAssured.*;
5  import com.jayway.restassured.http.ContentType;
6  import com.jayway.restassured.response.Response;
7
8  public class class1
9  {
10 public static void main(String[] args)
11 {
12     Response res=
13     given()
14     .contentType(ContentType.JSON)
15     .when()
16     .get("http://localhost:3000/friends");
17
18 System.out.println("Status Code"+res.getStatusCode());
19 System.out.println("Data is");
20 System.out.println(res.asString());
21 }
22 }
23
```

In the main program res reference variable of class Response is created to store the post hit results.

```
Response res=
```

We have one statement in the given section (i.e. preconditions)

```
given()
.contentType(ContentType.JSON)
```

ContetType is the method that have an argument as type of data we are going to use and as we are using a json data so we pass the argument as contentType.JSON

when() is used to initiate the hit request that is going to follow it and to store the results in res when get request is executed

```
.when()
```

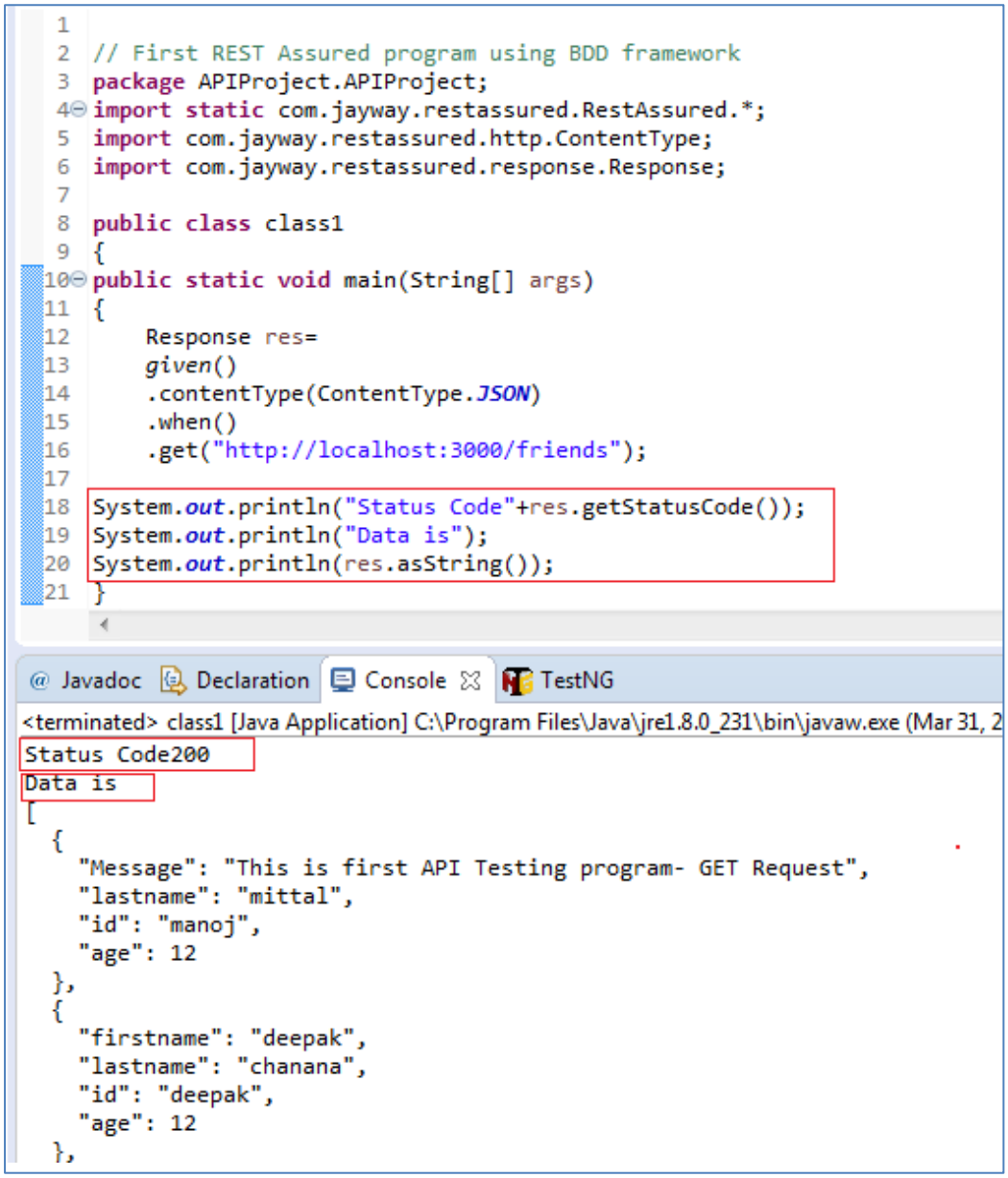
Copy the any say first URI "http://localhost:3000/friends" and pass this as an argument to get() method. HTTP get() request fetches the data from the at the given URI “http://localhost:3000/friends

```
.get("http://localhost:3000/friends");
```

And finally this is how the post request data is validated:

```
System.out.println("Status Code"+res.getStatusCode());
System.out.println("Data is");
System.out.println(res.asString());
```

7. Make sure that the Json server is up. Save and execute the program:



```
1
2  // First REST Assured program using BDD framework
3  package APIProject.APIProject;
4  import static com.jayway.restassured.RestAssured.*;
5  import com.jayway.restassured.http.ContentType;
6  import com.jayway.restassured.response.Response;
7
8  public class class1
9  {
10 public static void main(String[] args)
11 {
12     Response res=
13     given()
14     .contentType(ContentType.JSON)
15     .when()
16     .get("http://localhost:3000/friends");
17
18 System.out.println("Status Code"+res.getStatusCode());
19 System.out.println("Data is");
20 System.out.println(res.asString());
21 }
```

```
Javadoc  Declaration  Console  TestNG
<terminated> class1 [Java Application] C:\Program Files\Java\jre1.8.0_231\bin\javaw.exe (Mar 31, 2
Status Code200
Data is
[
  {
    "Message": "This is first API Testing program- GET Request",
    "lastname": "mittal",
    "id": "manoj",
    "age": 12
  },
  {
    "firstname": "deepak",
    "lastname": "chanana",
    "id": "deepak",
    "age": 12
  },
```

Finally, our first API automation program runs successfully and it has fetched data of type json with get() request from the .json file.

Note the status code of get request is 200 which is correct and we have fetched the post hit get() request status code by writing below statement:

```
System.out.println("Status Code"+res.getStatusCode());
```

And we have fetched the data with the statements:

```
System.out.println("Data is");
System.out.println(res.asString());
```

Note: The output data in your get() request will fetch different data as displayed here. It will fetch the data that you have maintained in your .json file in Json format and you pass there in the command prompt while starting the Json server.

Chapter-4: The Post Request – Body Data Creation Ways

We have already seen in the **chaper1 table 1** that in order to hit the Post http request we have to have some data which we want to post. Http get() request is used to fetch (read) the data from the resource (URI, .json file) and the Post request simply means to write the data on the given resource (URI, .json file).

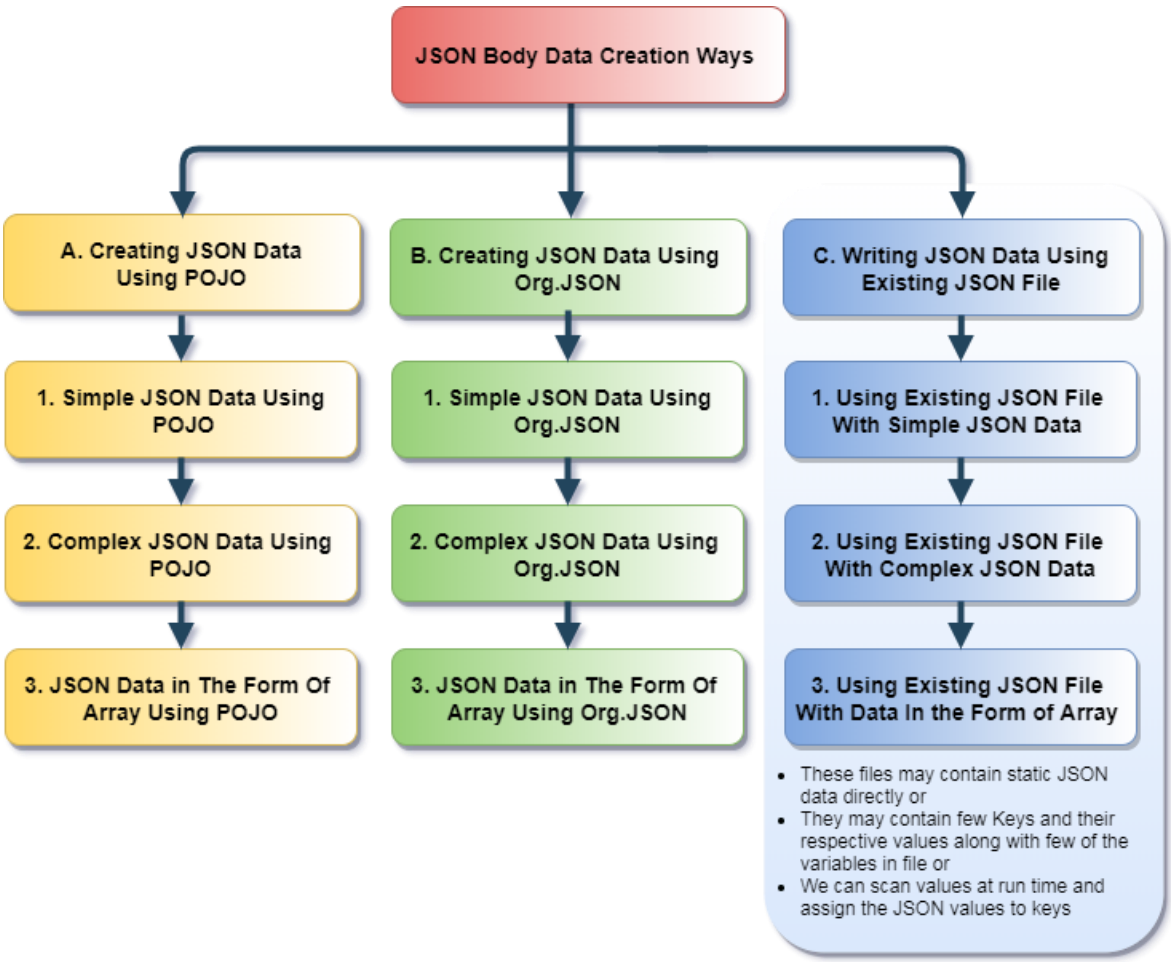
Http Request	Description
Get	No data is required for this request. it is used to fetch the data from the given resource location.
Post	Data is required for this kind of request. It is used to insert the data at the given resource location.

Let's learn first the types of body data creation ways and then we will be covering each of them one by one proceeded by using them in Http Post request.

There are three ways to create the body data

We can create body data using:

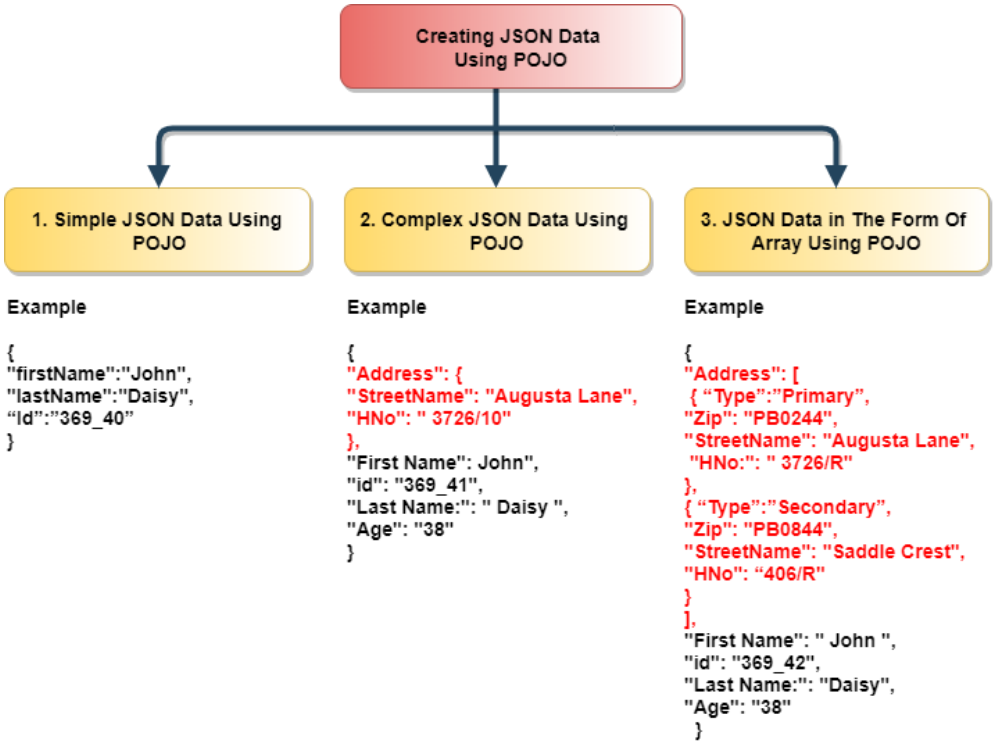
1. Java known as **P**lain **O**ld **J**ava **O**bject (POJO)
2. Using **O**rg.**J**SON
3. Using already **E**xisting **J**son **F**ile



Chapter-4 Diagram 1

Body Data Creation Using Plain Old Java Object (POJO): We can easily create the data for the .body() to pass in the hit request and further, we can create data using POJO in three different ways:

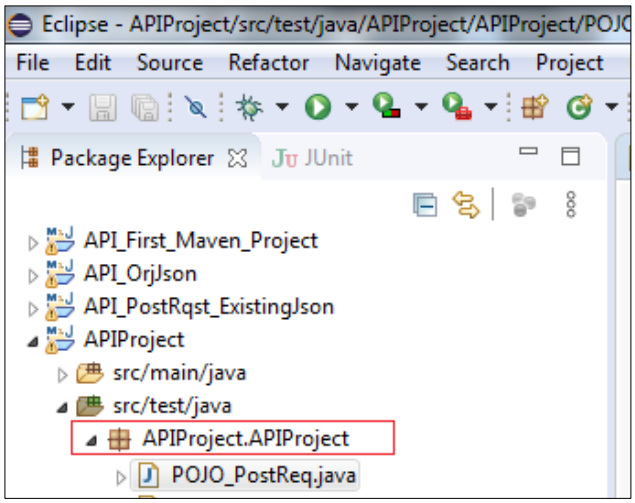
Note: *In Pojo we use getter and setter to assign the values. Simply remember Pojo means use of getter – setter*



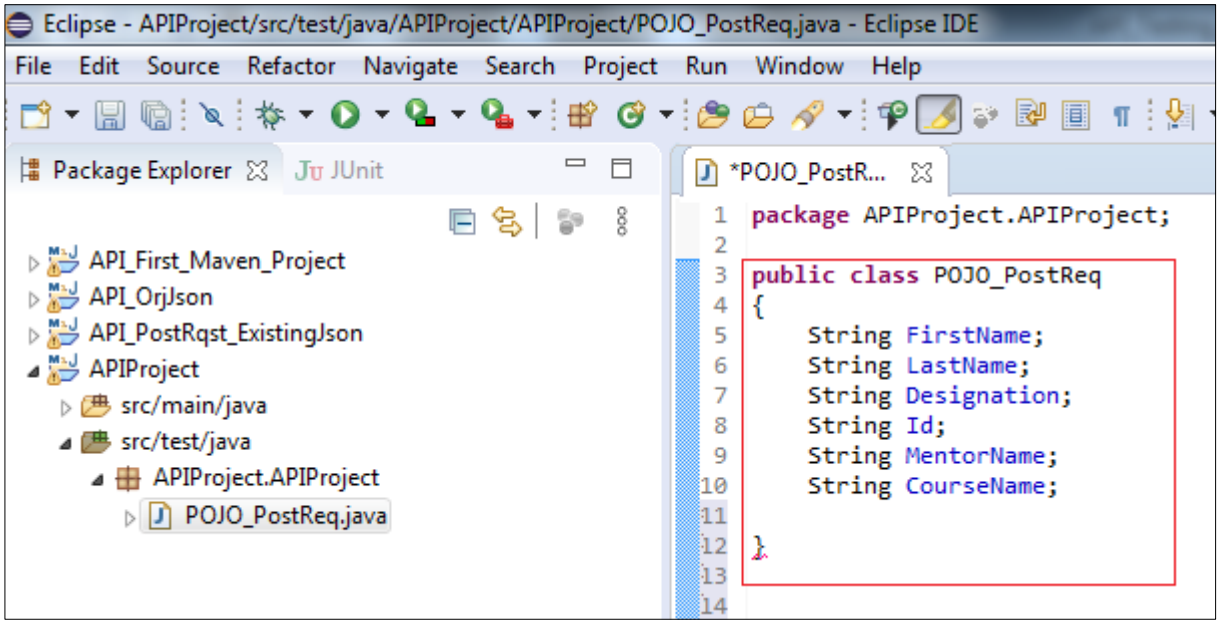
Chapter-4 Diagram 2

1. **Simple JSON Using POJO:** Simple JSON data is created in the form of key:value as below:

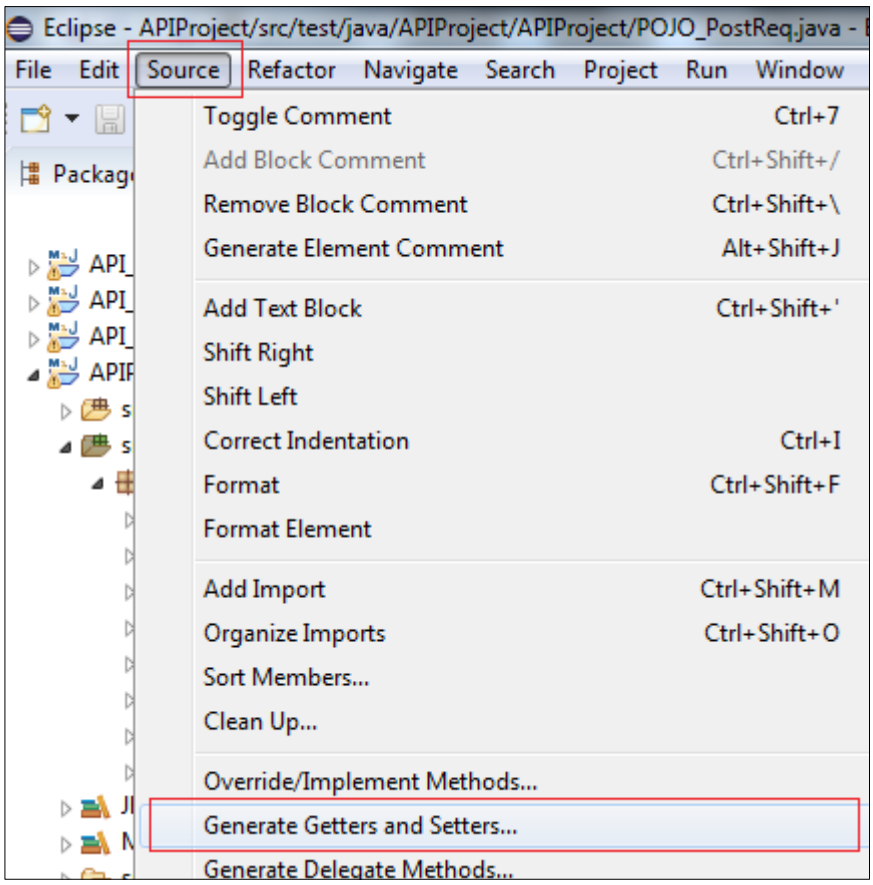
1. Create one package say APIProject and create one class say POJO_PostReq.java (though you can give any name) as shown below:



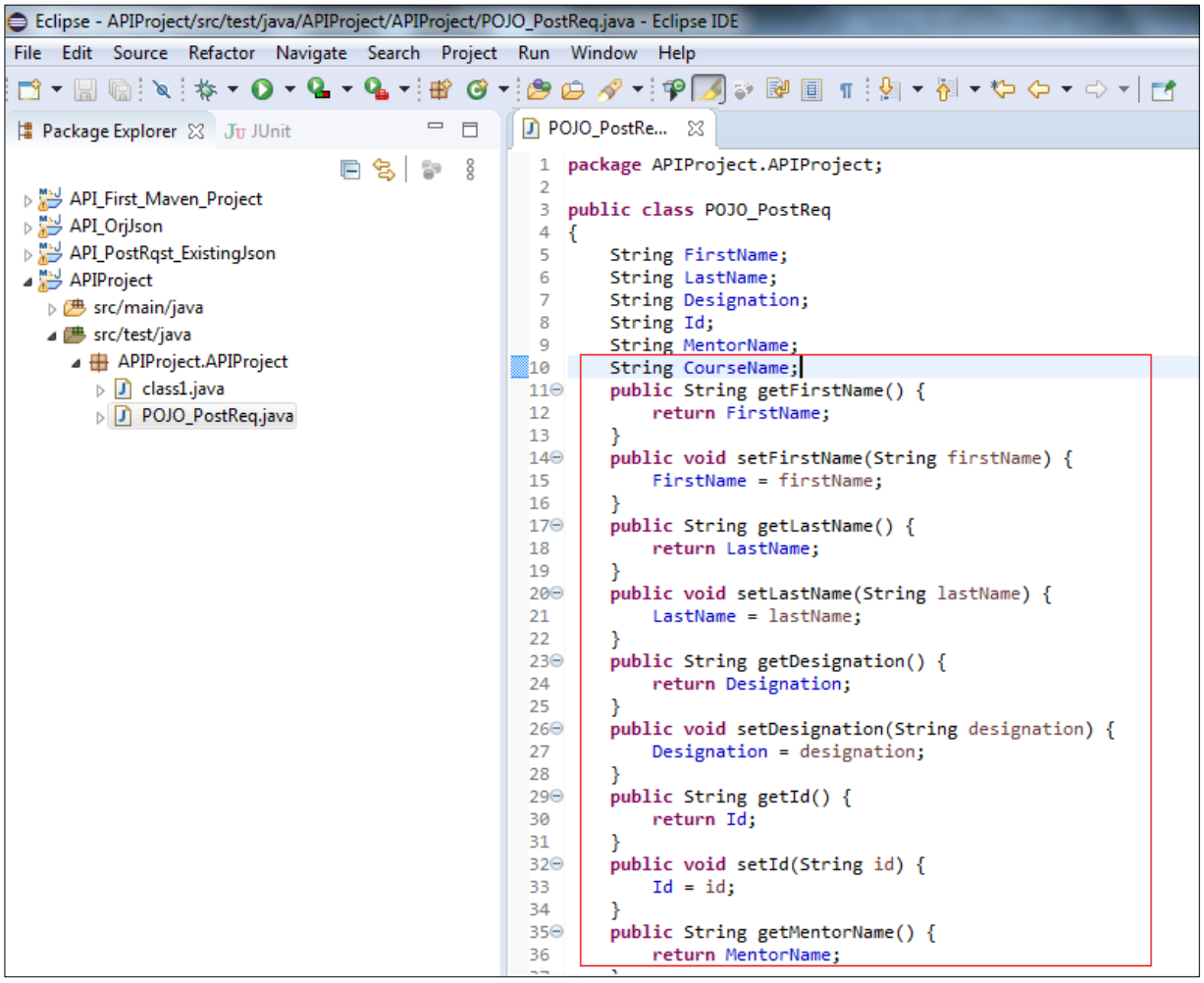
2. Declare variables as required in POJO_PostReq as shown as in the screen print below and this will further hold the data that is to posted using the Post Request



3. Click on Source menu option and click Generate Getter and Setter. Select all the check boxes corresponding to each of these variables and press enter.

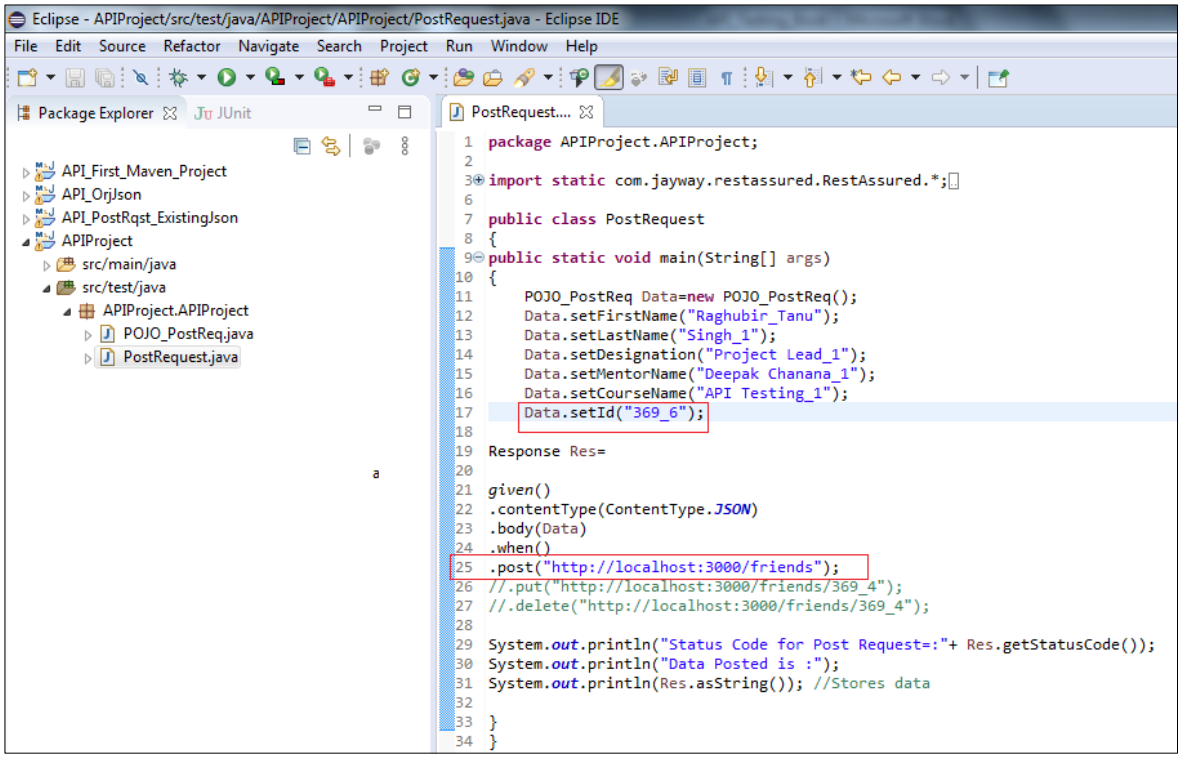


4. After generating Getter and Setter it will look like the one as shown below:

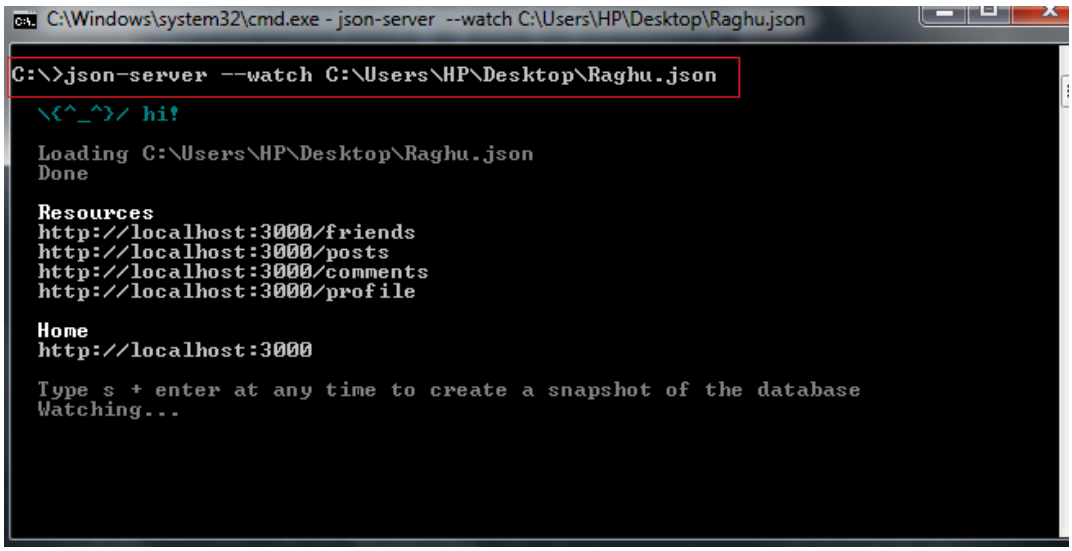


5. Create another class PostRequest.java under the same package. Declare the object of class POJO_PostReq and set the values of variables declared in that class. Finally, pass the object in the body of Post request as shown below:

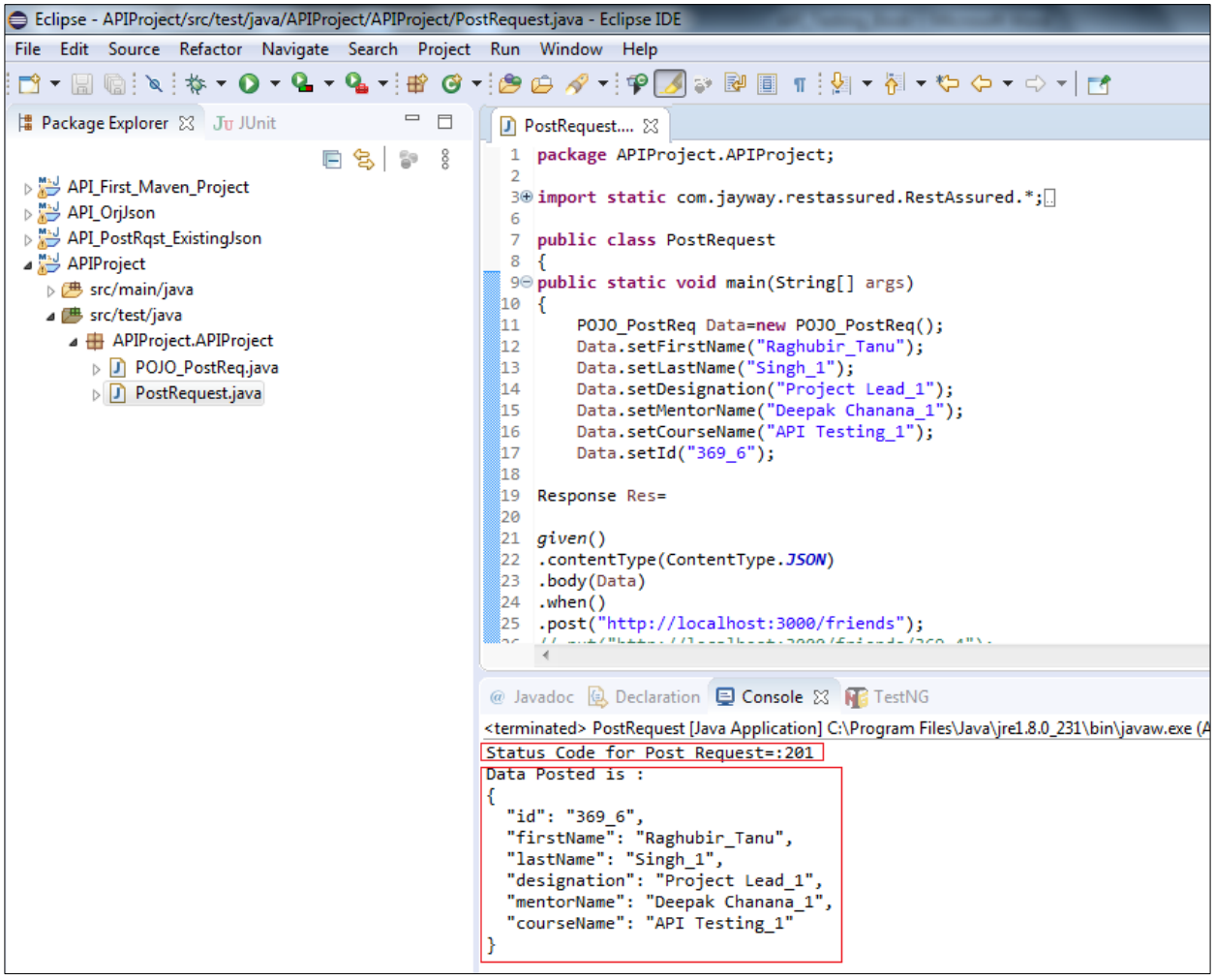
   Data.setFristName("Raghubir_Tanu") instruction is used to set the value of various declared variables under the respective class.



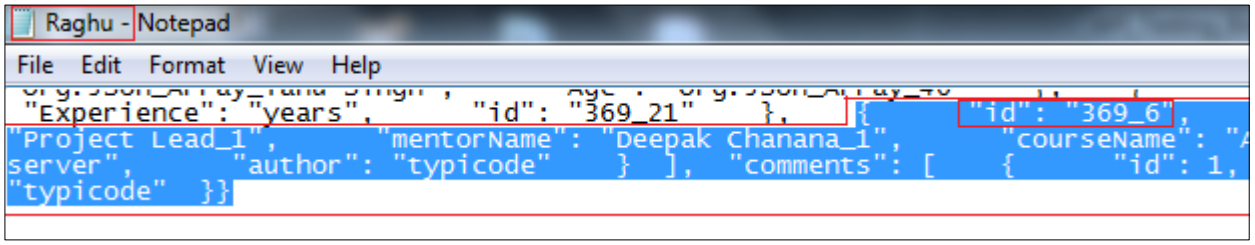
6. Start the Json Server



7. Run the program and it is very much expected that the data in the form of Complex json should be posted.

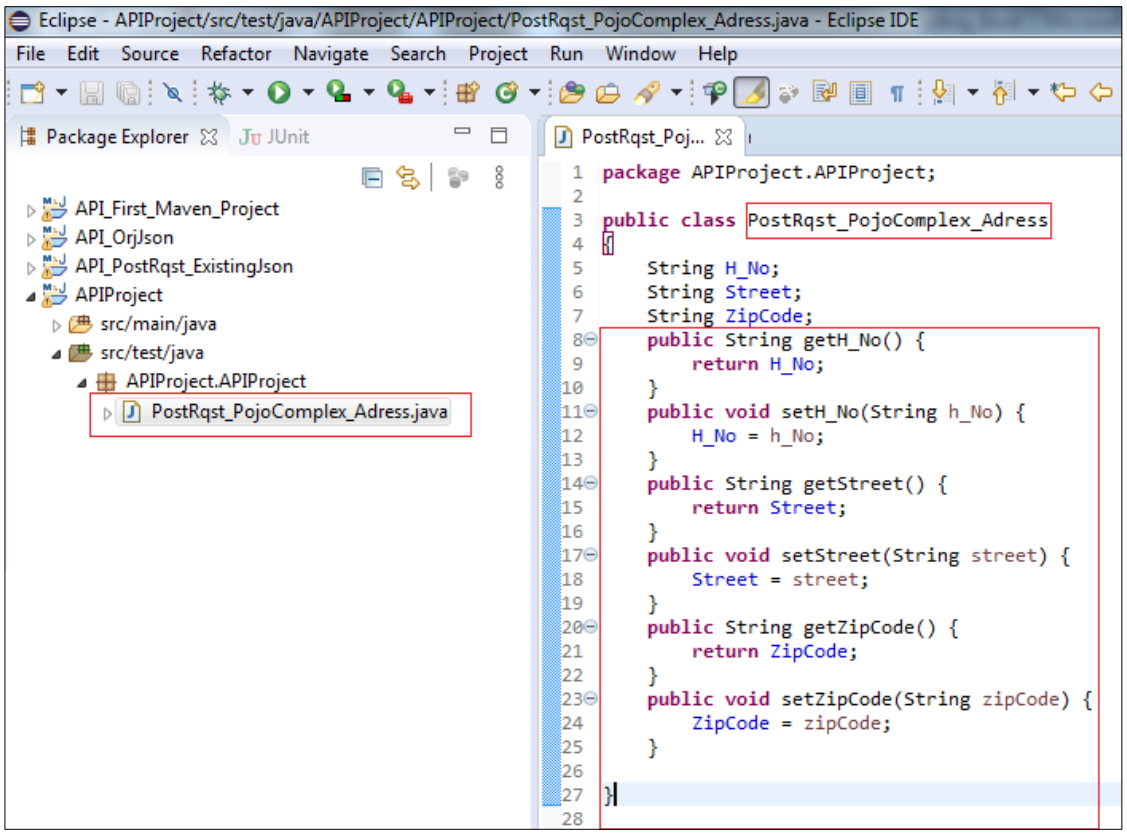


8. Open the target .Json file in the Notepad and search for the data posted



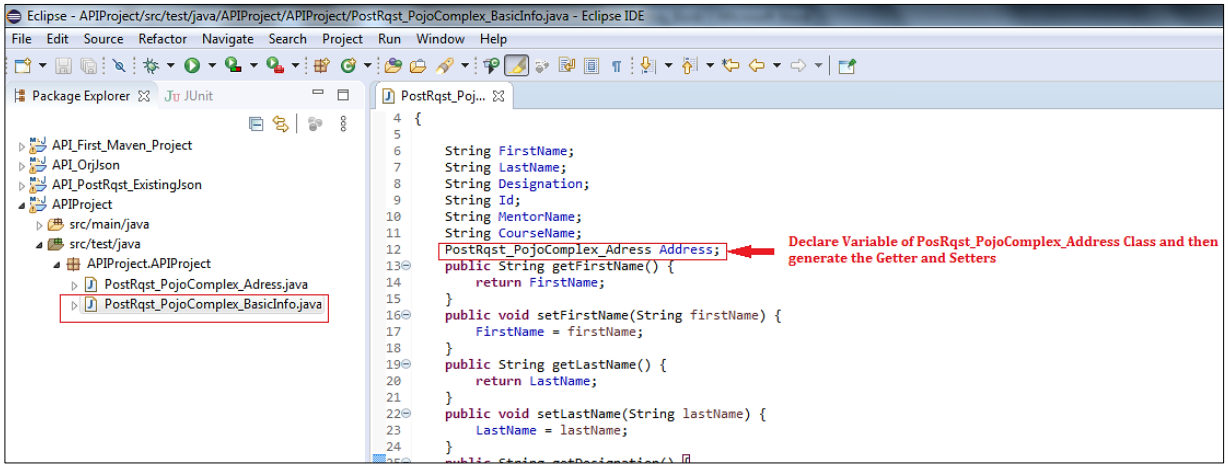
2. **Complex Json Using POJO:** Complex Json is *'Json within a Json'* or we can say nested Json and Body Data in Complex Json is created as below:

1. Create one class say PostRqst_PojoComplex_Address.java and declare some variables and again Generate Getter and Setter as done in the case of Simple Json using POJO above and as shown below:



2. Create another Class say PostRqst_PojoComplex_BasicInfo.java and declare some variables according to the data that you want to post in the form of Complex Json using POJO and then ***declare the object of Class PosRqst_PojoComplex_Address that has been created in the Step-1 above.*** Now generate the getter and setters as well as shown below:

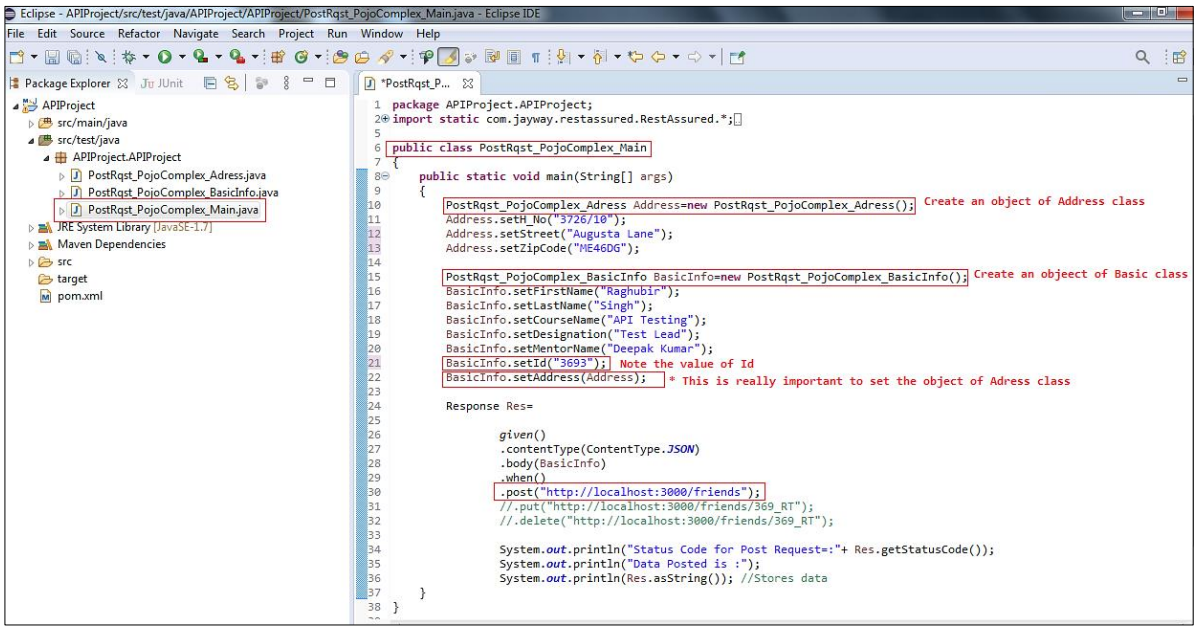
Very Important Note: *Just Declare the Object of class that you want to nest. In our case its PosRqst_PojoComplex_Address so we have declared the Object of this class while declaring the information of PostRqst_PojoComplex_BasicInfo class.*



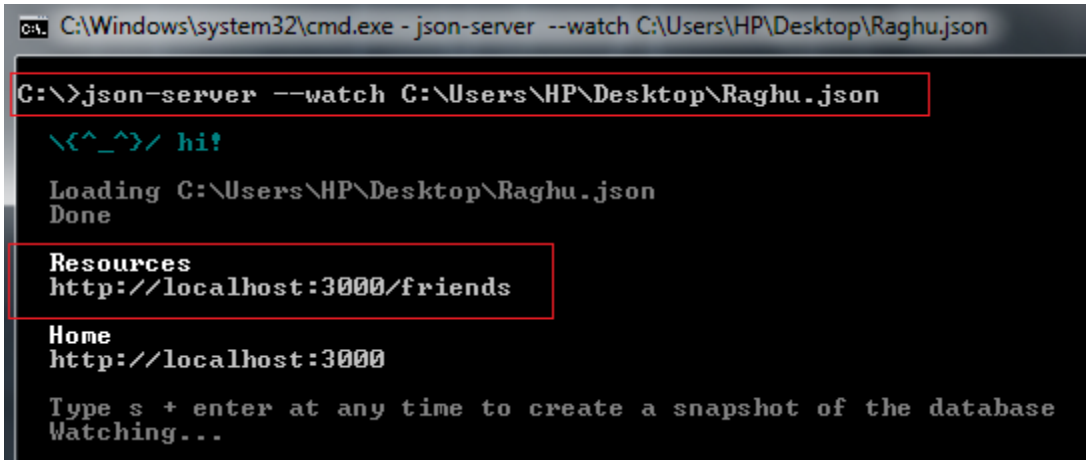
3. Create the main class say ***PostRqst_PojoComplex_Main.java*** and create the object of class ***PosRqst_PojoComplex_Address*** and set the value of variables declared in this class as shown below:

   Then, create an object of class ***PostRqst_PojoComplex_BasicInfo*** and set the values of variables declared earlier in this class as shown below:

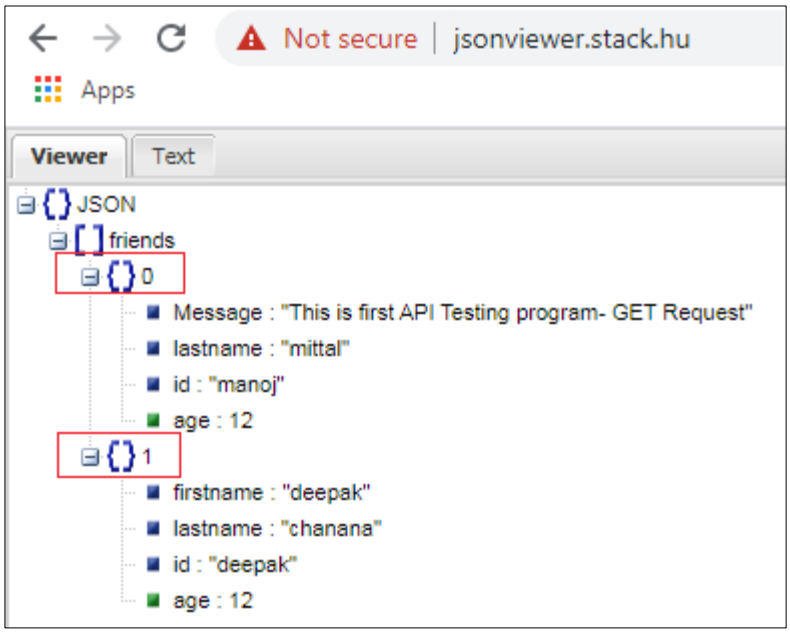
   Very Important Note: *Just set the Object of class PosRqst_PojoComplex_Address which we have declared earlier in the information of PostRqst_PojoComplex_BasicInfo class.*



4. Start the Json Server and the resource URI's will be created as per the structure of the file:

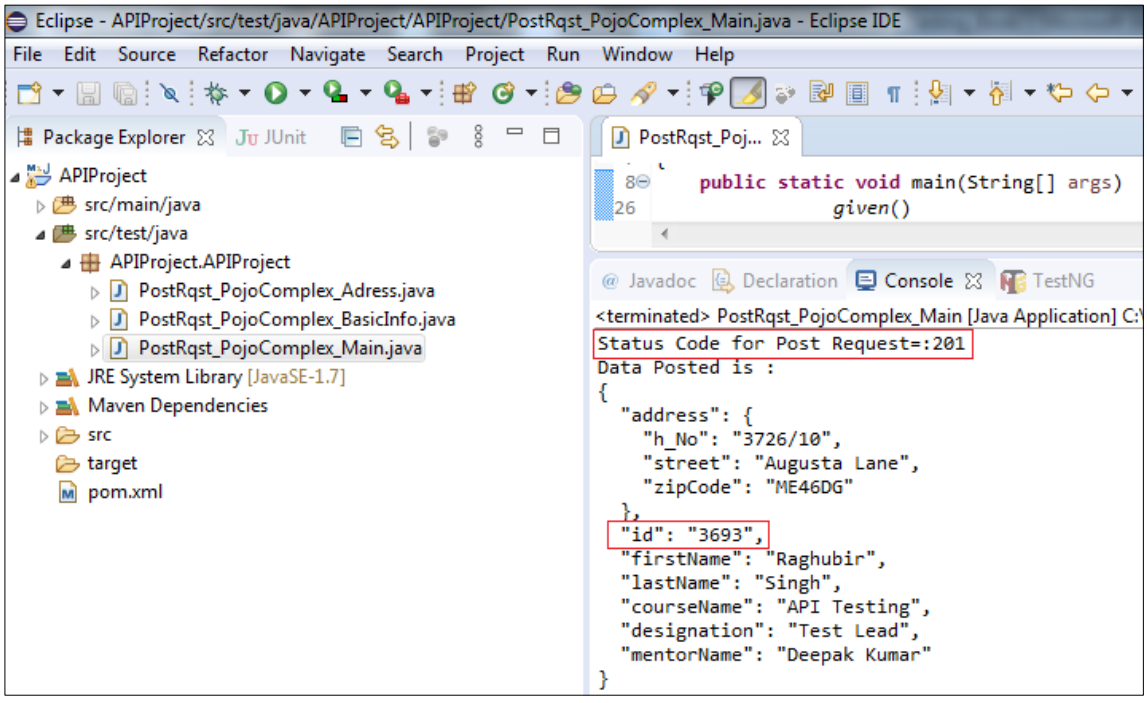


5. Before we proceed to hit the Post request let's see the data we do have in the target file by opening the .json file in Notepad and copy the entire data from the file and paste in the Text tab of online json viewer tool https://jsonviewer.stack.hu

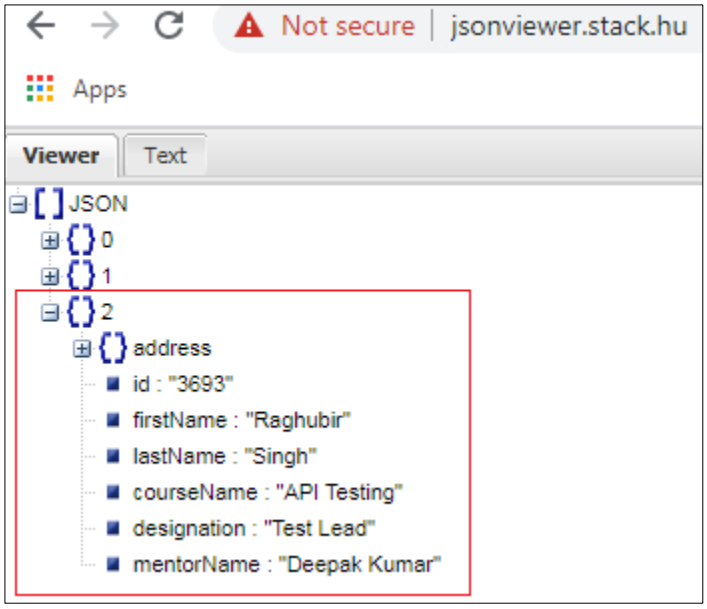


So as we can see that there are only two objects {}0 and {}1 under the Array friends.

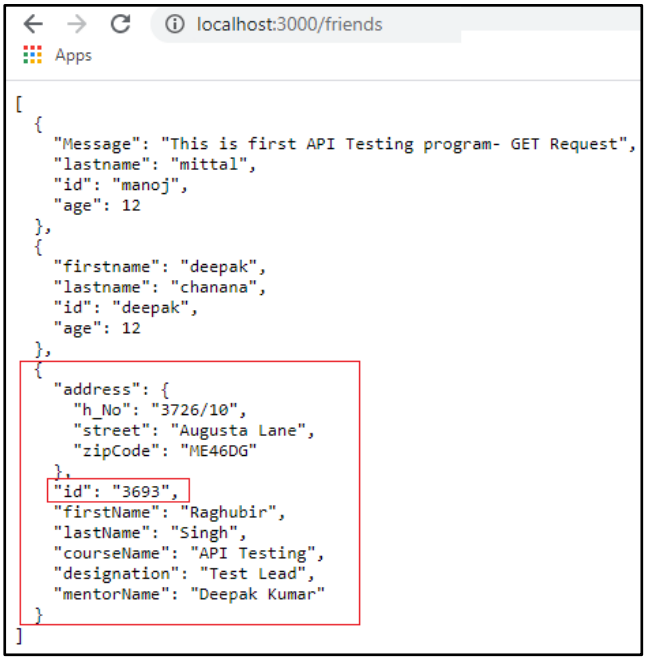
6. Run the program and verify the results in the output window. It is to be notice that the data in the form of Complex Json should be posted.



7. Verify the results into the targeted Json file also. Open the file in Notepad and verify the data in the form of Complex Json posted by the program above or copy whole of the data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and then view the data in the viewer tab:

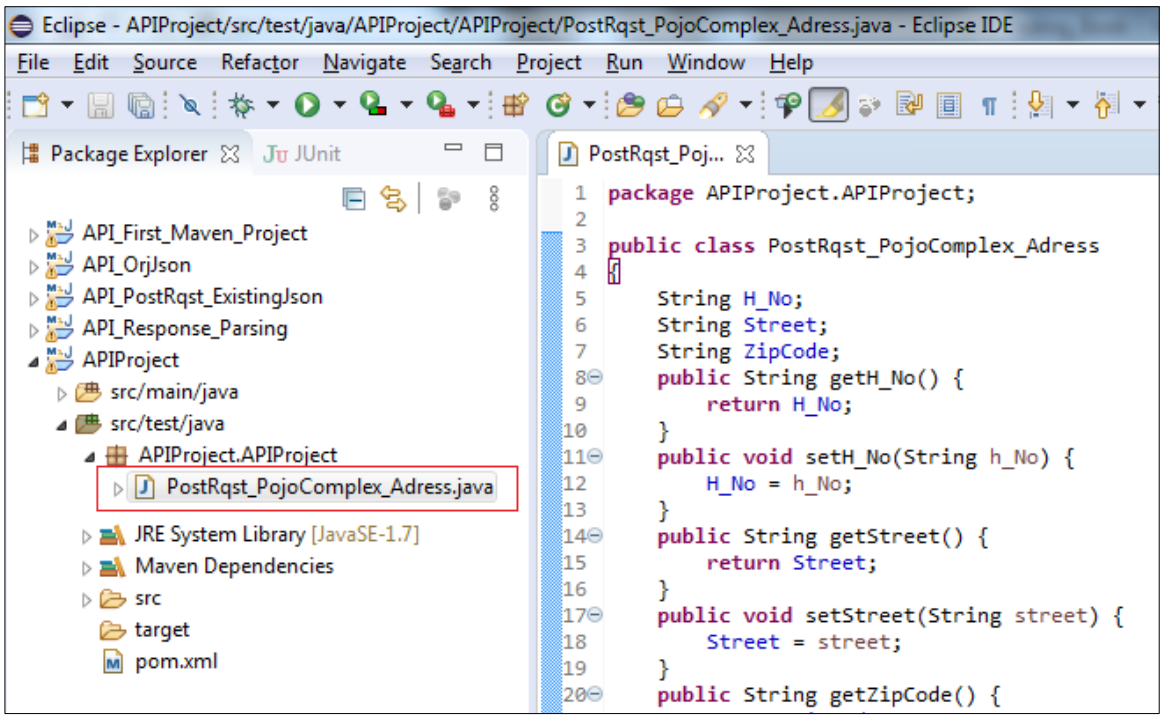


8. We can also copy our URI and paste it in the bowser to check our post request as shown below:

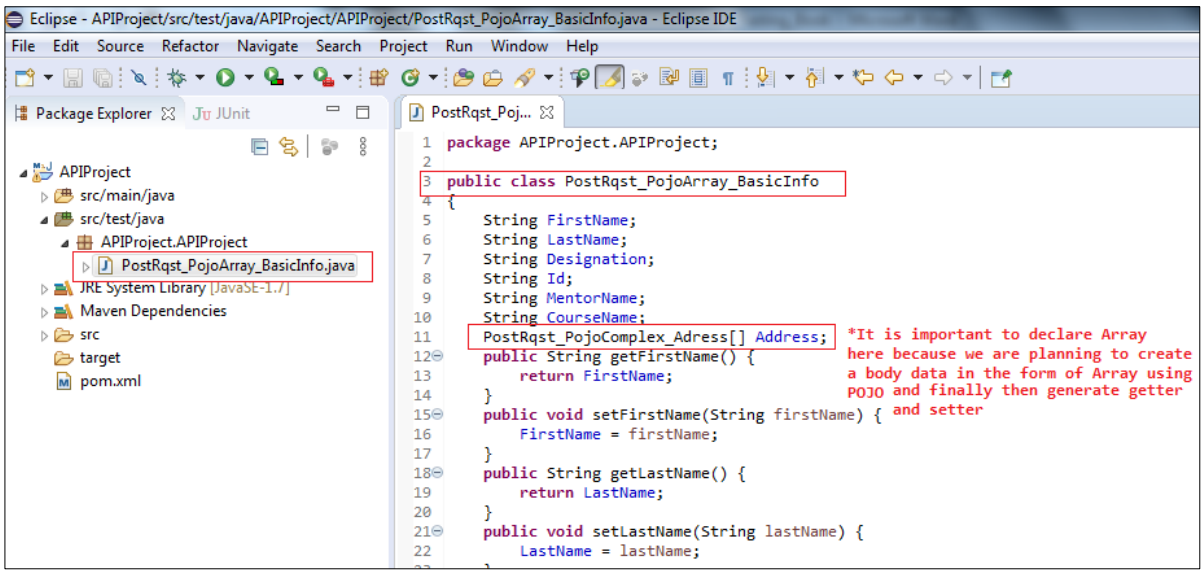


3. **Body Creation In The Form Of Array using POJO:** Body data creation in the form of array is created using the array notation as shown below:

1. Create one class say PostRqst_PojoComplex_Address.java (although you can give any name to your class). Declare some variables and generate getter and setter as we did in our previous programs and shown below:

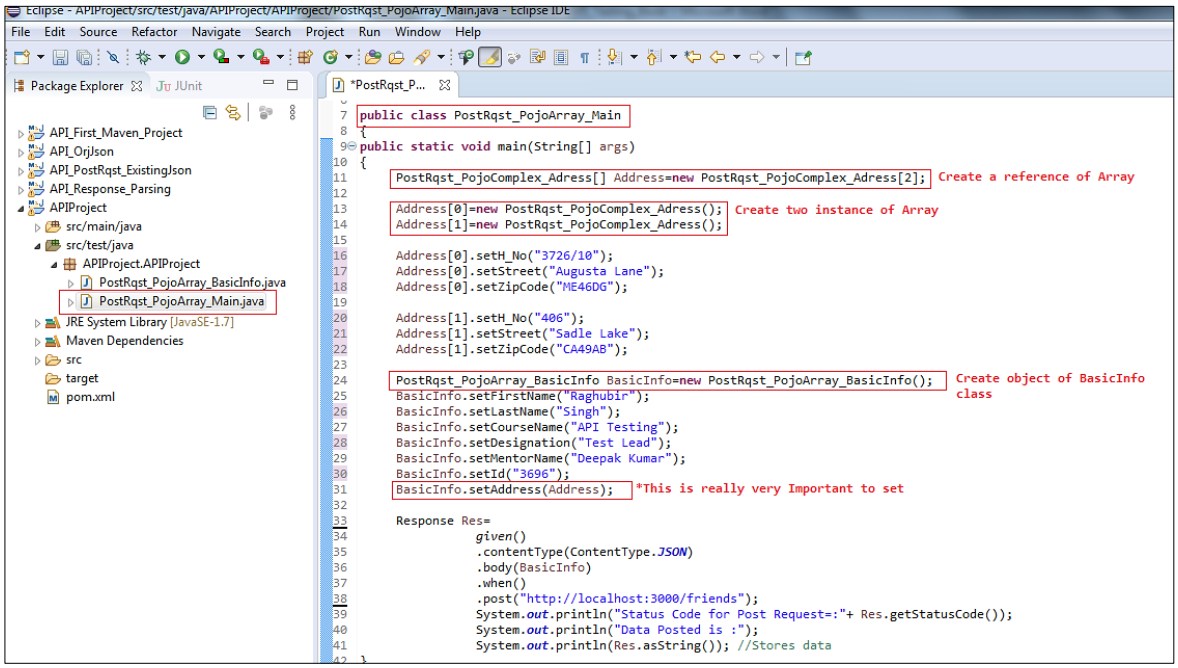


2. Create another class say PostRqst_PojoArray_BasicInfo.java and declare some variables and Array object of the class. Declare Array of object and Generate Getter and Setter as done in the case of Simple Json using POJO above and as shown below:

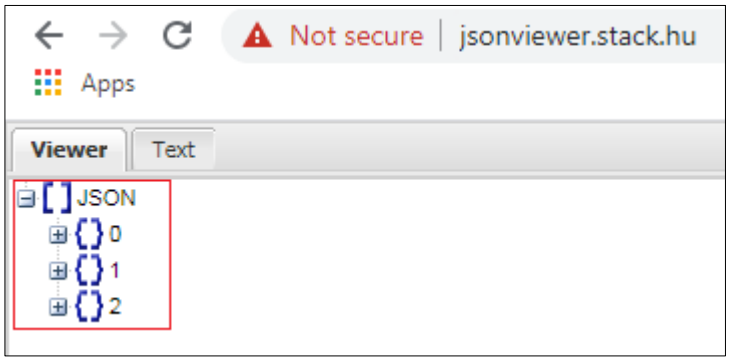


3. Create the main class say ***PostRqst_PojoArray_Main.java*** and create the Array object of class ***PosRqst_PojoComplex_Address.*** Create two instance of Array and set the values of both of the array objects. Create the object of the BasicInfo class and set its values too as shown below:

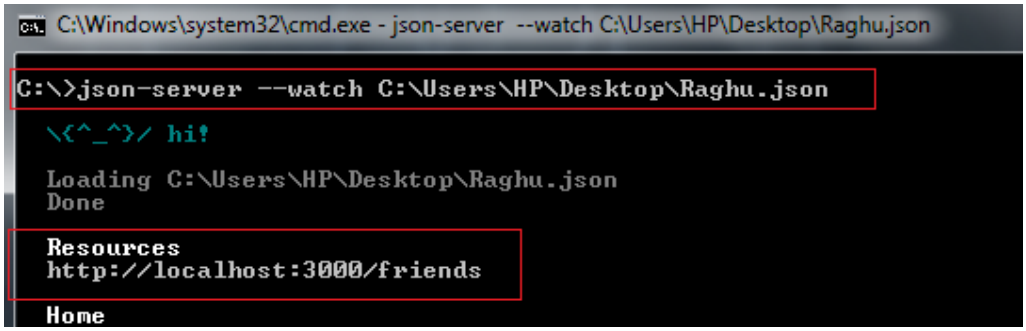
   Very Important Note: *Just set the Array of object also as shown in the Screenshot below the statement marked with *symbol.*



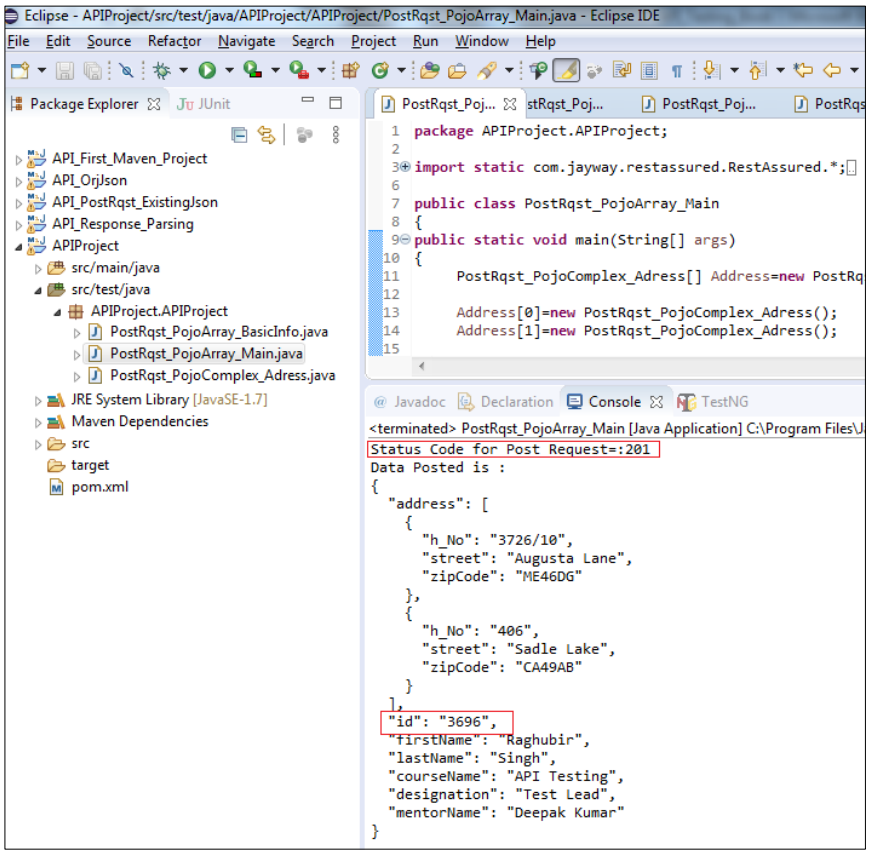
4. Before we proceed to hit the Post request let's see the data we do have in the target file by opening the .json file in Notepad and copy the entire data from the file and paste in the Text tab of online json viewer tool https://jsonviewer.stack.hu
   So as we can see that there is only three objects {}0, {}1 and {}2 under the Array friends.



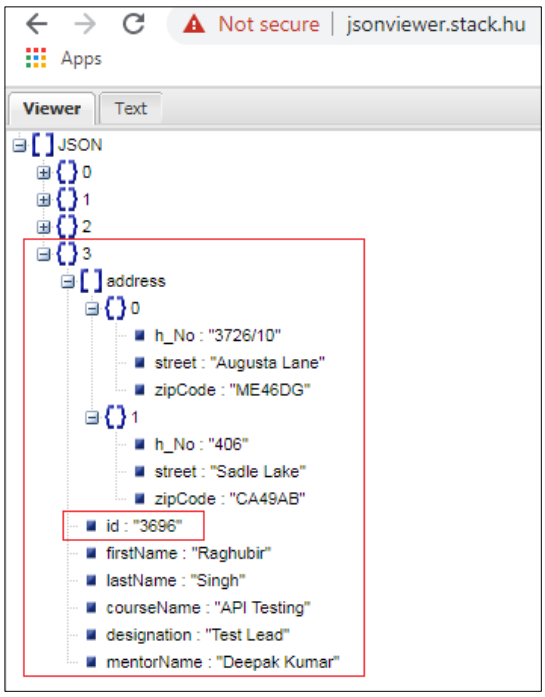
5. Start the Json Server and the resource URI's will be created as per the structure of the file:



6. Run the program and verify the results in the output window. It is expected that the data in the form of Array should be posted the provided URI and in the target .json file.



7. Verify the results into the targeted Json file also. Open the file in Notepad and verify the data in the form of Complex Json posted by the program above or copy whole of the data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and then view the data in the viewer tab:



8. We can also copy our URI and paste it in the bowser to check our post request as shown below:

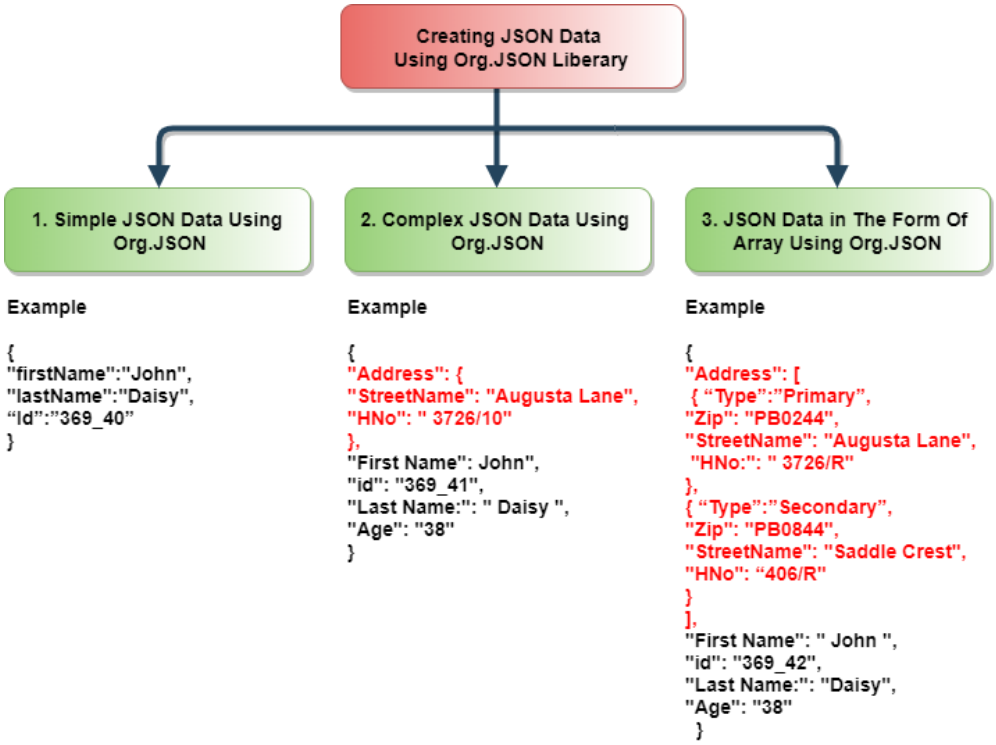
Chapter- 5: Body Data Creation using Org.Json Library

Body Data Creation Using Org.Json: First of all we need a corresponding jar file that corresponds to the org.json which we will copy from the site: https://mvnrepository.com/ by searching "org.json" as we did earlier while including the other dependencies in our maven project.

```
<!-- https://mvnrepository.com/artifact/org.json/json -->
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20190722</version>
</dependency>
```

Once you copy the same, just paste the same in the respective pom.xml file under <dependencies> tag and save the pom.xml file.

Further, we can create data using Org.Json in three different ways:



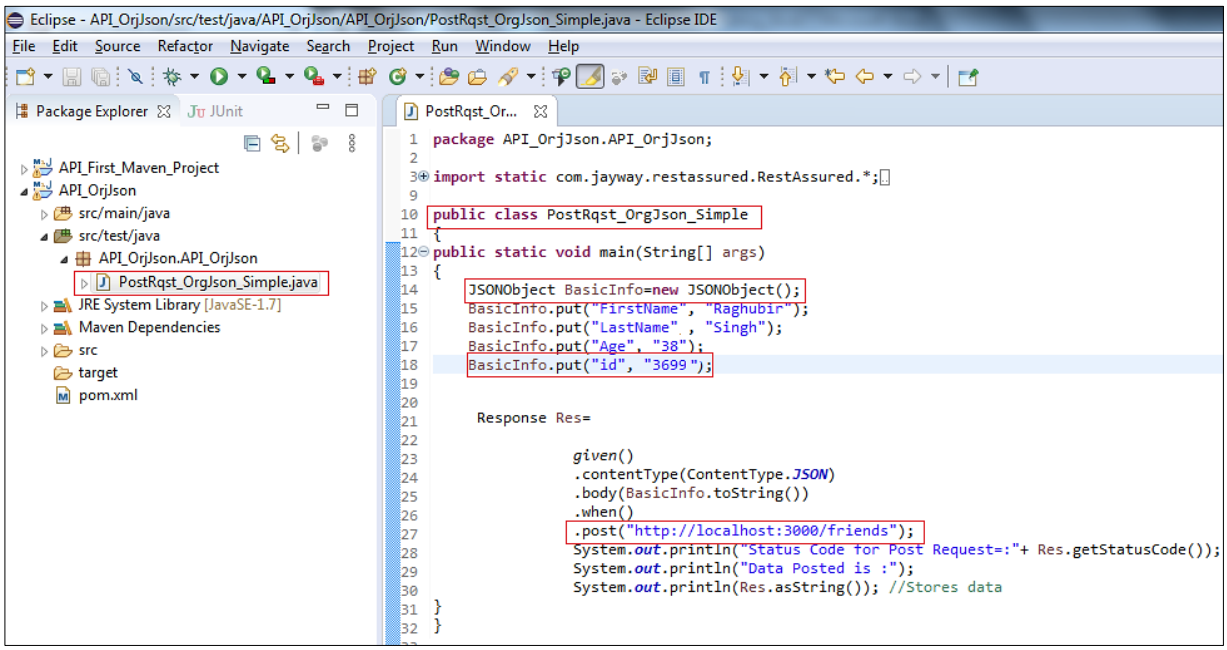
Chapter-5 Diagram 1

1. **Simple JSON Using Org.Json Library :** For creating body data using Org.Json liberary create one maven project say API_Orgjson and copy the required maven dependency like Rest Assured and Og.json in the respective pom.xml file.

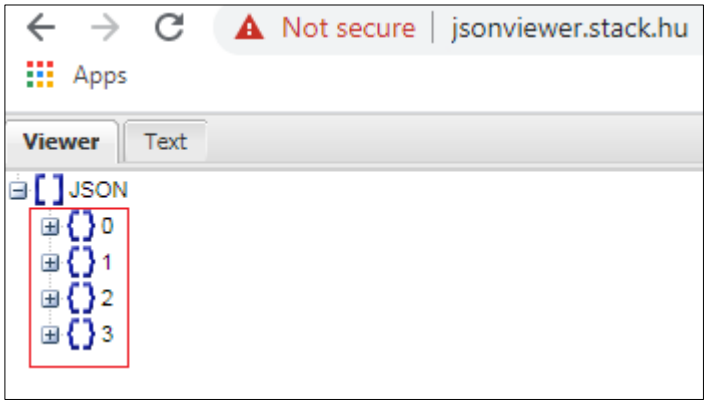
1. Create one class sat API_Orgjson and under the package API_Rqst_OrgJson_Simple.java
2. Import all the static methods of RestAssured
3. For creating the body data with Org.json we need to create an object of JSONObject class and we have created BasicInfo and we will be using object of the JSONObject class created along with .put method to assign the values to the keys as shown below:

Very Important Note: *It is to be noted here that we have to convert the object of the JSONObject class to string and then only it has be passed in the body method.*

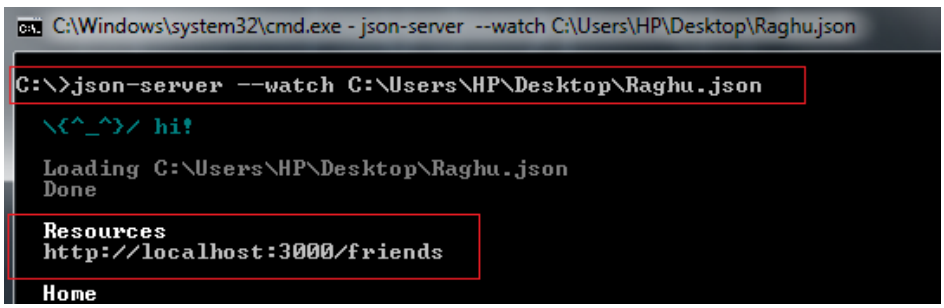
Note: *We will be using put() method to assign the values. Please donot confuse it with PUT Http request. Both of these are different.*



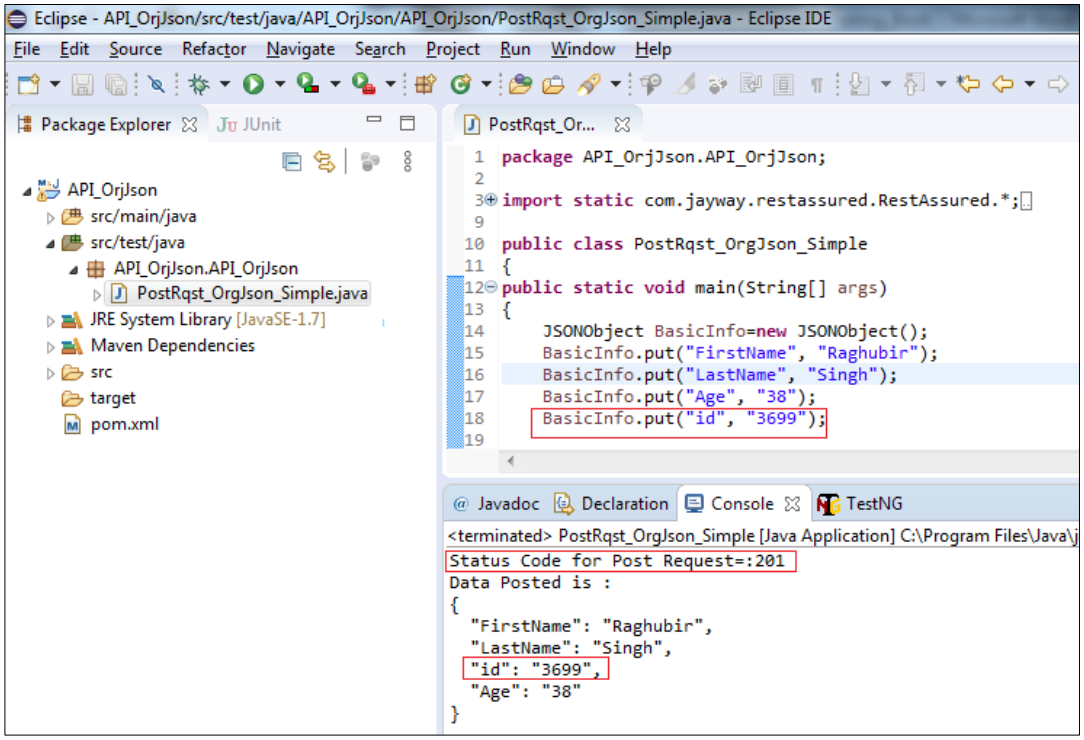
4. Not the existing data in the targeted Json file. Open the file in Notepad and note the data in the present in the Json file. Copy whole data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and view the json in the viewer tab:



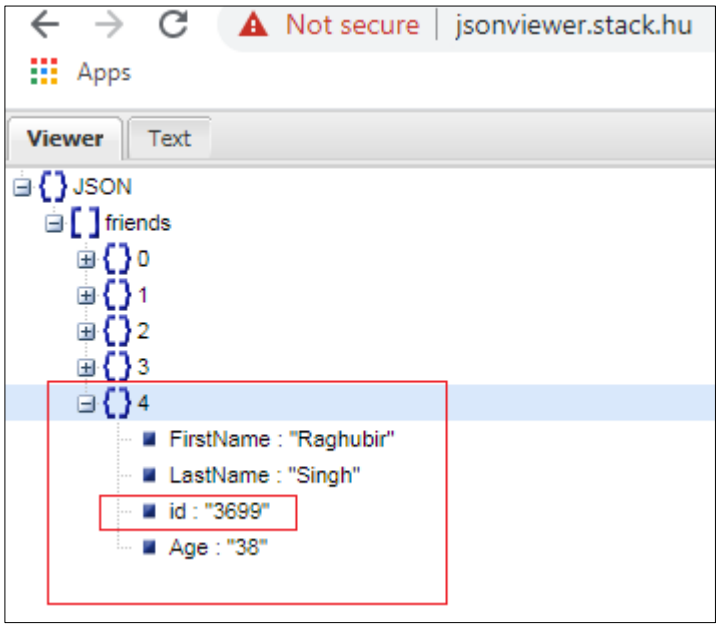
5. Start the Json Server and the resource URI's will be created as per the structure of the file:



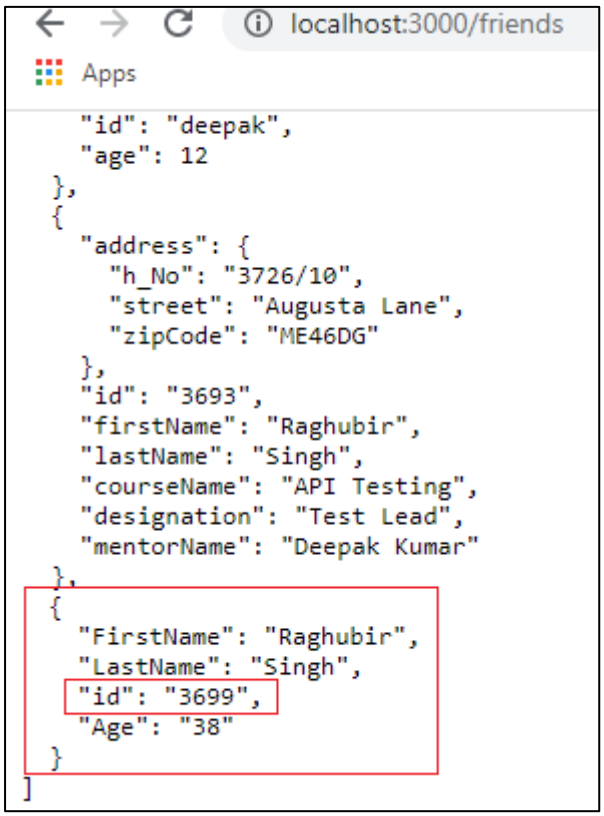
6. Run the program and verify the results in the output window. It is to be notice that the data in the form of Simple Json should be posted the provided URI and in the target .json file.



7. Verify the results into the targeted Json file also. Open the file in Notepad and verify the data in the form of Complex Json posted by the program above or copy whole of the data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and then view the data in the viewer tab:

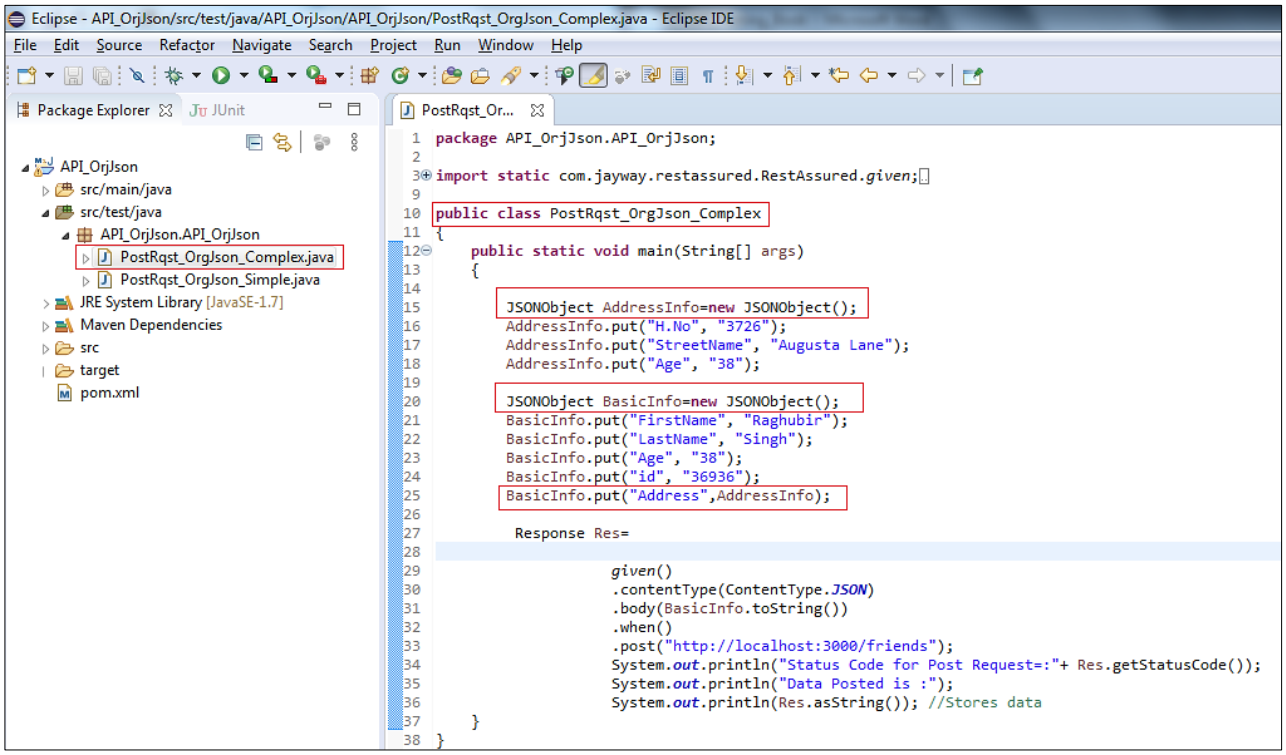


8. We can also copy our URI and paste it in the bowser to check our post request as shown below:

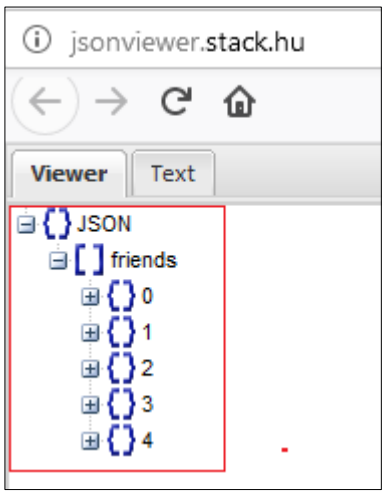


2. **Complex Json Using Org.Json:** Complex json is json with in a json or we can say nested json and Body Data in Complex Json is created as below:

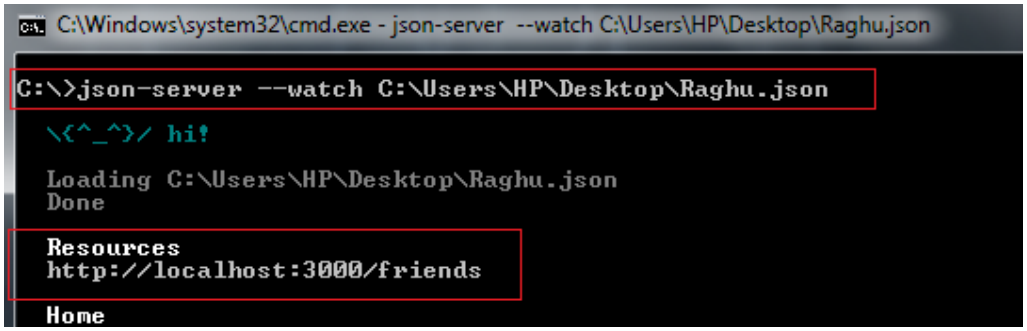
1. Create one class say API_Orgjson and under the package API_Rqst_OrgJson_Simple.java
2. Import all the static methods of RestAssured
3. For creating the body data in the form complex format with Org.json we need to create an object of JSONObject class and for that we have created AddressInfo object. This object of the JSONObject class created is used along with .put method to assign the values to the keys. We have to create another object of JSONObject class as shown below:



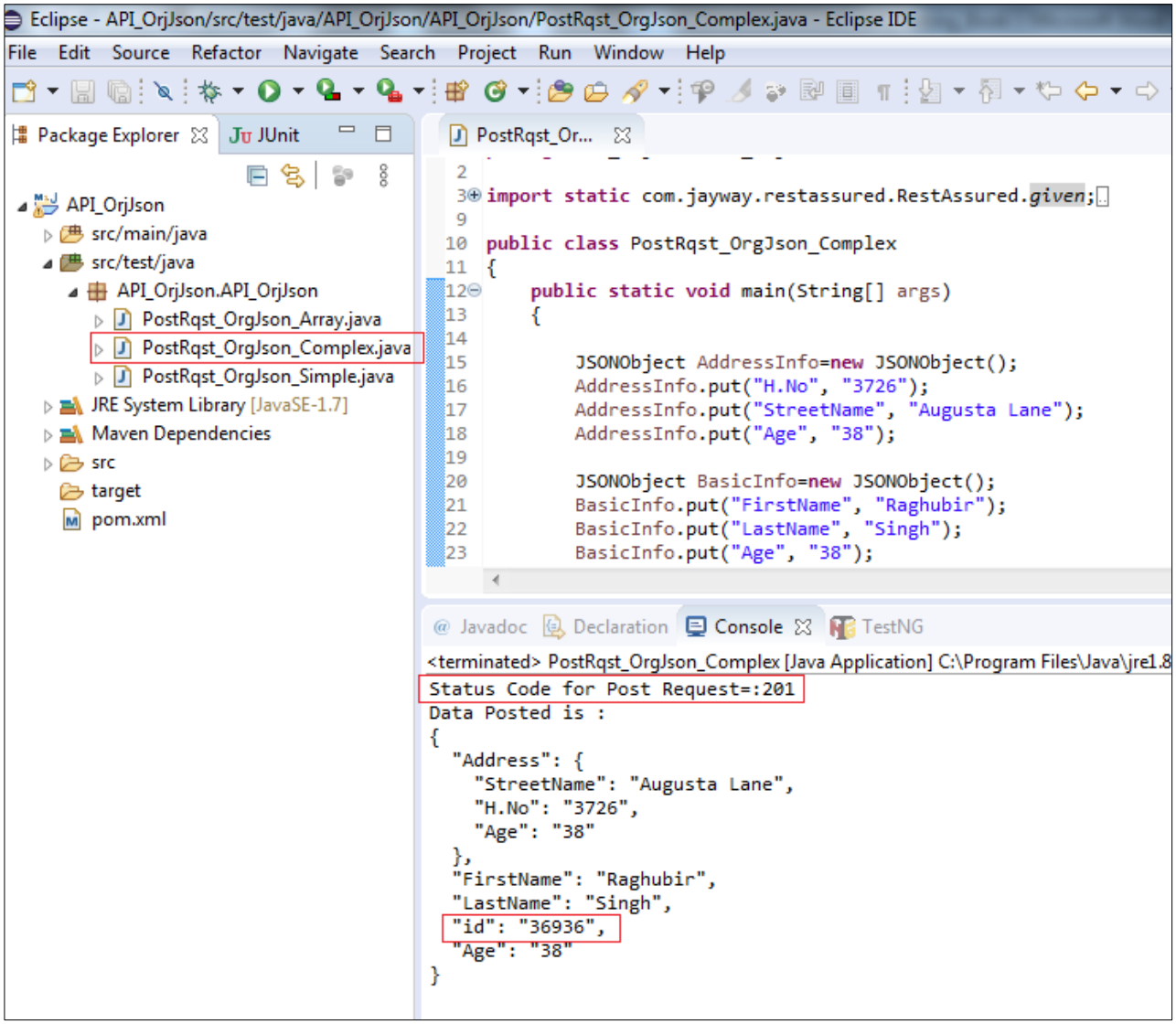
4. Not the existing data in the targeted Json file. Open the file in Notepad and note the data in the present in the Json file. Copy whole data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and view the json in the viewer tab:



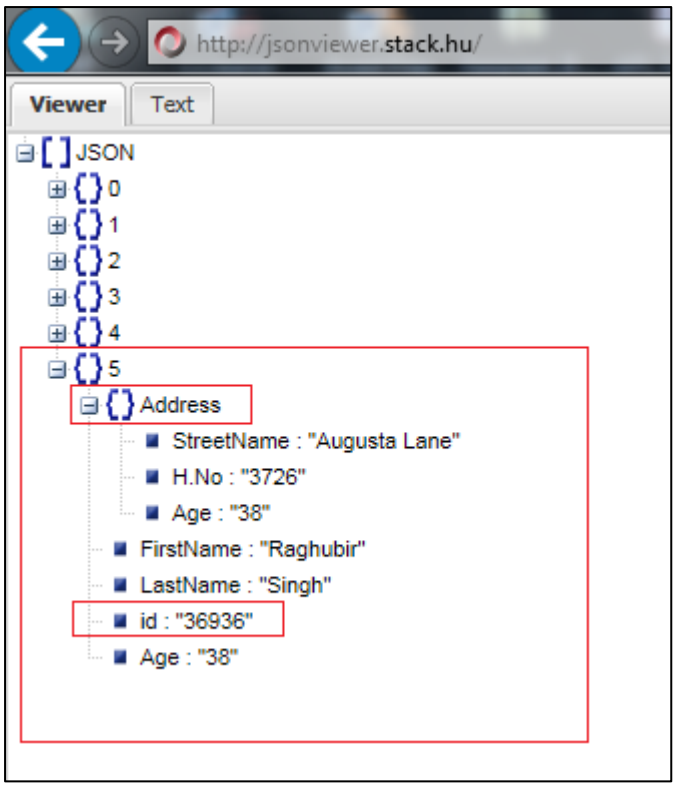
5. Start the Json Server and the resource URI's will be created as per the structure of the file:



6. Run the program and verify the results in the output window. It is to be notice that the data in the form of Complex Json should be posted at the provided URI and in the target .json file.



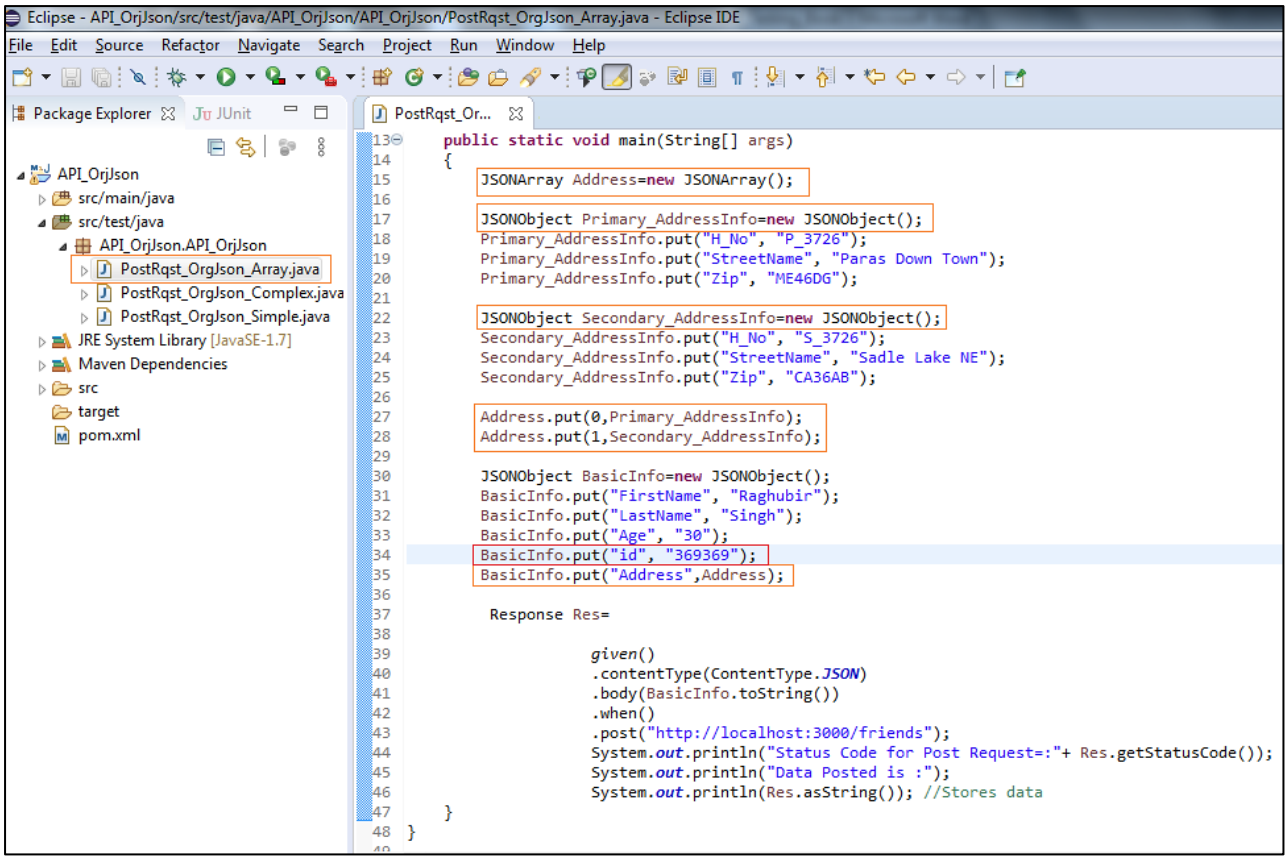
7. Verify the results into the targeted Json file also. Open the file in Notepad and verify the data in the form of Complex Json posted by the program above or copy whole of the data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and then view the data in the viewer tab:



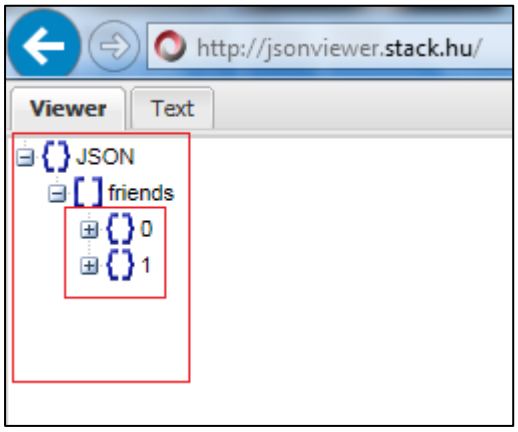
Locating the data posted in the Text tab.

3. **Body Creation In The Form Of Array:** Body data creation in the form of array is created using the array notation as shown below:

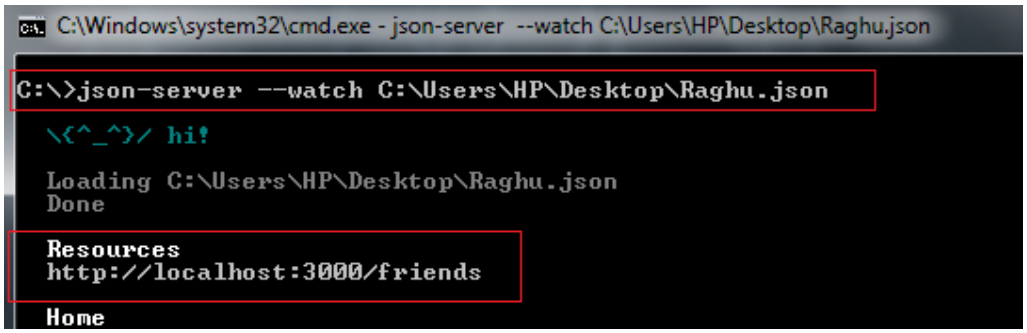
1. Create one class say API_Orgjson and under the package API_Rqst_OrgJson_Array.java
2. Import all the static methods of RestAssured
3. For creating the body data in the form of array with Org.json we need to create an object of JSONArray class and for that we have created BasicInfo object. This object of the JSONObject class created along with .put method will be used to assign the values to the keys as shown below:



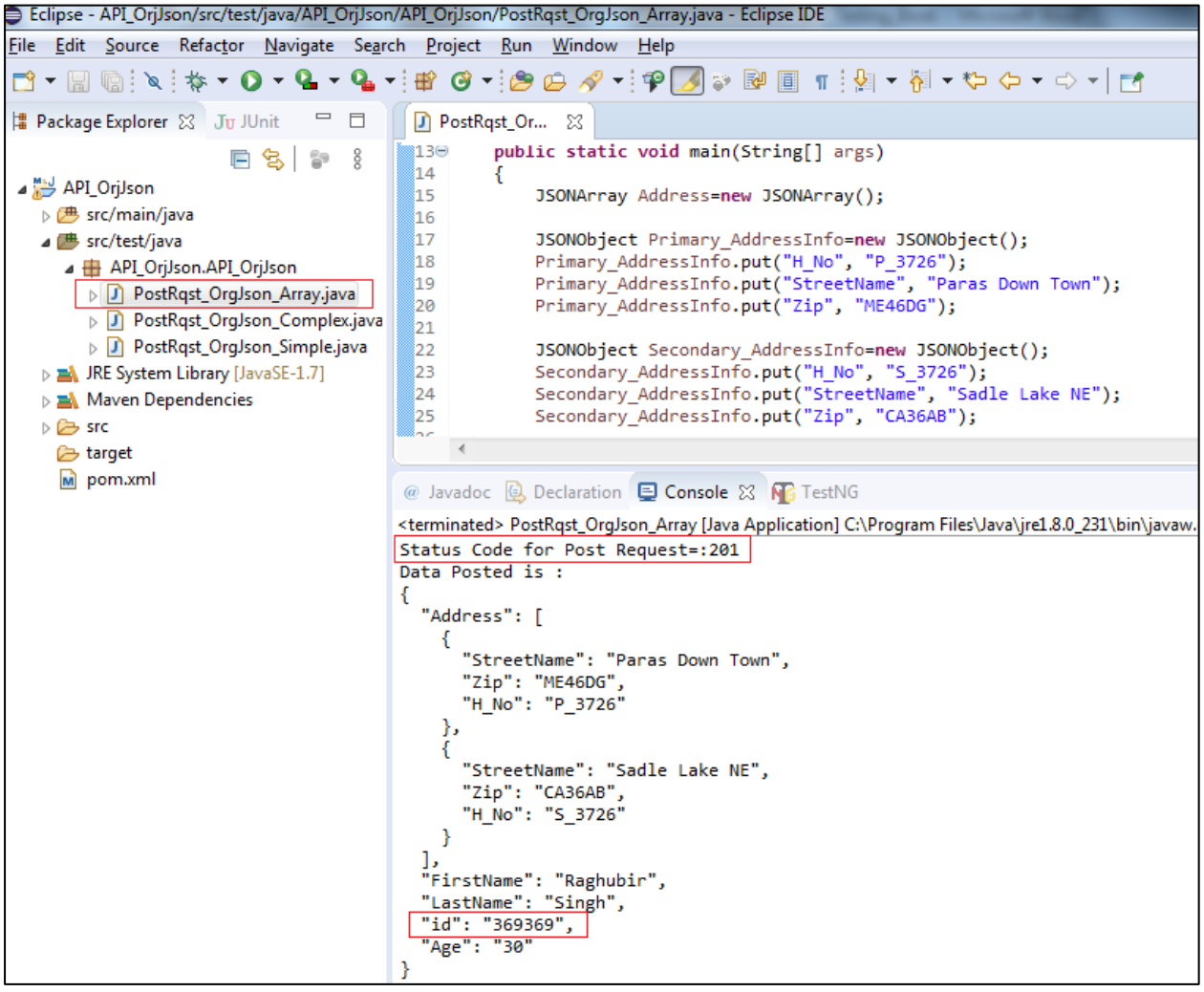
4. Not the existing data in the targeted Json file. Open the file in Notepad and note the data in the present in the Json file. Copy whole data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and view the json in the viewer tab:



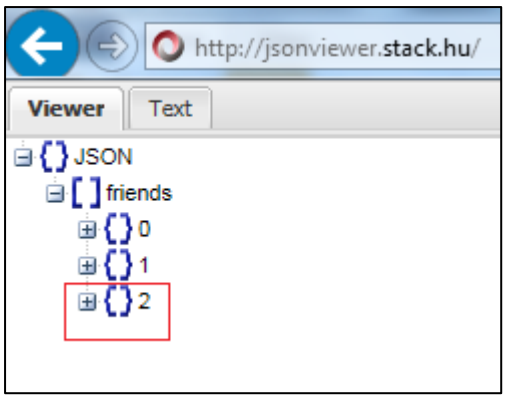
5. Start the Json Server and the resource URI's will be created as per the structure of the file:



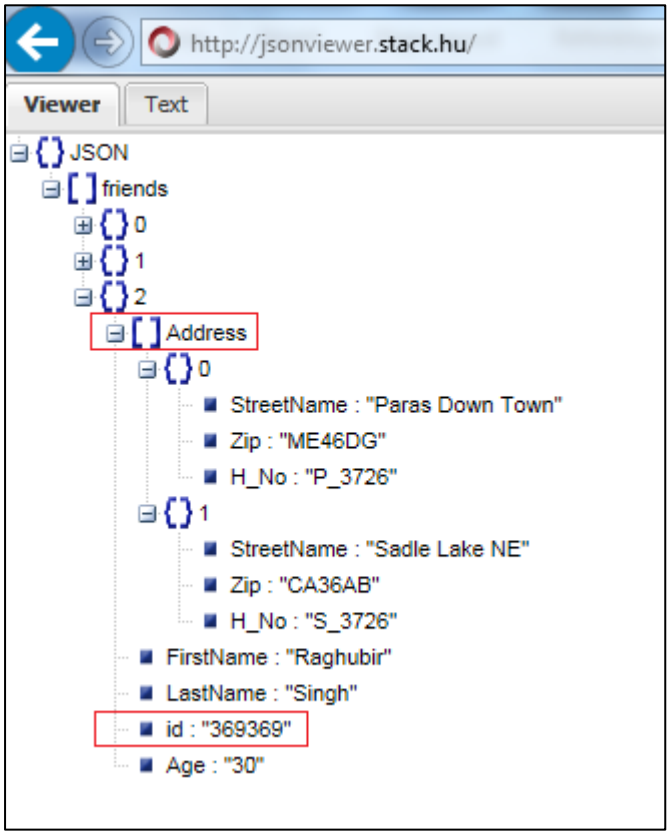
6. Run the program and verify the results in the output window. It is to be notice that the data in the form of Array Json should be posted at the provided URI and in the target .json file.



7. Verify the results into the targeted Json file also. Open the file in Notepad and verify the data in the form of Complex Json posted by the program above or copy whole of the data from this Notepad file and Navigate to online tool : http://jsonviewer.stack.hu/ and paste the whole data in the Text tab and then view the data in the viewer tab:



So, we do have the data in the form of Array Json posted successfully as shown:



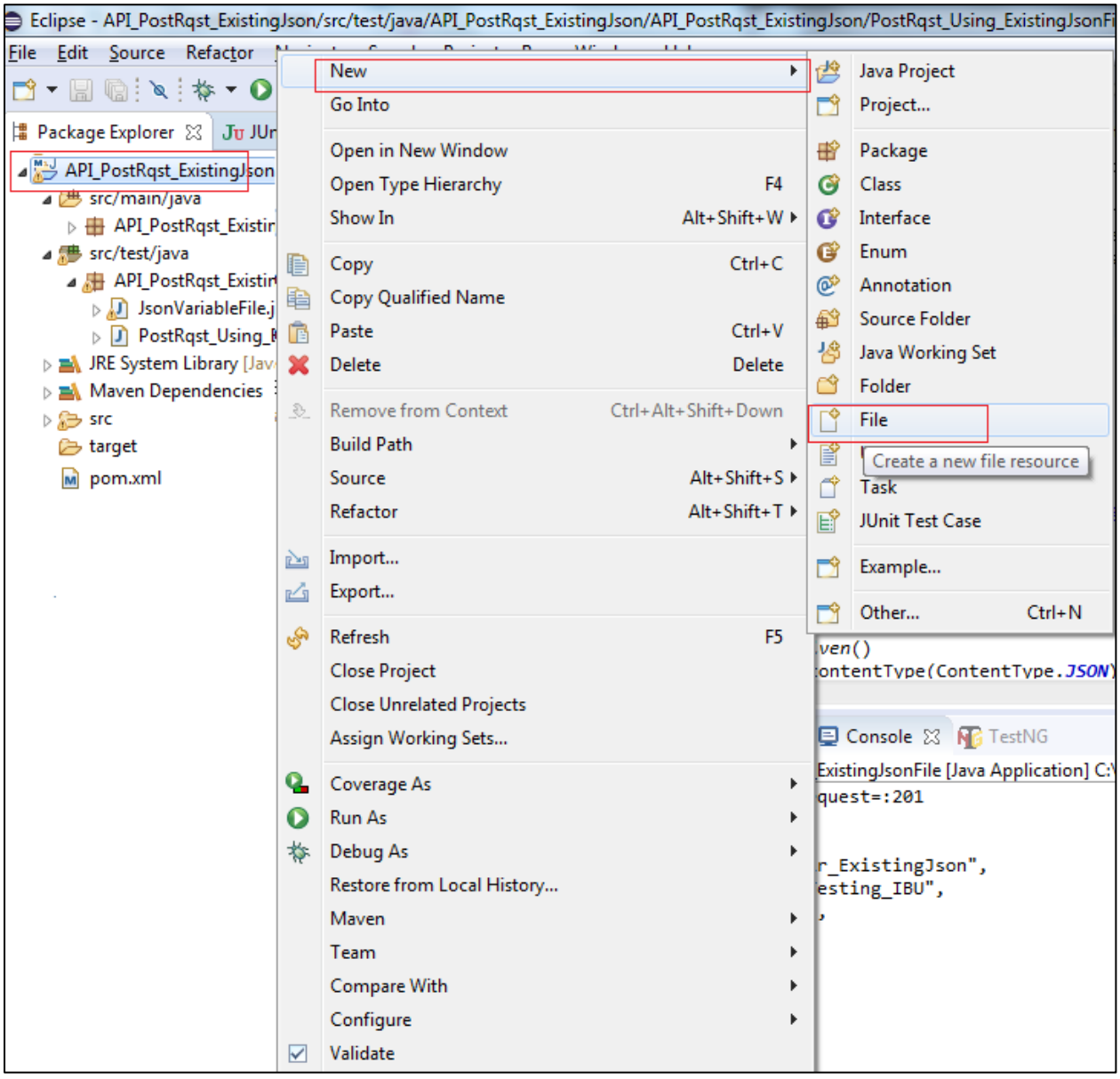
Chapter -6: Body Data Creating Using Existing Json File

Body Data Creation using existing JSON File: This is one of the most interesting methods of creating and passing the Json data. For this we need a Org.Json maven dependencies to work around this methodology of body data creation. We have already discussed in Chapter 3 that how to include the maven dependency of Org.json.

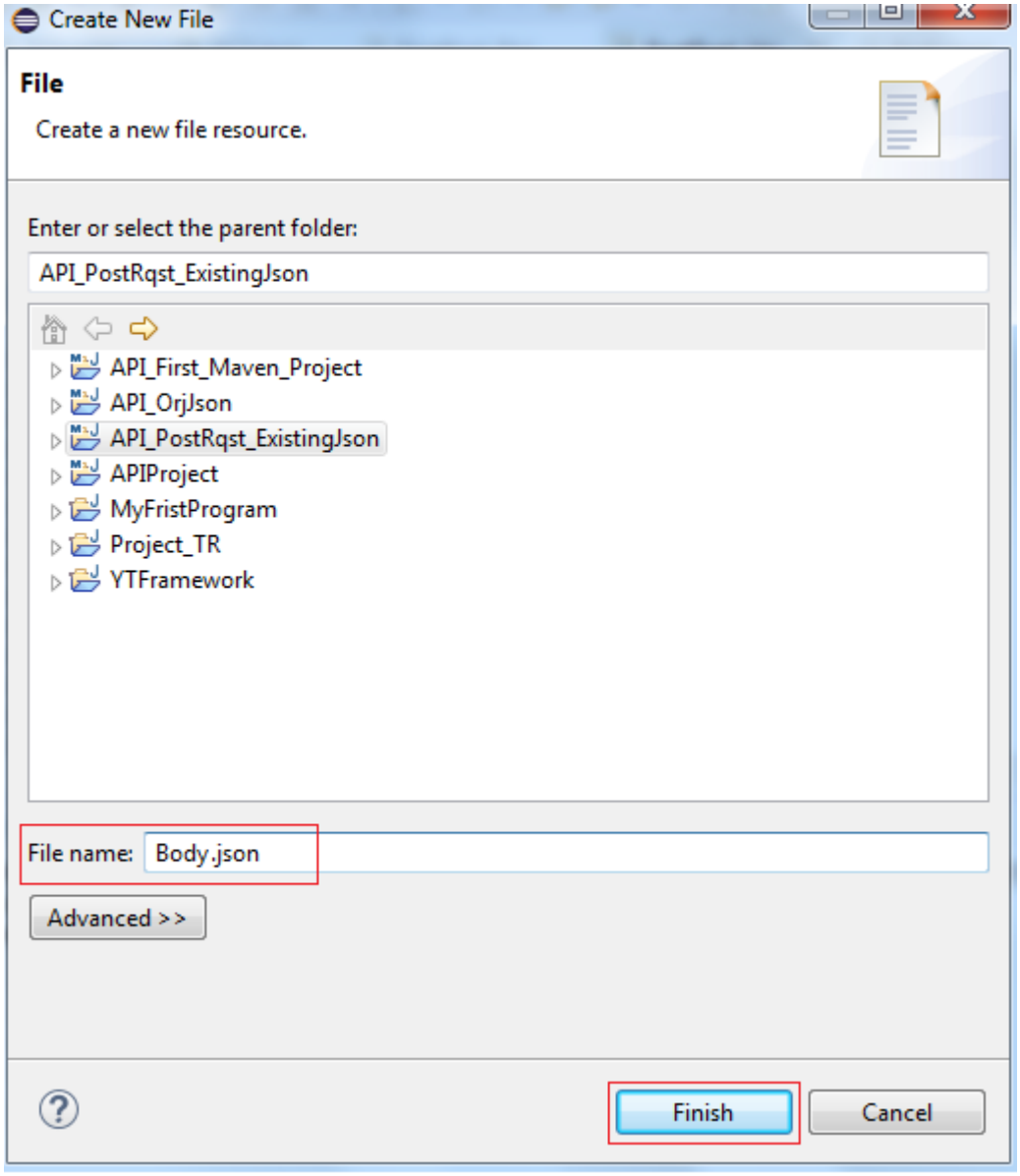
1. **Practical Practices:** Hitting the Post request after reading data from existing Json file where data is already present in the Json file in Json format.

Steps:
Assumption: There exist or create a project with Org.Json and Rest Assured Maven Dependencies.

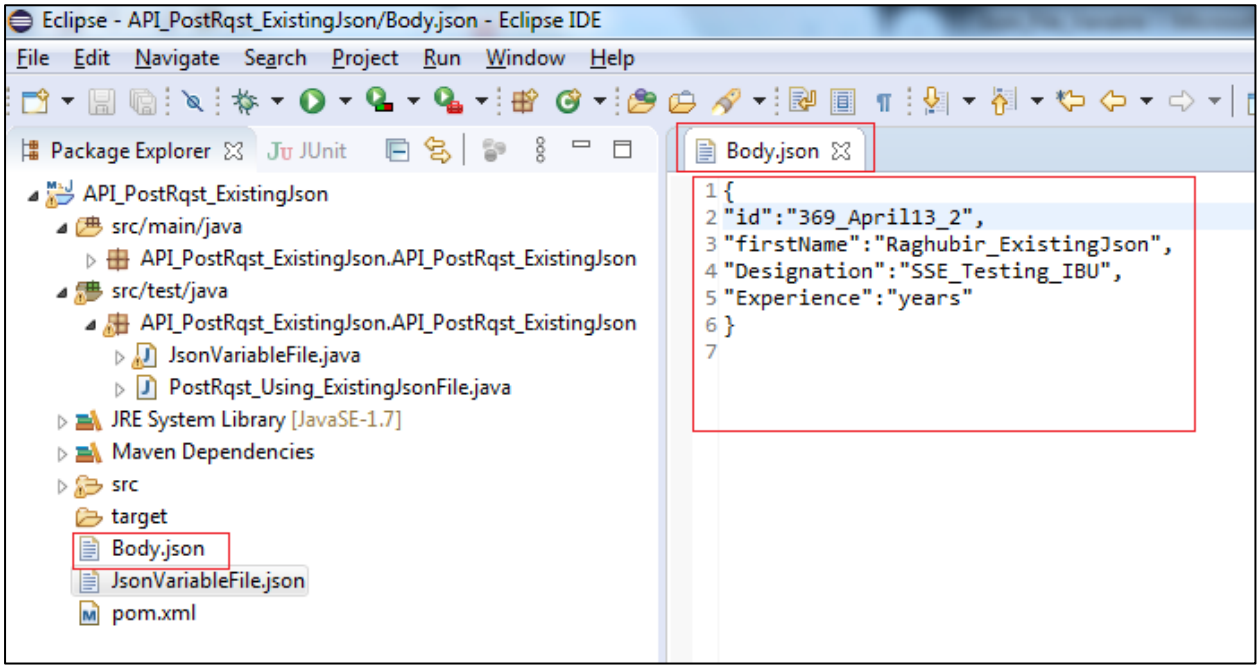
1. Right click on the maven project and click on New> File



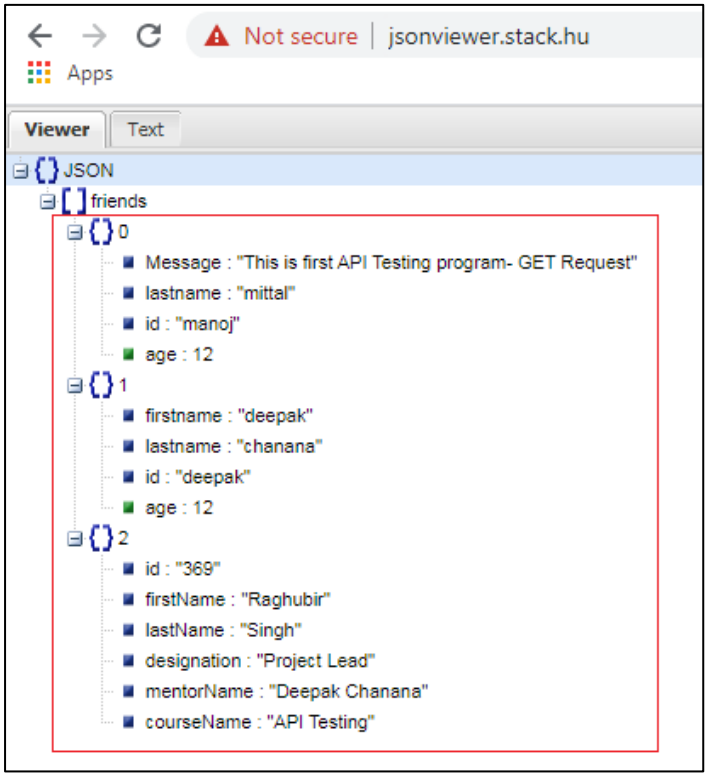
2. The following dialogue box pops up. Give the name say Body.json (Please note that here name could be anything but extension must be **.json**) and click on Finish button.



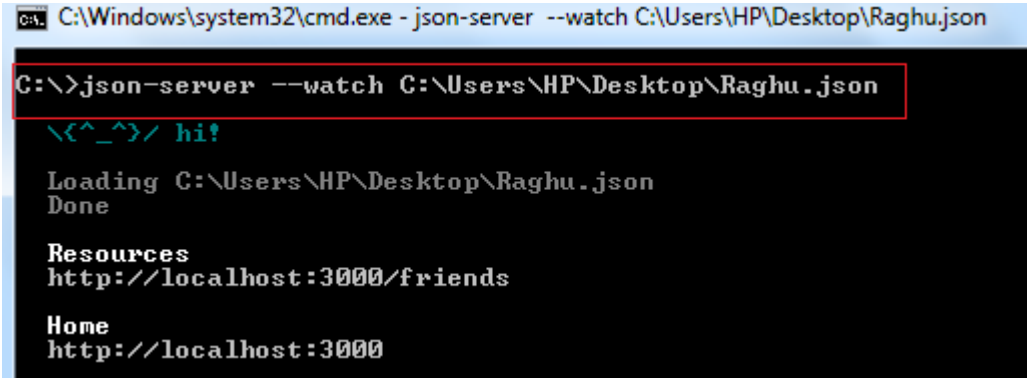
3. Open the the Body.json file created above and just type in the simple Json data as shown below and save the same:



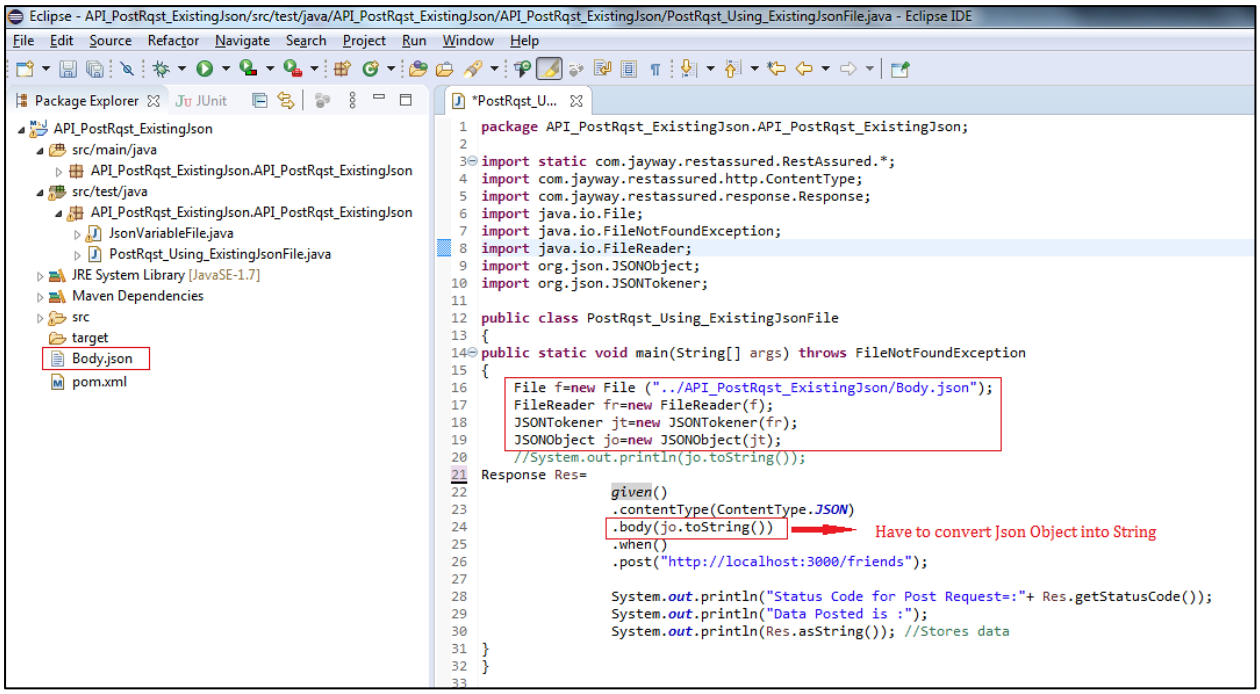
4. Let's check the data in the target Json file, the number of objects it has at the moment before hitting the Post Request and then we will again check the same file after hitting the post request. For this just open the target file copy the entire data from this file and navigate to the online tool and https://jsonviewer.stack.hu and paste under the text tab and then click on the Viewer tab as shown below. So far, we have three objects in the target file {}0,{}1 and {}2.



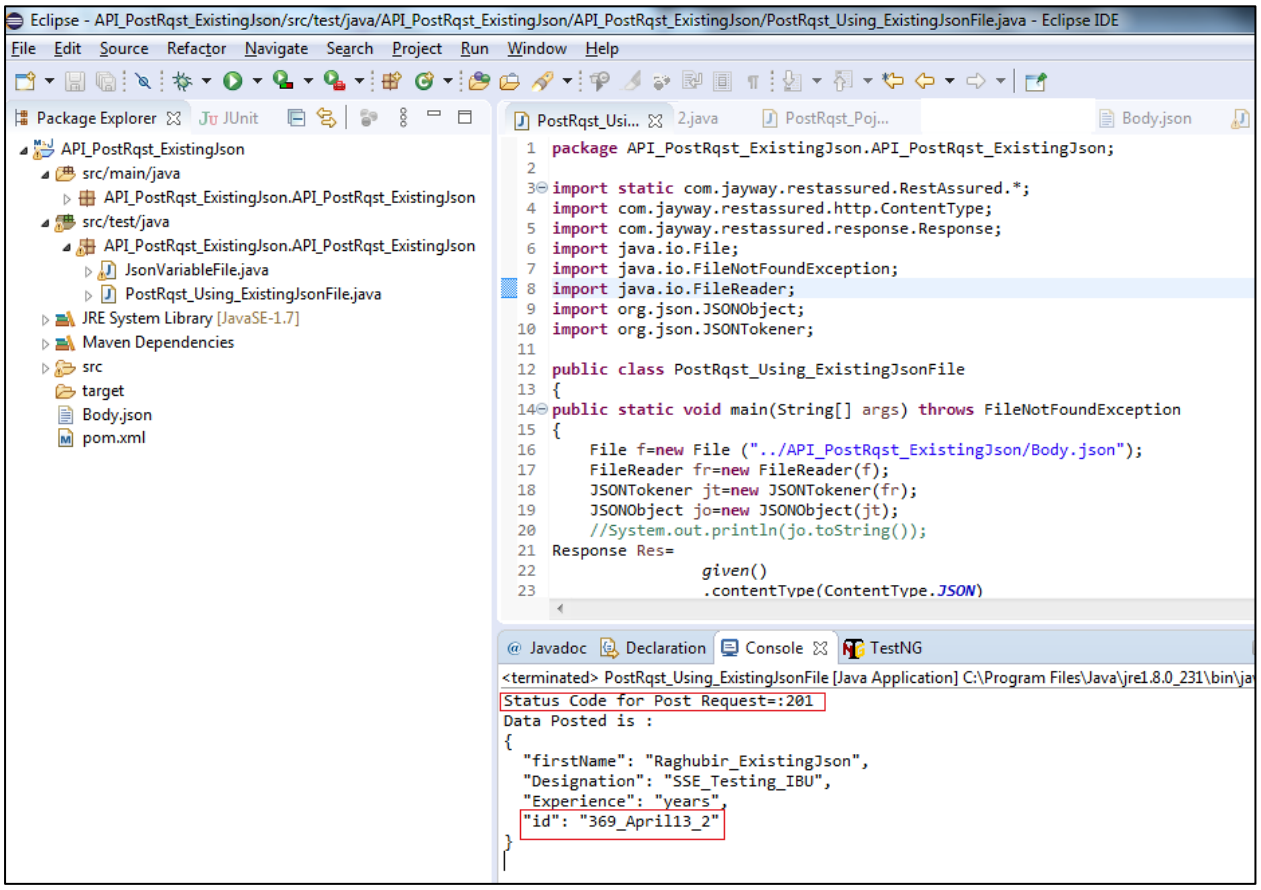
5. Let's start the Json server and pass the path of our target .json file as shown below:



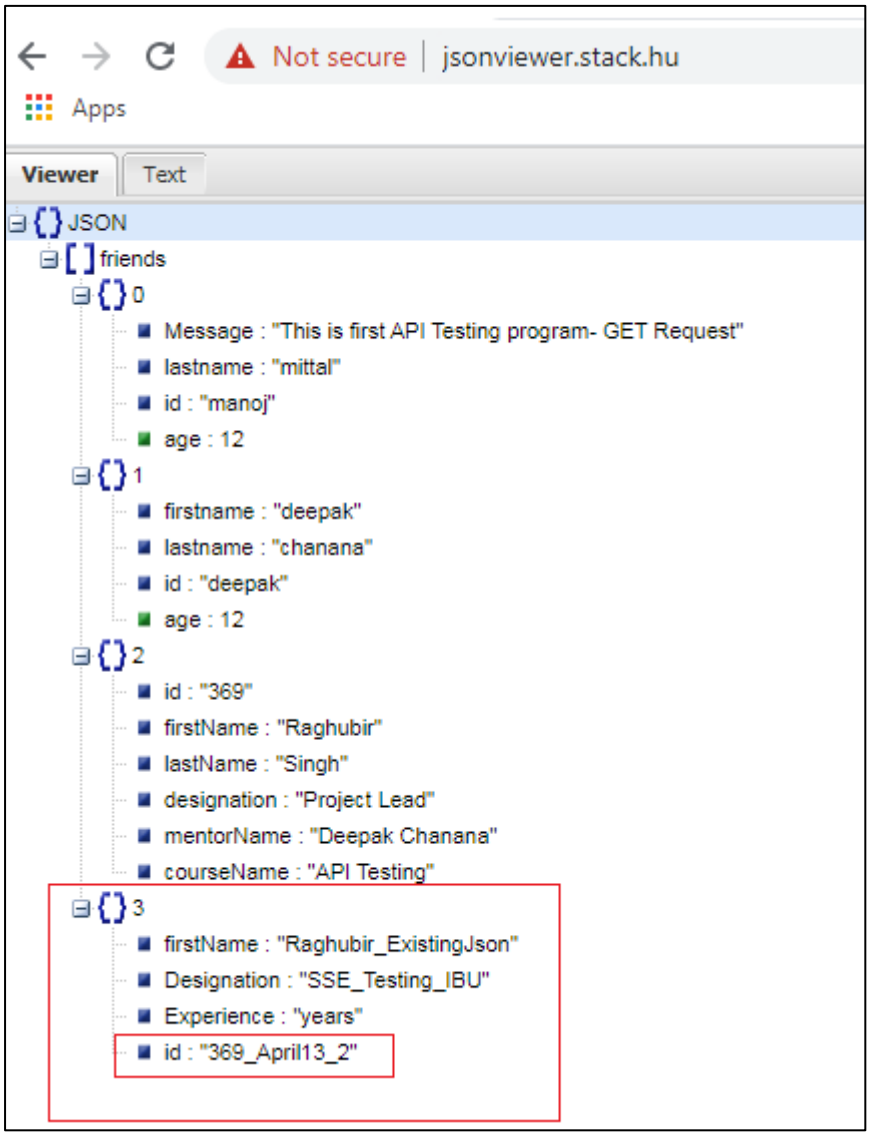
6. Just write the code as written below and it will be explained further once the program is completed.



7. Hit the Post Request by executing the program:



8. Verify the data in the Target file by copying all the data from the file and check online at jsonviewer.stack.hu



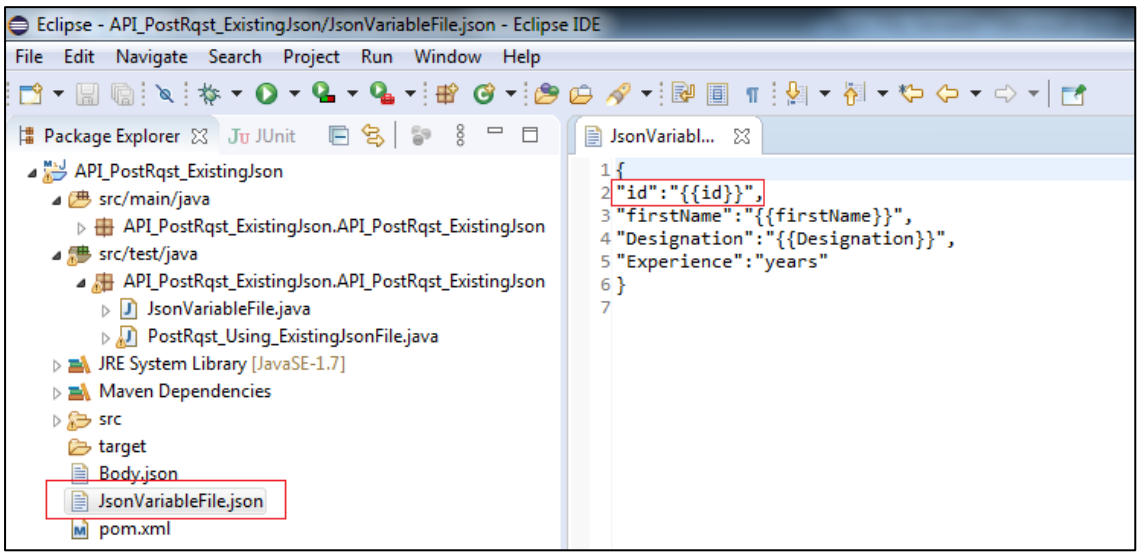
The data is posted successfully in the target .json file under the URI.

2. **Practical Practices:** Hitting the Post request after reading data from existing Json file where some data is already present in the Json file in Json format while few data variables are declared in that file and value to these variables are provided in program itself.

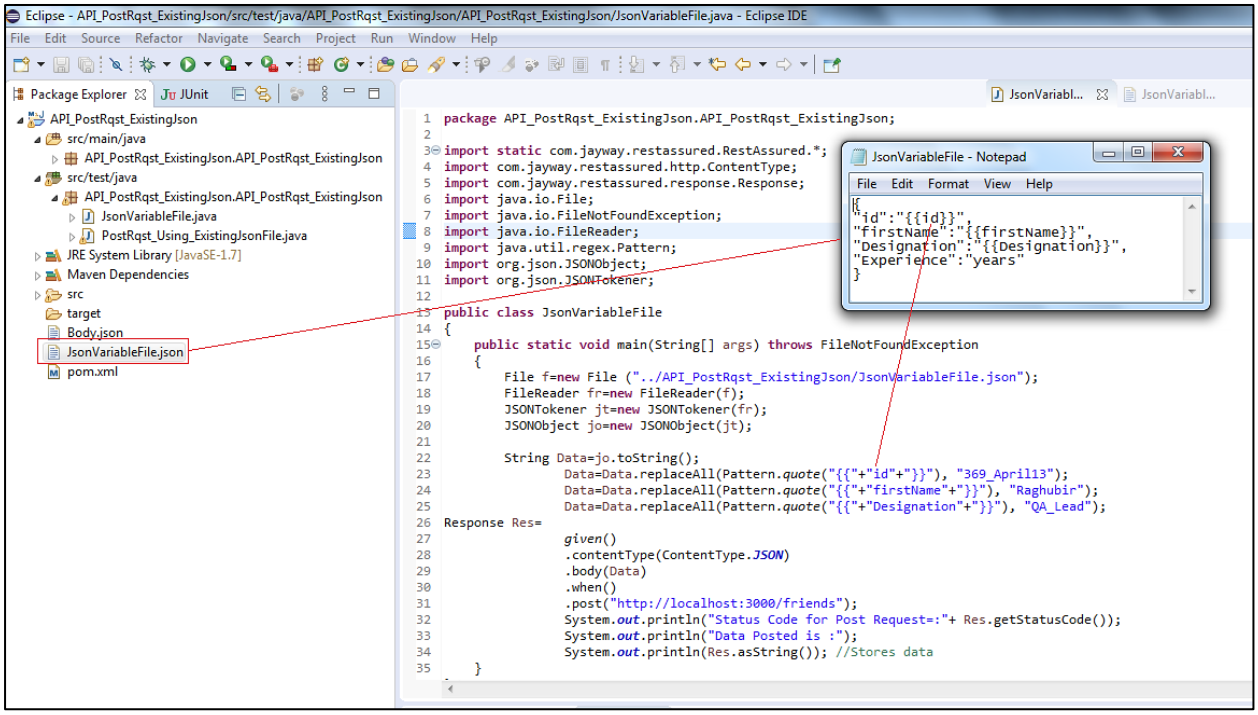
Create one ***Json file*** under your respective project and define variable as:

{ “id”: “{{id}}” }
The way to define a variable in a Json file is to start with {{ and then the key name close again with }}

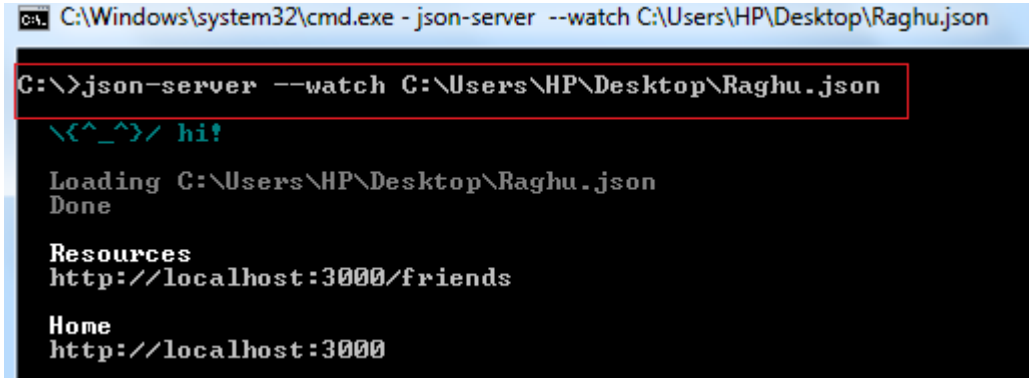
1. Right click on project and select new > file the same way as we did in Practical Practice 1
2. Give filename say JsonVariableFile.json
3. Write the following code in this file and save the file.



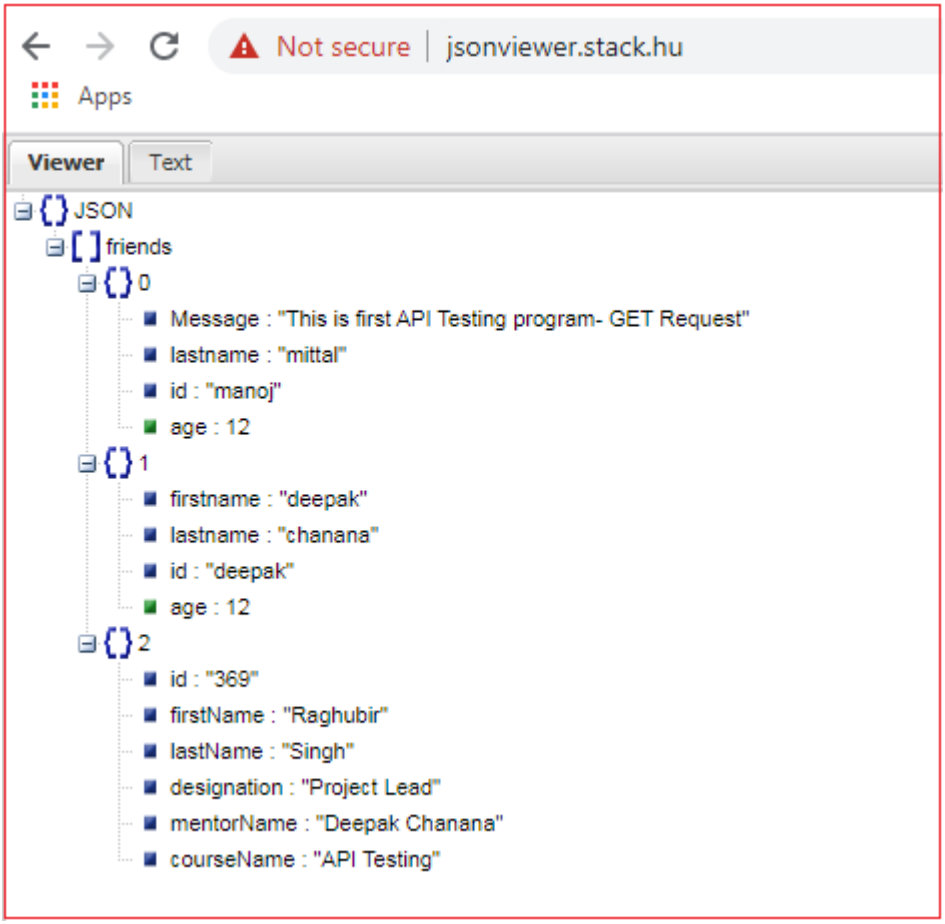
4. Create on class say JsonVaribleFile.java and write the following code as shown below. Please note the way values are assigned to the variables.



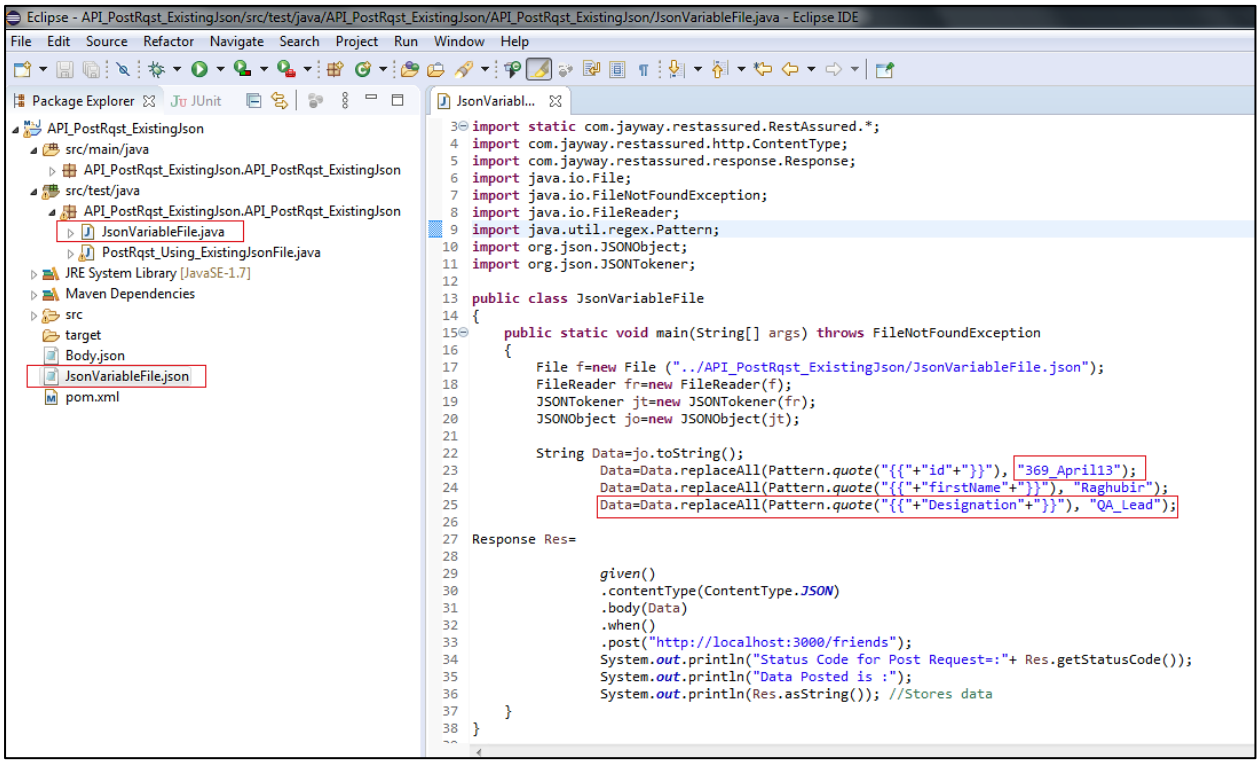
5. Let's start the Json server and pass the path of our target .json file as shown below:



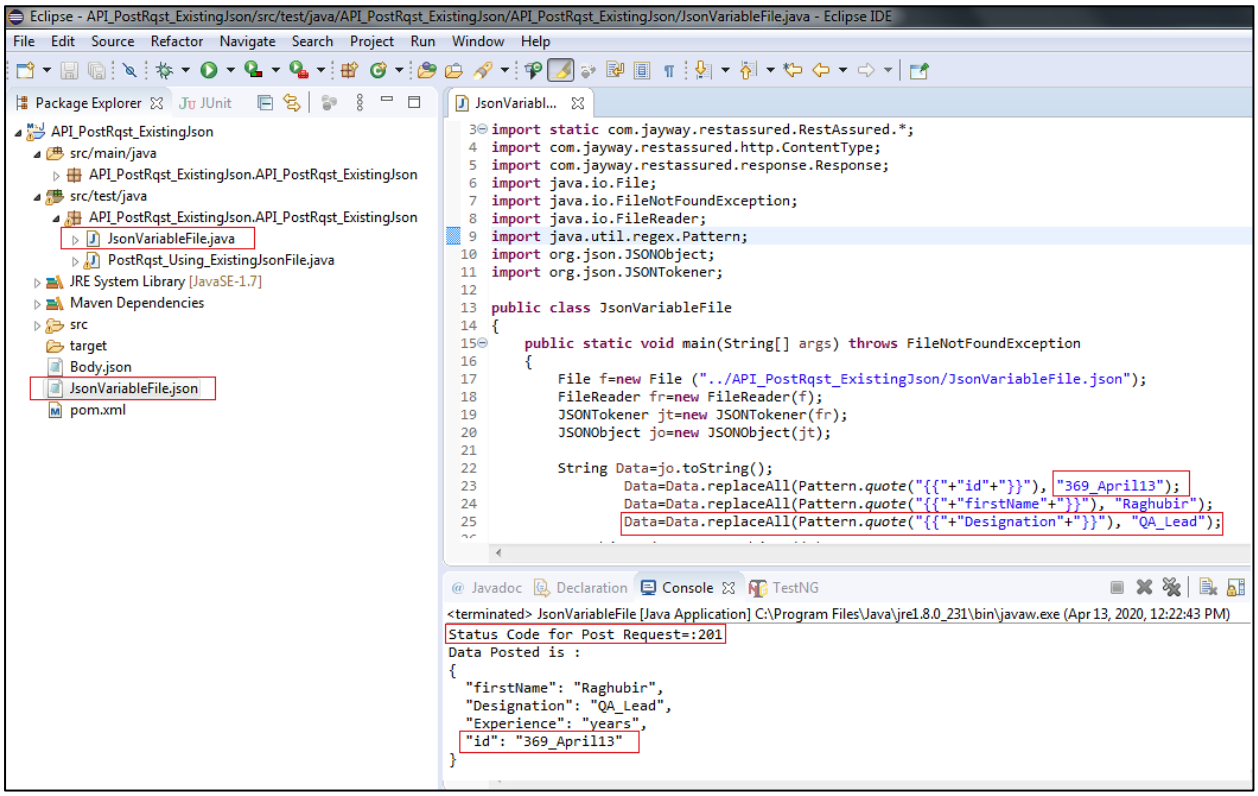
6. Let's check the data in the target Json file, the number of objects it has at the moment before hitting the Post Request and then we will again check the same file after hitting the post request. For this just open the target file copy the entire data from this file and navigate to the online tool and https://jsonviewer.stack.hu and paste under the text tab and then click on the Viewer tab as shown below. So far, we have three objects in the target file {}0,{}1 and {}2.



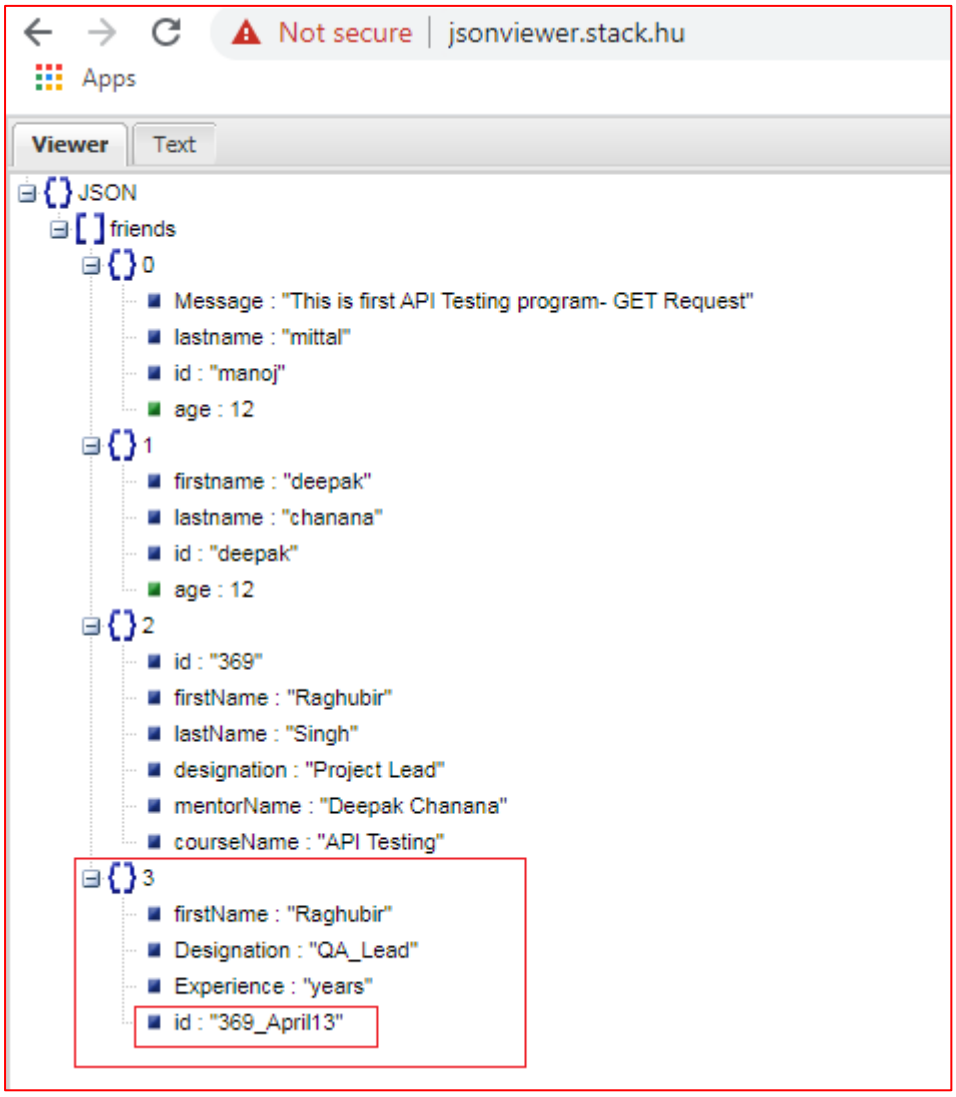
7. Start the Json Server and provide the target Json file name in command line while starting the Json server as we usually do.



8. Execute the program and see the output that the post request hit successfully:



9. Now the post request is hit. Open the target file in Notepad and copy the data and view online at jsonviewer.stack.hu



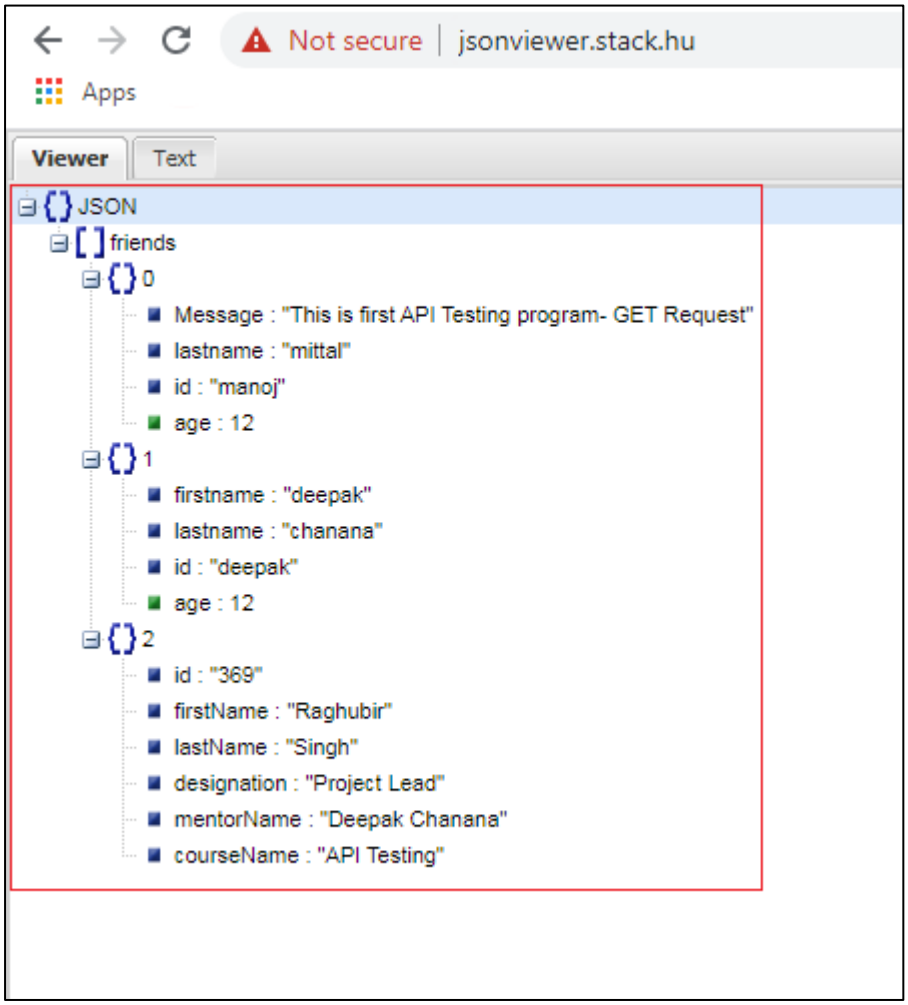
The data is posted successfully in the target .json file under the URI.

3. **Practical Practices:** Hitting the Post request after reading data from existing Json file where some data is already present in the Json file in Json format while few data variables is declared in this file and value to these variables are provided by the user at run time.

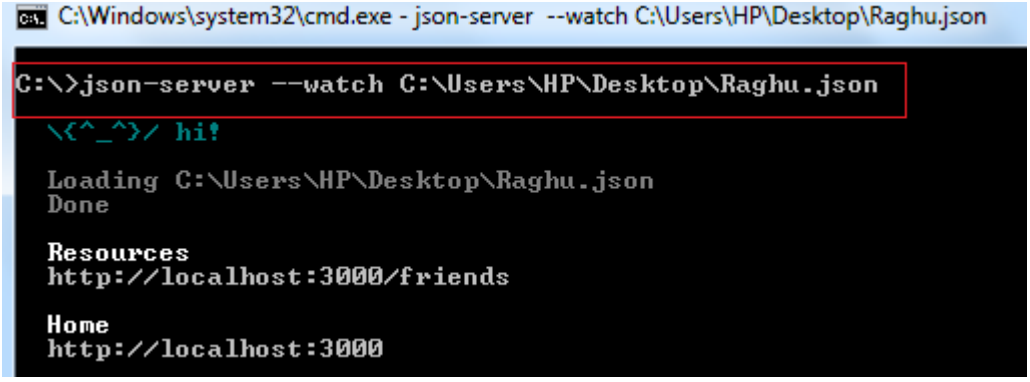
1. Lets amend the above code in the above program by decalaring some variables and scanning thise variables from at the Run time.

```
public class JsonVariableFile
{
        public static void main(String[] args) throws FileNotFoundException
    {
            File f=new File ("../API_PostRqst_ExistingJson/JsonVariableFile.json");
            FileReader fr=new FileReader(f);
            JSONTokener jt=new JSONTokener(fr);
            JSONObject jo=new JSONObject(jt);

            String firstname, id, Designation;
            Scanner Variables=new Scanner(System.in);

            id=Variables.next();
            firstname=Variables.next();
            Designation=Variables.next();

            String Data=jo.toString();

            Data=Data.replaceAll(Pattern.quote("{{"+"id"+"}}"),id );
            Data=Data.replaceAll(Pattern.quote("{{"+"firstName"+"}}"),firstname);
            Data=Data.replaceAll(Pattern.quote("{{"+"Designation"+"}}"),Designation);

            Response Res=
                        given()
                        .contentType(ContentType.JSON)
                        .body(Data)
                        .when()
                        .post("http://localhost:3000/friends");


    System.out.println("Status Code for Post Request=:"+ Res.getStatusCode());
    System.out.println(("Data Posted is :"+Res.asString()); //Stores data
    }
}
```

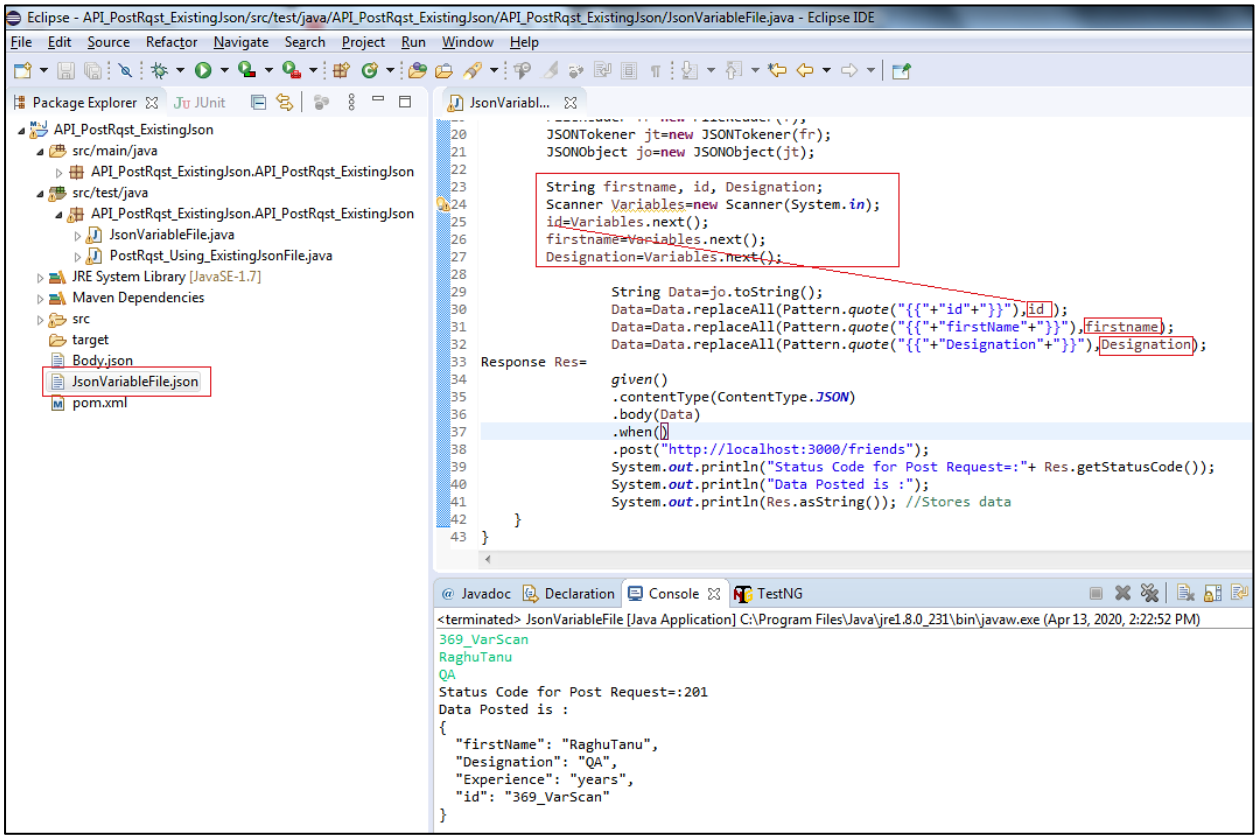
2. Let's check the data in the target Json file, the number of objects it has at the moment before hitting the Post Request and then we will again check the same file after hitting the post request. For this just open the target file copy the entire data from this file and navigate to the online tool and https://jsonviewer.stack.hu and paste under the text tab and then click on the Viewer tab as shown below. So far, we have three objects in the target file {}0,{}1 and {}2.



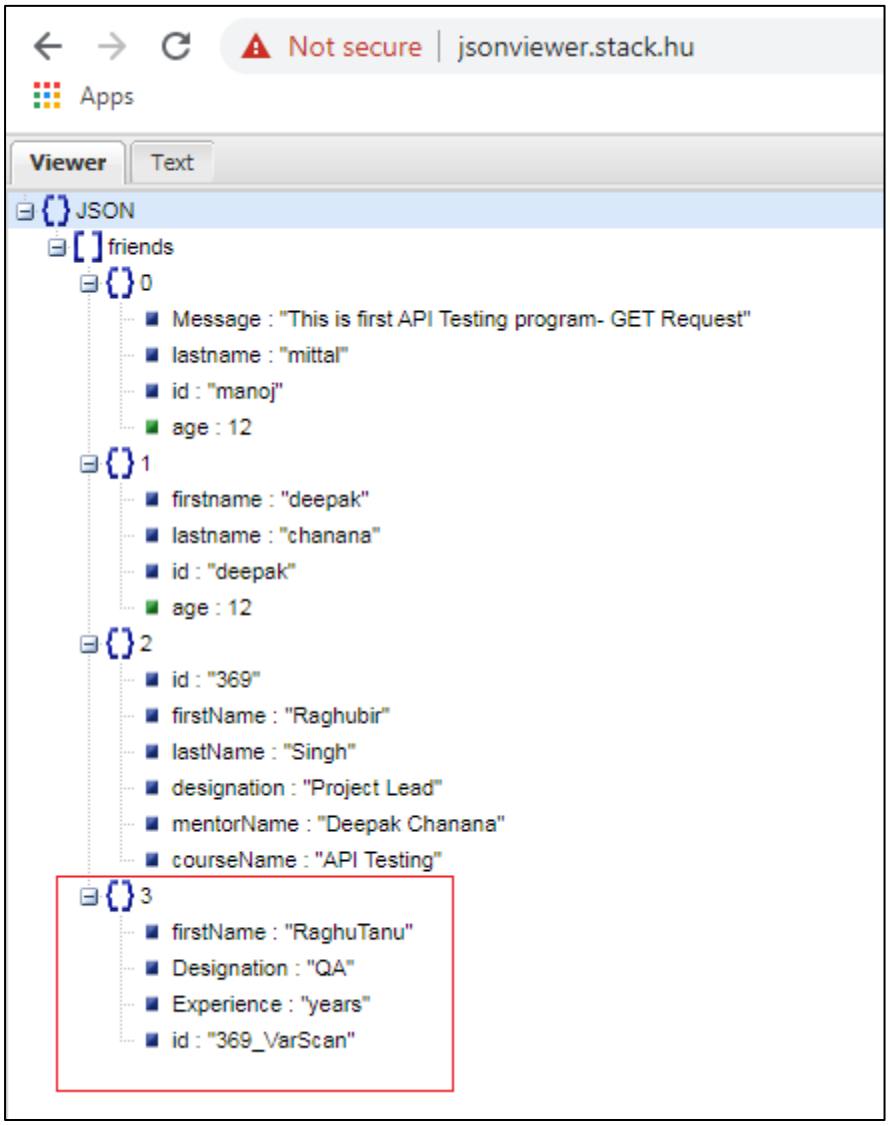
3. Let's start the Json server and pass the path of our target .json file as shown below:



4. We have already replaced the values those are hard code in the program with the variable defined in the program and these variables will already have these values at the run time from the user once the program executed.



5. Now the post request is hit. Open the target file in Notepad and copy the data and view online at jsonviewer.stack.hu



The data is posted successfully in the target .json file under the URI.

Chapter -7: Response Data Parsing

Response Data Parsing and JSONPath

JSONPath tool lets you analyze and selectively extract data from a JSON structure. JSONPath is similar to XPath for XML. JSONPath uses JSONPath expressions to select elements from a JSON structure.
Let us take a very simple use case where we want to extract the event name from the following JSON document.

```
{
 "event": {
 "name": "agent",
 "data": {
          "name": "James Bond"
           }
           }
}
```

You can query for event name using following JSONPath expression
event.name

JSONPath expressions support following operators
Supported Operators

Operator	Description
$	The root element to query. This starts all path expressions.
@	The current node being processed by a filter predicate.
*	Wildcard. Available anywhere a name or numeric is required.
..	Deep scan. Available anywhere a name is required.
.<name> OR [ ]	Dot-notated child
[,]	Union Operator
[ ]	Subscript Operator
['<name>' (, '<name>')]	Bracket-notated child or children
[<number> (, <number>)]	Array index or indexes
[start:end:step]	Array slice operator
[?(<expression>)]	Filter expression. Expression must evaluate to a boolean value.
()	Script expression, using the underline engine

Chapter-7 Table 1

Functions
Functions can be invoked at the tail end of a path - the input to a function is the output of the path expression. Following functions are supported

Function	Description	Output
min()	Provides the min value of an array of numbers	Double
max()	Provides the max value of an array of number	Double
avg()	Provides the average value of an array of numbers	Double
stddev()	Deep scan. Available anywhere a name is required.	Double
length()	Dot-notated child	Double

Chapter-7 Table 2

Note: One can practice the Json path creations with online tool provided at: http://jsonpath.com/

Filter Operators

Filters are logical expressions used to filter arrays. A typical filter would be [?(@.age > 18)] where @ represents the current item being processed. More complex filters can be created with logical operators && and ||. String literals must be enclosed by single or double quotes ([?(@.color == 'blue')] or [?(@.color == "blue")]).

Operator	Description
==	left is equal to right (note that 1 is not equal to '1')
!=	left is not equal to right
<	left is less than right
>	left is greater than right
>=	left is greater than or equal to right
=~	left matches regular expression [?(@.name =~ /foo.*?/i)]
in	left exists in right [?(@.size in ['S', 'M'])]
nin	left does not exists in right
subsetof	left is a subset of right [?(@.sizes subsetof ['S', 'M', 'L'])]
size	size of left (array or string) should match right
Empty	left (array or string) should be empty

Chapter-7 Table 3

JSONPath expression examples
For the following JSON:

```
{
  "event": {
    "name": "Bond Movies",
    "movies": [
      {
        "name": "Licence to Kill",
        "star": "Timothy Dalton",
        "rating": 6.6
      },
      {
        "name": "GoldenEye",
        "star": "Pierce Brosnan",
        "rating": 7.2
      },
      {
        "name": "Tomorrow Never Dies",
        "star": "Pierce Brosnan",
        "rating": 6.5
      },
      {
        "name": "Skyfall",
        "star": "Daniel Craig",
        "rating": 7.8
      }
    ]
  }
}
```

JSONPath	Result
$	Entire object
$.event	Event object
$.event['name']	Event name
$.event.name	Event name
$..name	All names
$.event.movies[0]	First movie
$.event.['movies']][0]	First movie
$.event.movies[0,2]	First three movies
$.event.movies[:2]	First two movies
$.event.movies[-2:]	Last two movies
$.event.movies[?(@.rating > 7)]	All movies with rating>7
$.event.movies[?(@.star == 'Pierce Brosnan')]	All movies by Pierce Brosnan
$.event.movies.length()	Number of movies

Chapter-7 Table 4

Response data parsing ways

There are two ways by which we can parse the response data:

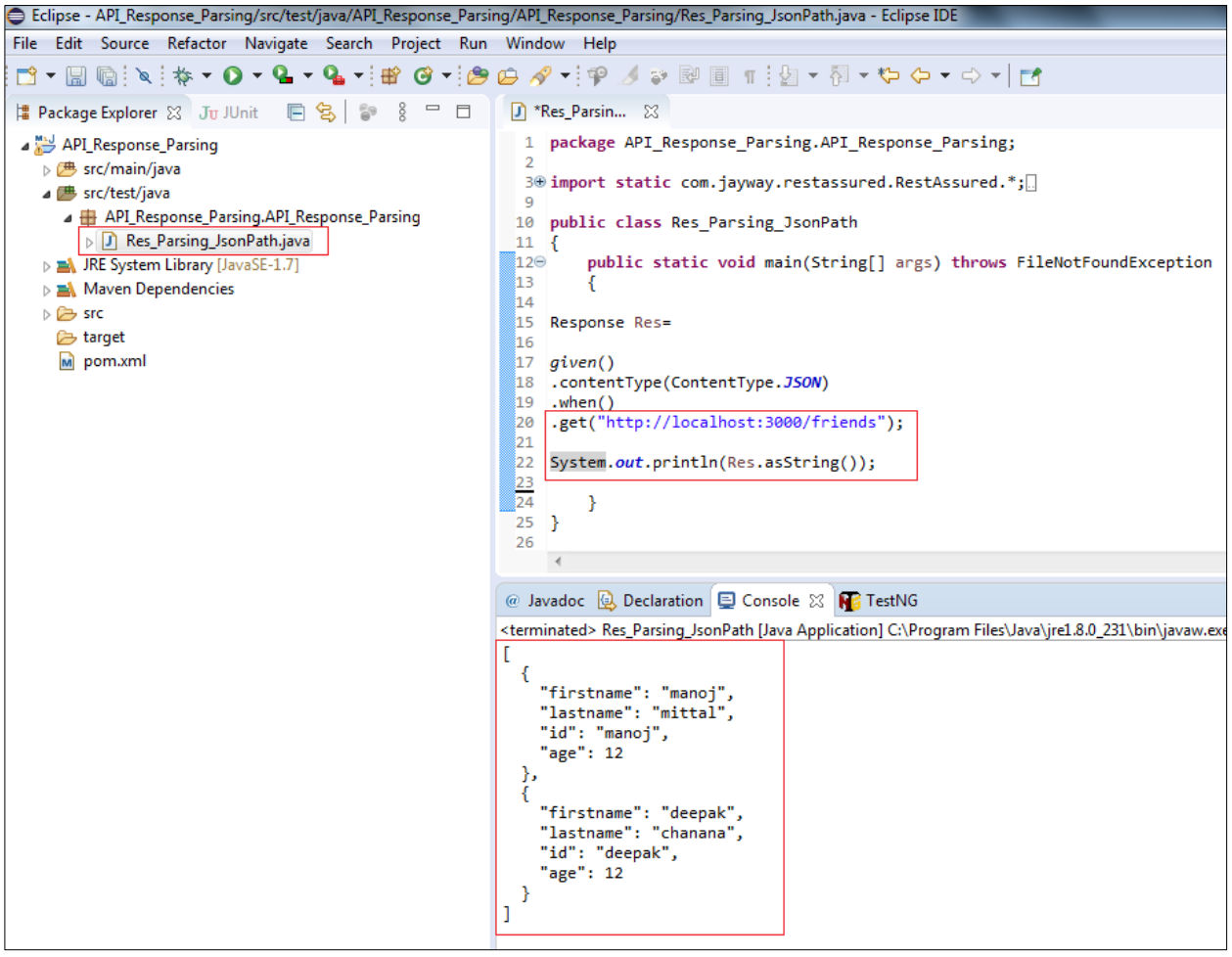
- JSONPath
- Using Org.Json Library

Practicing Json Response Parsing

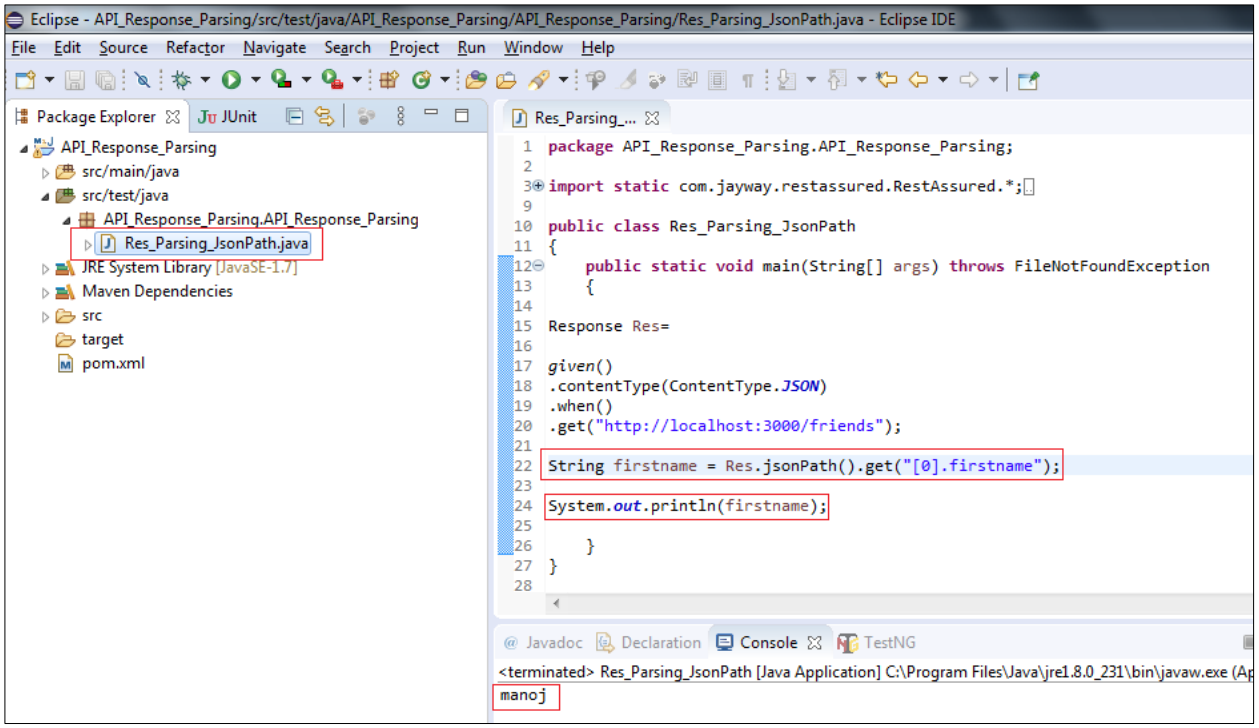
1. Create one maven project say API_Response_Parsing
2. Copy the desires dependencies in respective pom.xml file of this maven project:
   - RestAssured
   - Org.Json and

```
<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
  <version>2.0.0</version>
</dependency>
```

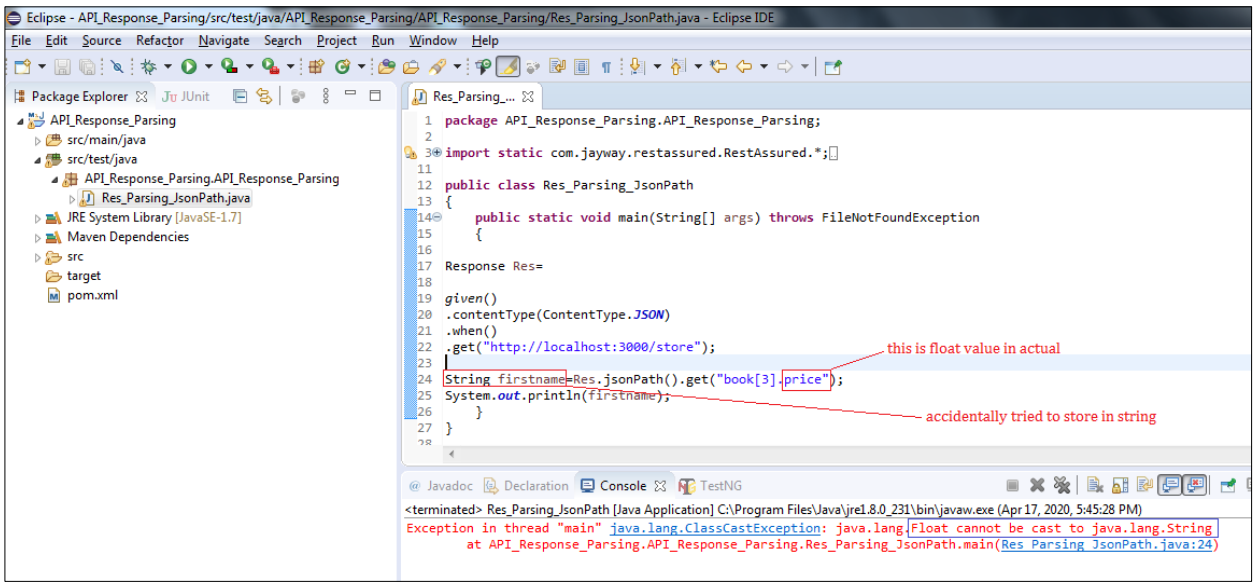
3. Create one class under the API_Response_Parsing. API_Response_Parsing package say Res_Parsing_JsonPath.java
4. Simply write the program to hit the get request to fetch the Json data from your Json file
5. Start the Json Server by passing the path of Json file from which data is to be fetched using get request
6. Write simple code for get https request as shown below and display the response data.



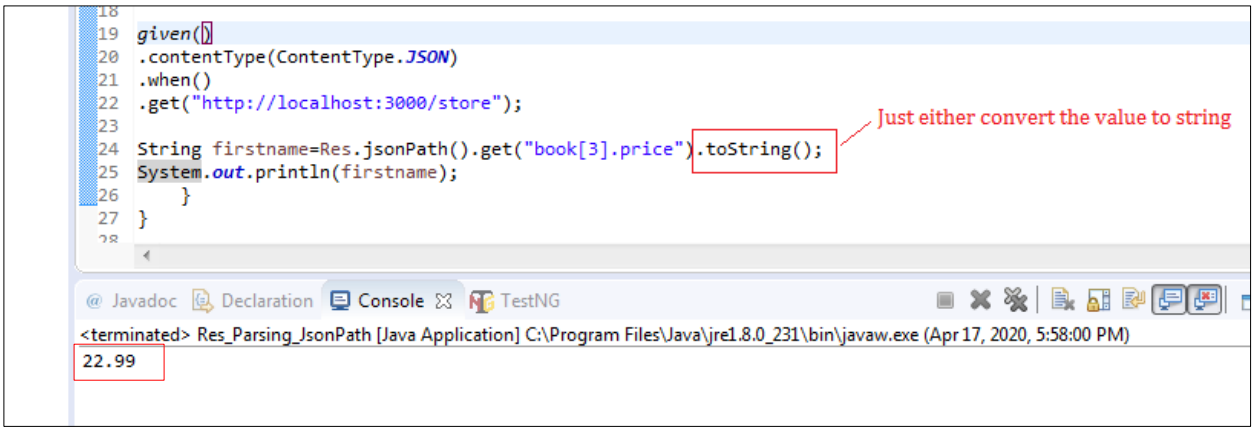
7. It is very clearly observed from the response data output that, there are two objects. Now we want to fetch only the data of say the first object and that's too only firstname. We can achieve this response parsing by two ways either using JSONpath or using Org.json liberary. Lets practice them one by one.



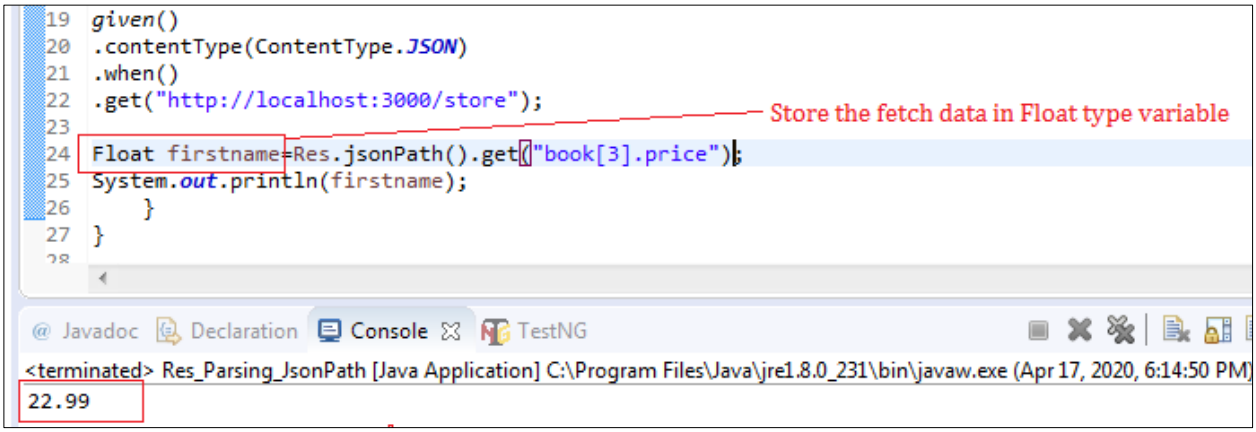
8. This is very important to take care of the data type of the fetched parsed response data as illustrated below:



One of the solutions is to either convert the parsed data into the String or fetch and store the parsed data into the same data type variable as that of parsed data.



Or simply we can declare variable of the same data type as that of data retuned or data parsed:



Response Data parsing Using Json Path:
The jsonPath() method: Get a JsonPath view of the response body. This will let you use the JsonPath syntax to get values from the response. Example:

Assume that the GET request (to http://localhost:8080/lotto) returns JSON as:

```
{
"lotto":{
        "lottoId":5,
        "winning-numbers":[2,45,34,23,7,5,3],
        "winners":[{ "winnerId":23,
                      "numbers":[2,45,34,23,3,5]   },
                    { "winnerId":54,
                      "numbers":[52,3,12,11,18,22]   }]
         }
}
```

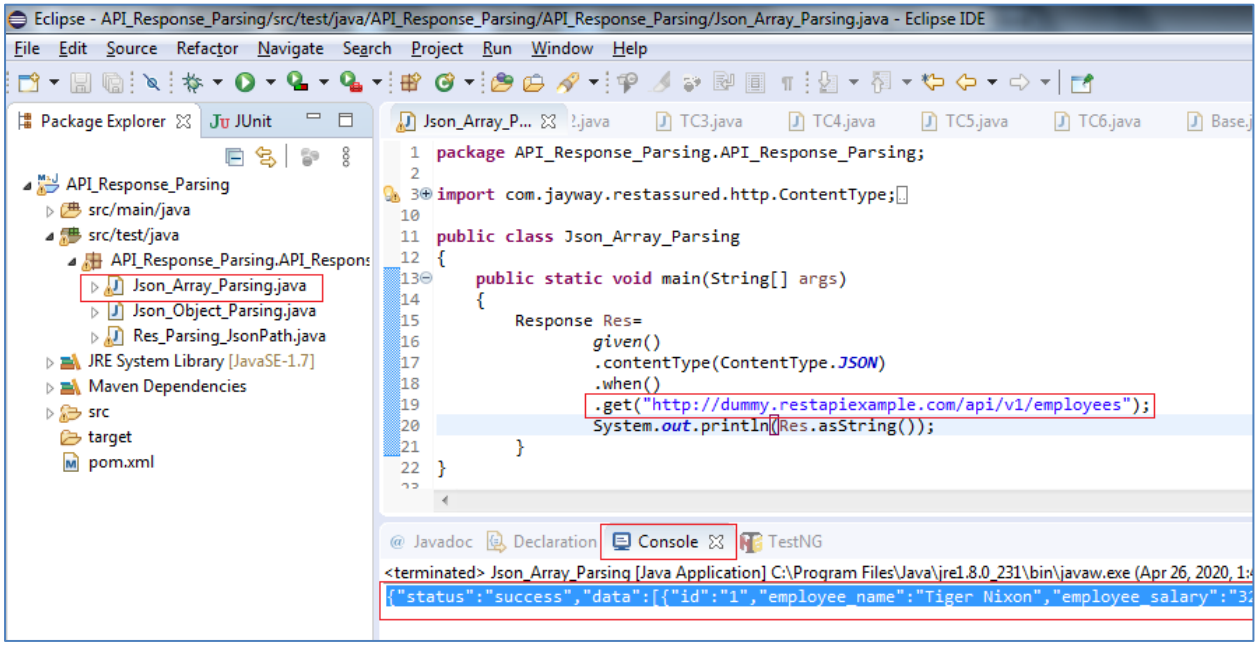
You can make the request and get the winner id's by using JsonPath:

List winnerIds = get("/lotto").jsonPath().getList("lotto.winnders.winnerId");

Let's Practice some more programs to get insight clarity of how to parse response data in different scenario's like when the json start with Json Object and when it starts with Json Array.

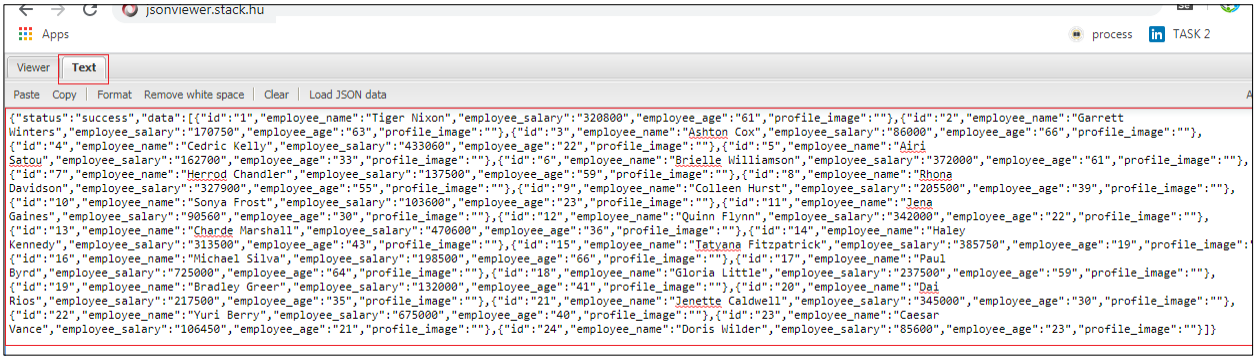
We have used the url given on the website for practicing and fetched the data as per this given moment of time on the URI:- http://dummy.restapiexample.com/api/v1/employees.

The Json starts with an Json Object which constitutes Json Array named "data". There are total of 24 records as the response of the get() hit request provides:

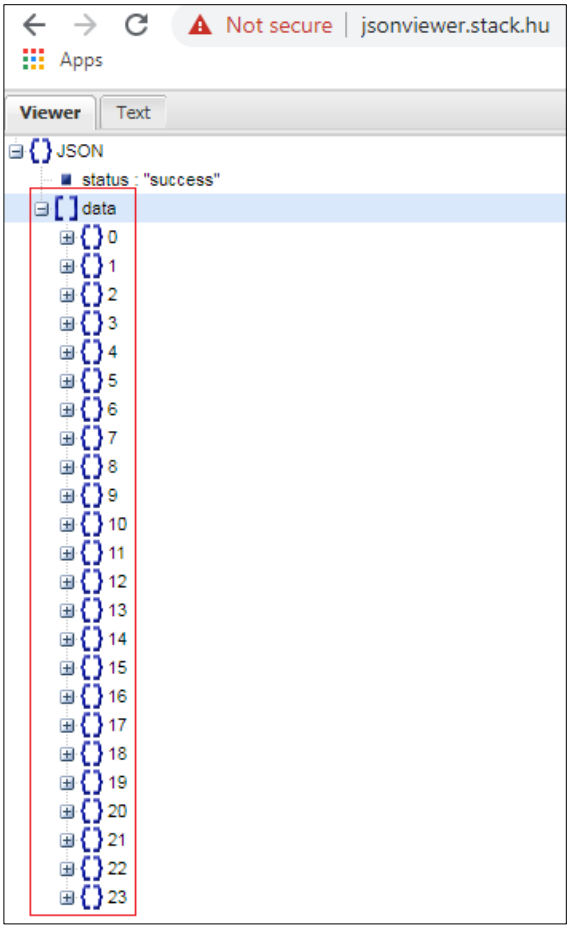



```
package API_Response_Parsing.API_Response_Parsing;

import com.jayway.restassured.http.ContentType;
import com.jayway.restassured.response.Response;
import static com.jayway.restassured.RestAssured.given;
import static com.jayway.restassured.response.Response.*;

public class Json_Array_Parsing
{
	public static void main(String[] args)
	{
		Response Res=
				given()
				.contentType(ContentType.JSON)
				.when()
				.get("http://dummy.restapiexample.com/api/v1/employees");
				System.out.println(Res.asString());
	}
}
```

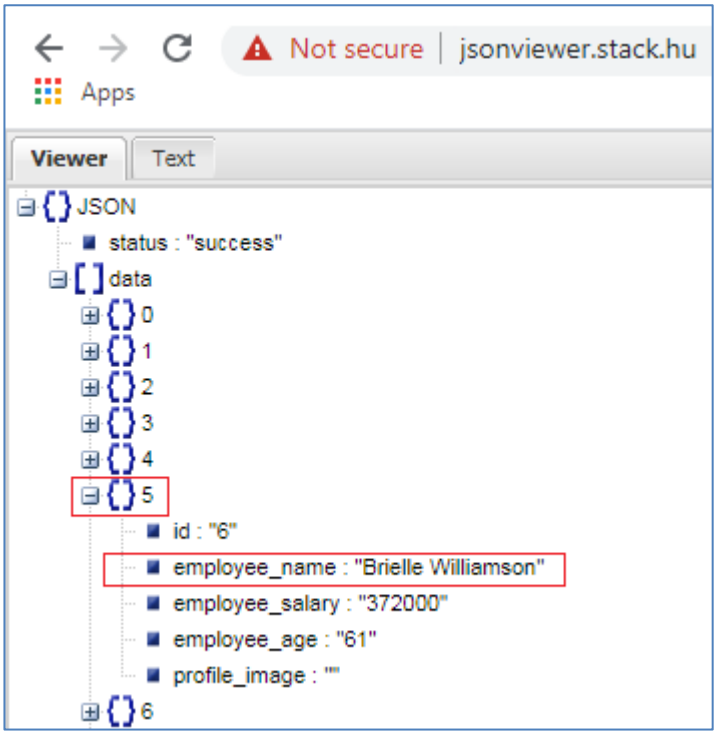
Copy the entire data from the console (output window) and paste in the Text tab at: http://jsonviewer.stack.hu/



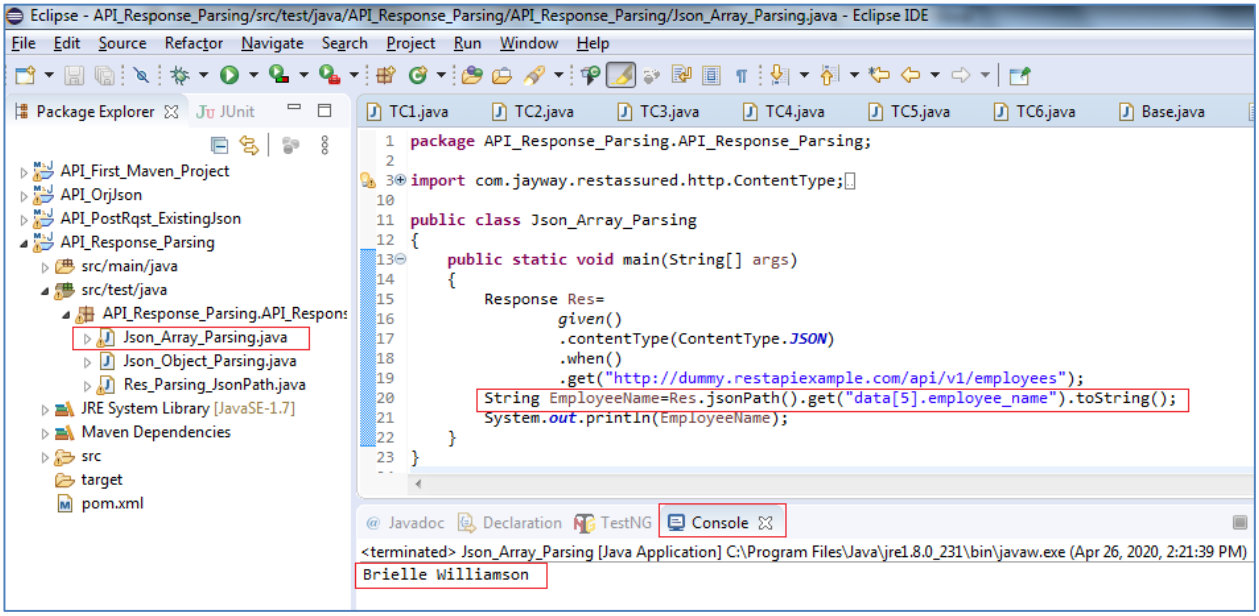
We can see that there is lots of response data retrieved out by this .get() hit request from the given uri. Just click on the View Tab at http://jsonviewer.stack.hu/ to watch the complete JSON data. And from the Json it is also very clear that the data is in the form of JSON Object that further contains JSON Array “data”



Q1: Write a program to parse response data Employee Name of the 6th Record using JsonPath and Org.Json library



I. Fetching the value of Key: employee name of 6th object from response data using JSONPath :



```
public class Json_Array_Parsing
{
	public static void main(String[] args)
	{
		Response Res=
				given()
				.contentType(ContentType.JSON)
				.when()
				.get("http://dummy.restapiexample.com/api/v1/employees");

String EmployeeName=Res.jsonPath().get("data[5].employee_name").toString();

System.out.println(EmployeeName);
	}
}
```

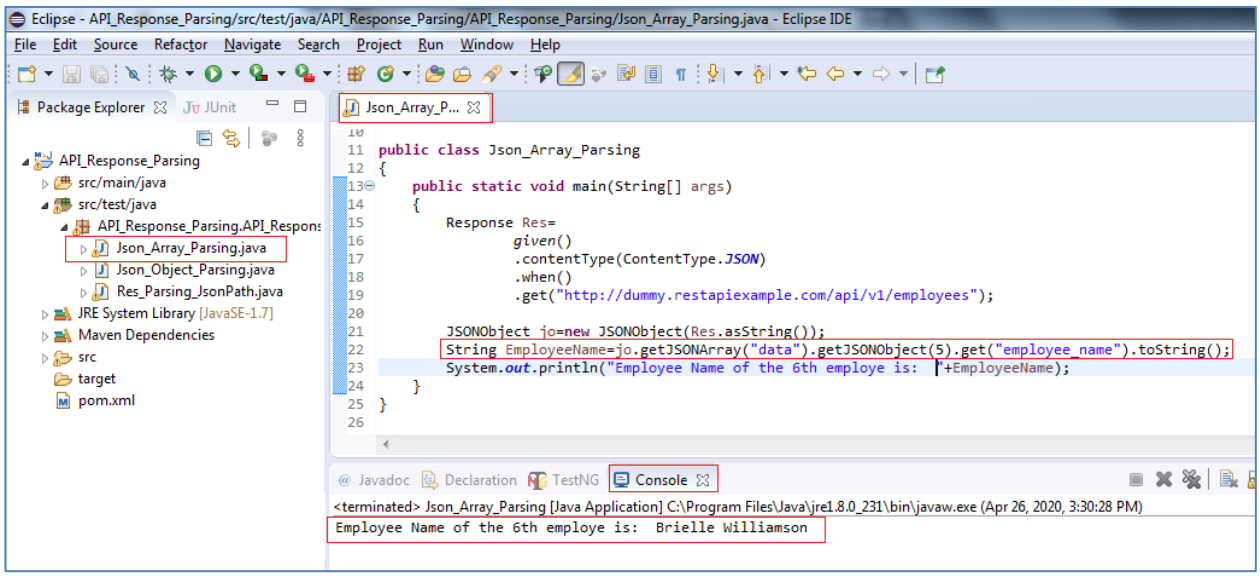
Explanation:

Rest of the program is very familiar and we will focus on the main instruction:

```
String EmployeeName=Res.jsonPath().get("data[5].employee_name").toString();
```

We know that our JSON starts with JSONObject and inside that we have the JSONArray "data" and we need to fetch the employee name of the 6th record so data[5].employee_name (Array index starts from 0) will give that particular data to us. We have converted the data into String with .toString() and .jsonPath() reaches to this data from Res.

II. Fetching the employee name of 6th record from response using Org.Json :



```
public class Json_Array_Parsing
{
	public static void main(String[] args)
	{
		Response Res=
				given()
				.contentType(ContentType.JSON)
				.when()
				.get("http://dummy.restapiexample.com/api/v1/employees");

		JSONObject jo=new JSONObject(Res.asString());

		String EmployeeName=jo.getJSONArray("data")
		.getJSONObject(5).get("employee_name").toString();

System.out.println("Employee Name of the 6th employe is:  "+EmployeeName);
	}
}
```

Explanation:

Technically, Our concern is to reach to the value of key: employeename of the 6th object in the Json response data. We are already aware that the Json response data is in the form og JSONObject do we have created one Object of JSONObject class and stored the entire JSON Response data in that with the below statement:

```
JSONObject jo=new JSONObject(Res.asString());
```

This is really very interesting example where we fetch the particular data out of the entire response data. So let's dissect the below statement (ignore the white spaces between the statements it is only the one single statement):

```
String EmployeeName=jo.getJSONArray("data")
	.getJSONObject(5).get("employee_name").toString();
```

To reach to the JsonArray "data" inside the JSONObject data which we got from Res.asString() method we have to use:

```
jo.getJSONArray("data")
```

further to that we want to access 6th JSONObject and further its key "employee_name" therefore the below statement servers out this purpose:

```
.getJSONObject(5).get("employee_name").toString();
```

Finally, we have changed the fetched data to type string and is stored in the String variable EmployeeName:

```
String EmployeeName=
```

Q2: Write a program using Org.Json to parse response data where only one key value of every record is fetched and displayed

```
import static com.jayway.restassured.RestAssured.given;
import static com.jayway.restassured.response.Response.*;
import org.json.JSONArray;
import org.json.JSONObject;
public class Json_Array_Parsing
{
    public static void main(String[] args)
    {
            Response Res=
                  given()
                  .contentType(ContentType.JSON)
                  .when()
                  .get("http://dummy.restapiexample.com/api/v1/employees");

            JSONObject jo=new JSONObject(Res.asString());

            for(int i=0; i<jo.getJSONArray("data").length();i++)
                  {
                        String Record=jo.getJSONArray("data").getJSONObject(i)
                        .get("employee_name").toString();
                        System.out.println(i+"   "+Record);
                        }
    }
}
```

Explanation:

We have fetched the data from the Uri : http://dummy.restapiexample.com/api/v1/employees as provided in the .get() request. We have seen that the data is in the form of JSONObject and need object of the JSONObject class to play around the response data and so is the below statement where we are storing the entire response on JSONObject jo:

```
JSONObject jo=new JSONObject(Res.asString());
```

What if our data starts with a JSON Array? Then we have to have store the response that in the object of JSONArray class something like this:

```
JSONArray ja=new JSONArray(Res.asString());
```

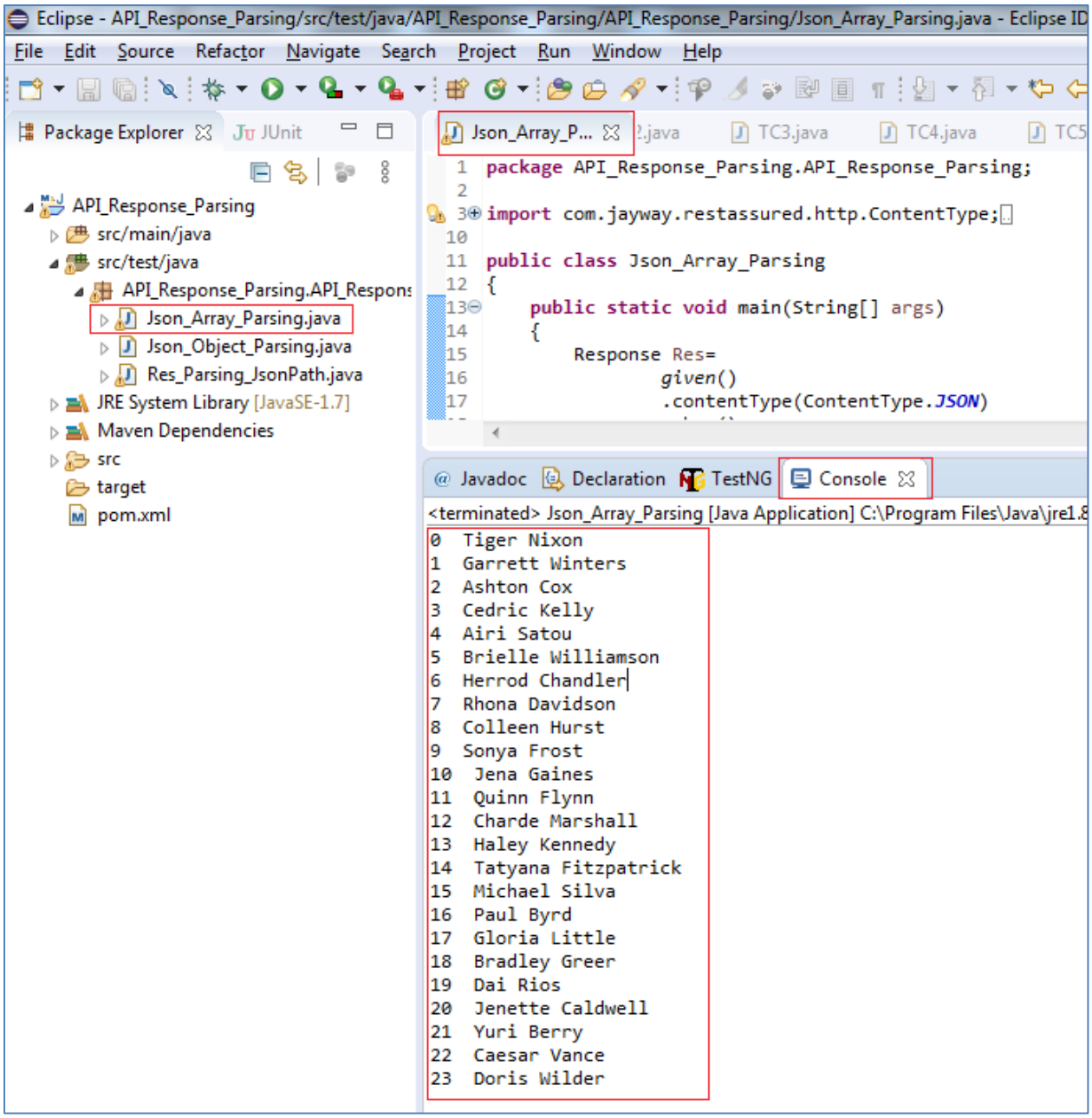
Now please make it very clear understanding in mind, it doesn't matter in context to the concept of response parsing whether the data is in the form of JSONObject or JSONArray we have further methods like .getJSONObject(), .getJSONArray() further to parse our target data.
Next to that we have find the length of the total number of objects in a response data by the statement:

```
jo.getJSONArray("data").length();
```

and we have utilized this lenght to read each of the object and reach to the target data we want to fetch from the entire response data using the for loop:

```
String Record=jo.getJSONArray("data").getJSONObject(i)
                  .get("employee_name").toString();
System.out.println(i+"   "+Record);
```

Please note the output of the data in the console below:



Chapter- 8: Designing the Rest Assure Framework Project

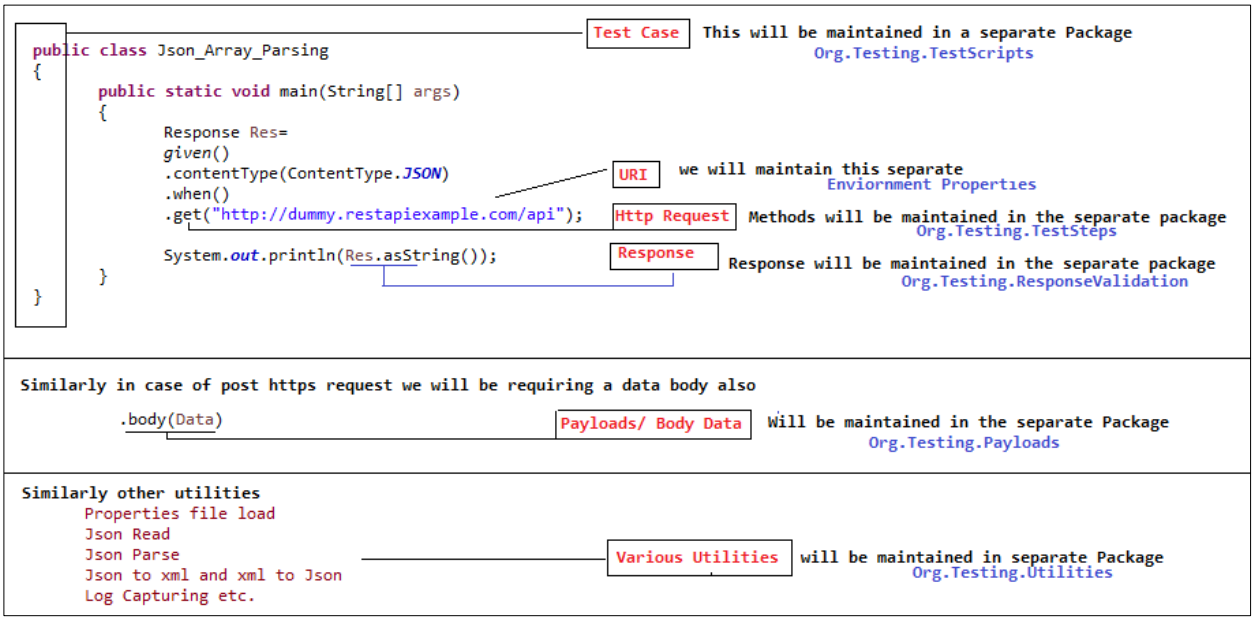
Part- I: Designing the Framework Wire Frame

Framework:

The best way to design a framework is to think in a reverse direction that is not from steps to result but think of result first and then plan the steps required further, segregate and arrange the steps accordingly. Framework is organizing the project files strategically in order to increase the modularity, reusability, debugging, sharing, enhance performance and finally to reduce the overall complexity.

It is dividing the program into different packages where each package will be responsible for its own function. Packages may or may not dependents upon each other and this is purely depends upon the type of functionality and project in hand.

Let's see the simple https get() request program:



Diagrammatic Understanding the Framework:

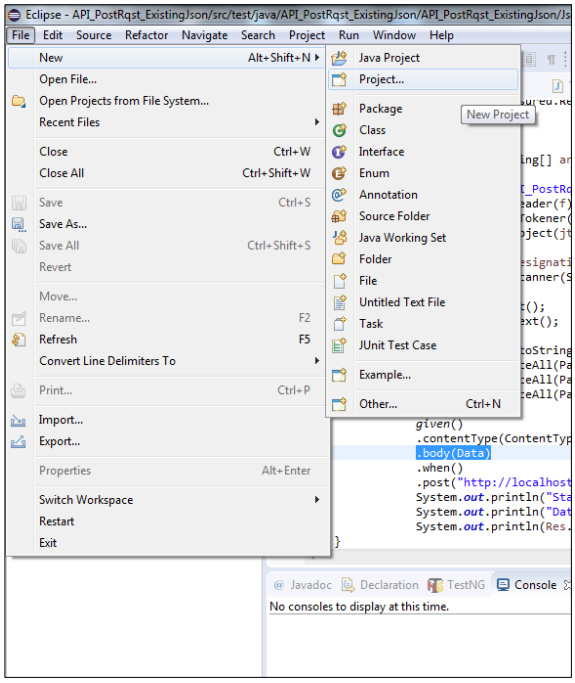
Every organization has their own experts and policies, the framework presented here or any framework in any organization is not a benchmark or hard-n-fast to be followed everywhere in the industry. But all-in-all the basic structure of framework will remain common and the primary objective behind the framework will also remain the same i.e. to increase the modularity in order to facilitate debugging and sharing the units.

All the test cases will be Maintained here org.testing.TestCases	All the Payloads will be Maintained here org.testing.Payloads	Orgnaising Test Case Execution Parameters in Excel
All the http Methods will be Maintained here org.testing.TestSteps	All the test Utilities will be Maintained here org.testing.Utilities	
All the Responses will be Validated here org.testing.ResponseValidation	Trigger Class Contains Sequence of Execution of Test Cases org.testing.Trigger	All URI's are Maintained here Env.properties

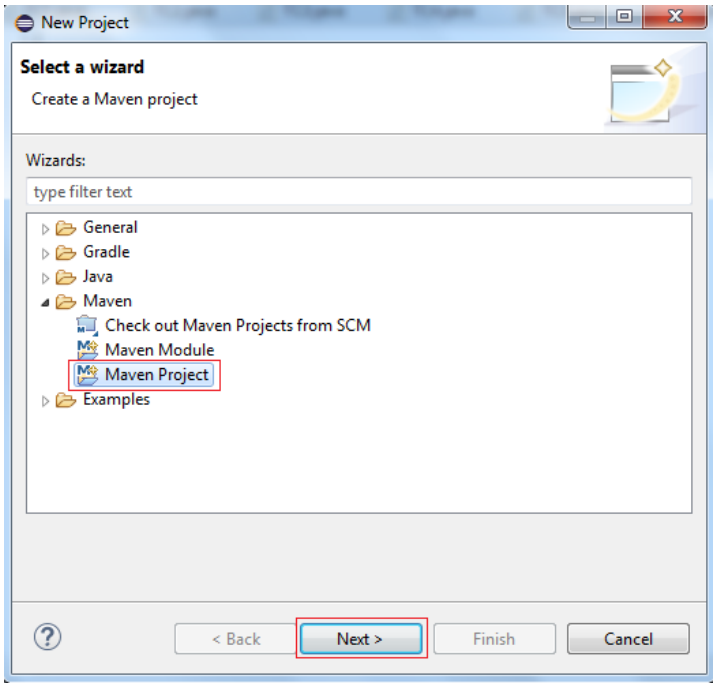
Chapter-8 Table 1

Creating the Framework Project (Maven)- The Wireframe

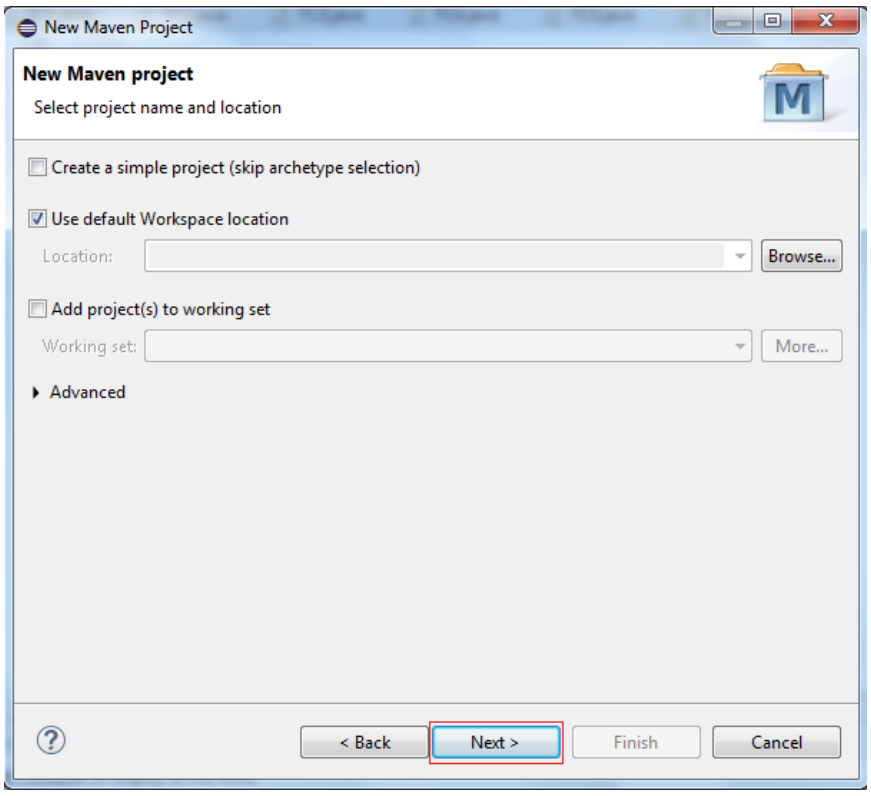
1. Create one maven Project say: RestArruredFrameWork_Project



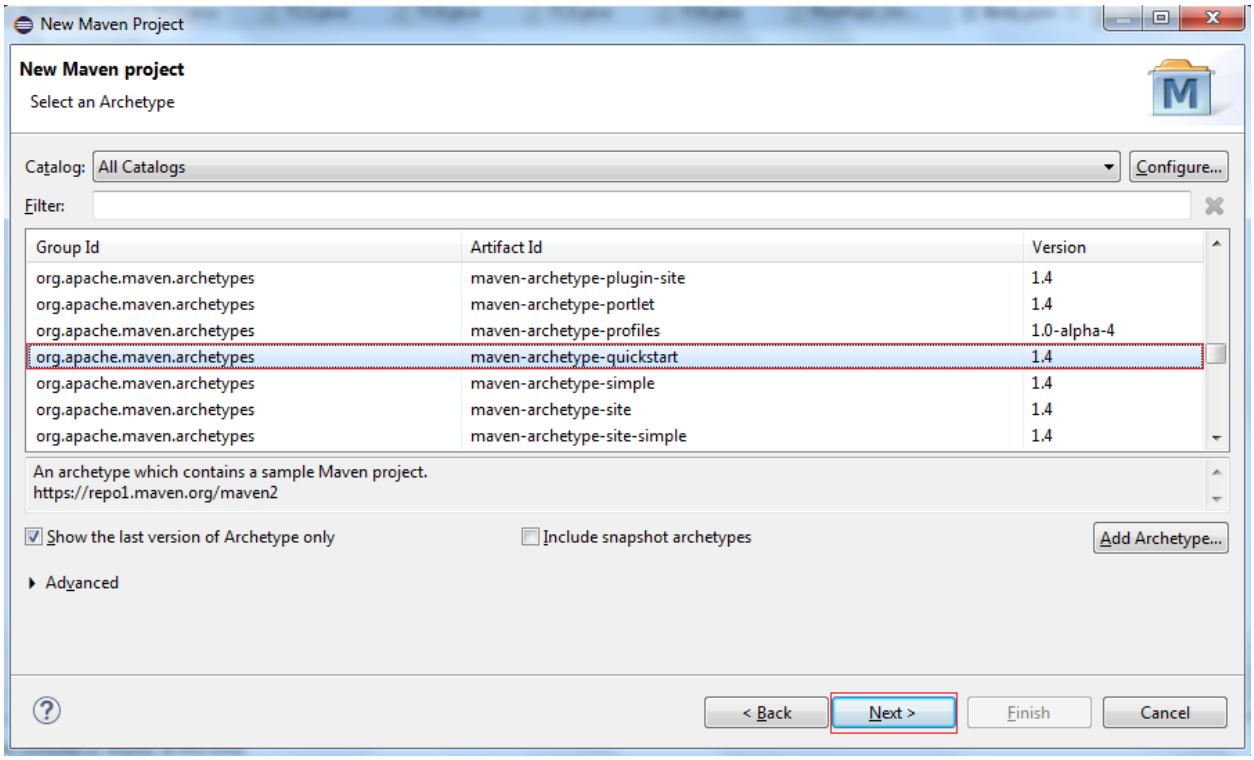
2. New Project window appears , select Maven Project and on Next button



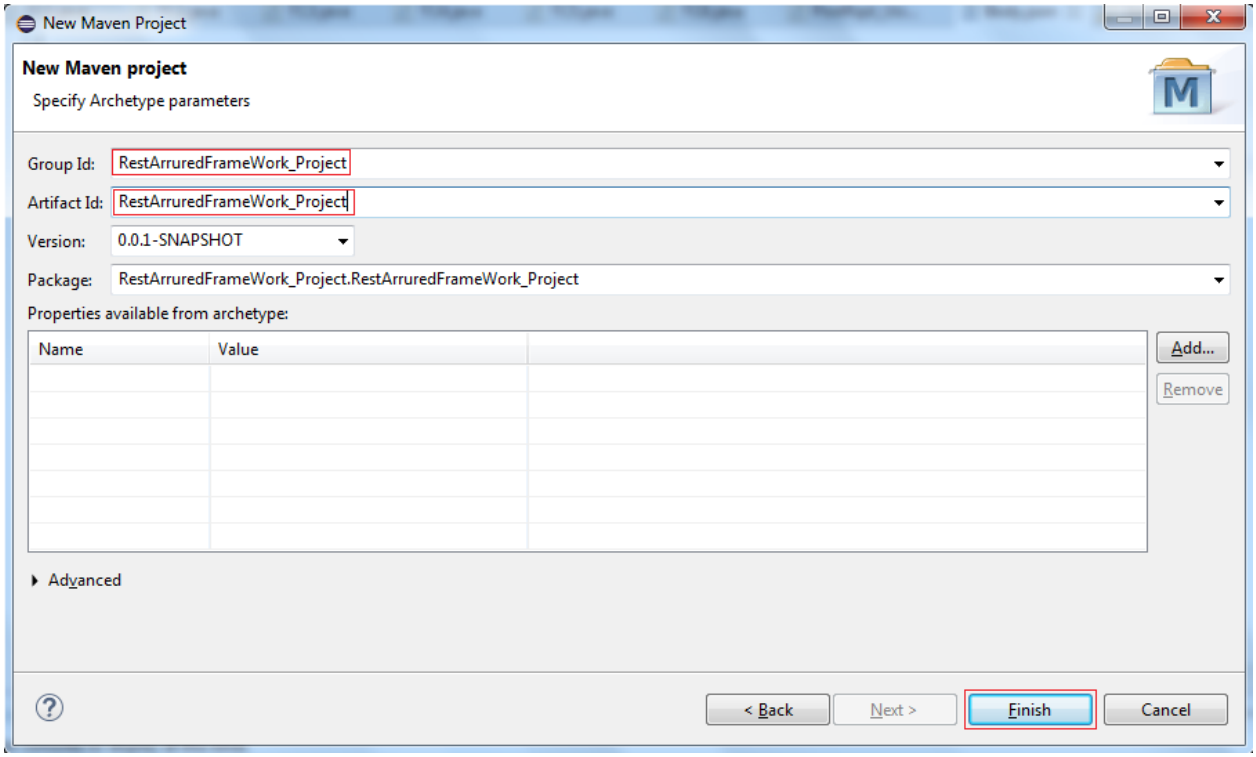
3. New Maven window continues, Simply Click Next Button



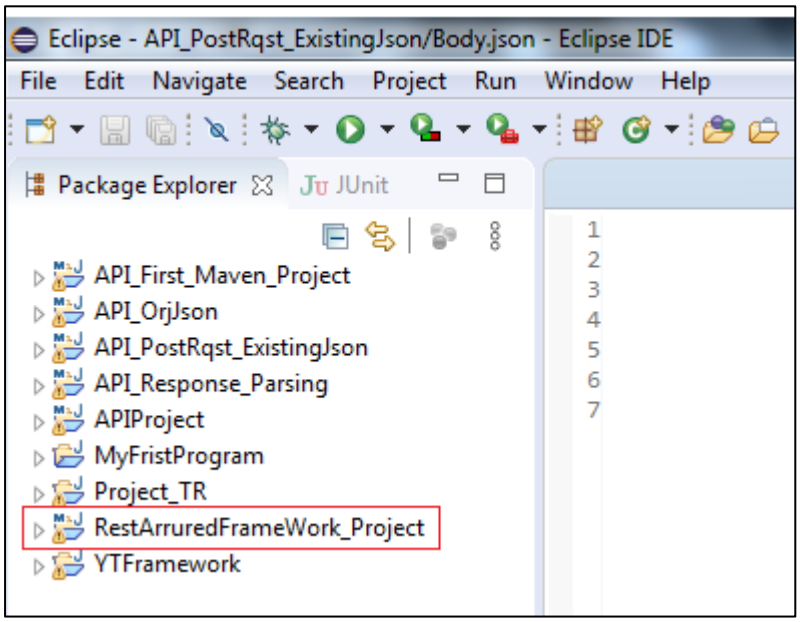
4. Select an Archetype and click in next button



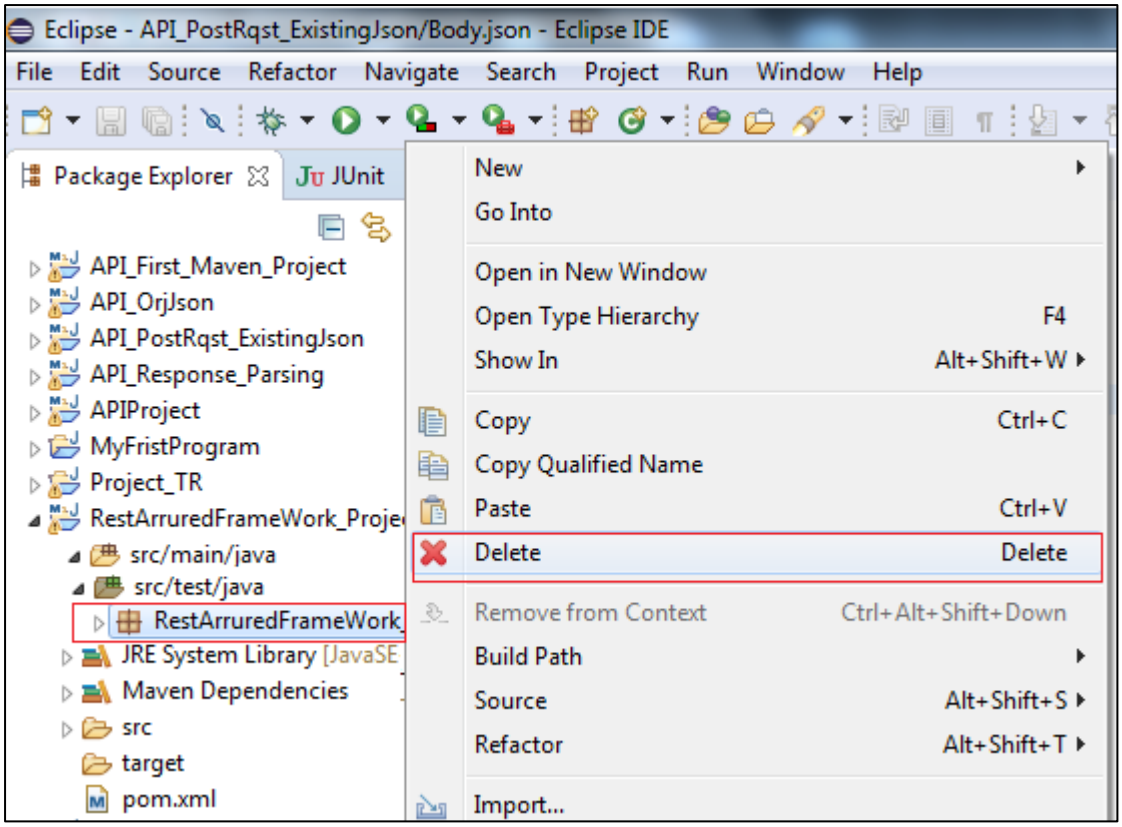
5. Specify ArchTpe Parameters Group Id as: RestAssuredFrameWork_Project and Artifact Id as: RestAssuredFrameWork_Project (You can give any name to both the Id's) and click on Finish Button



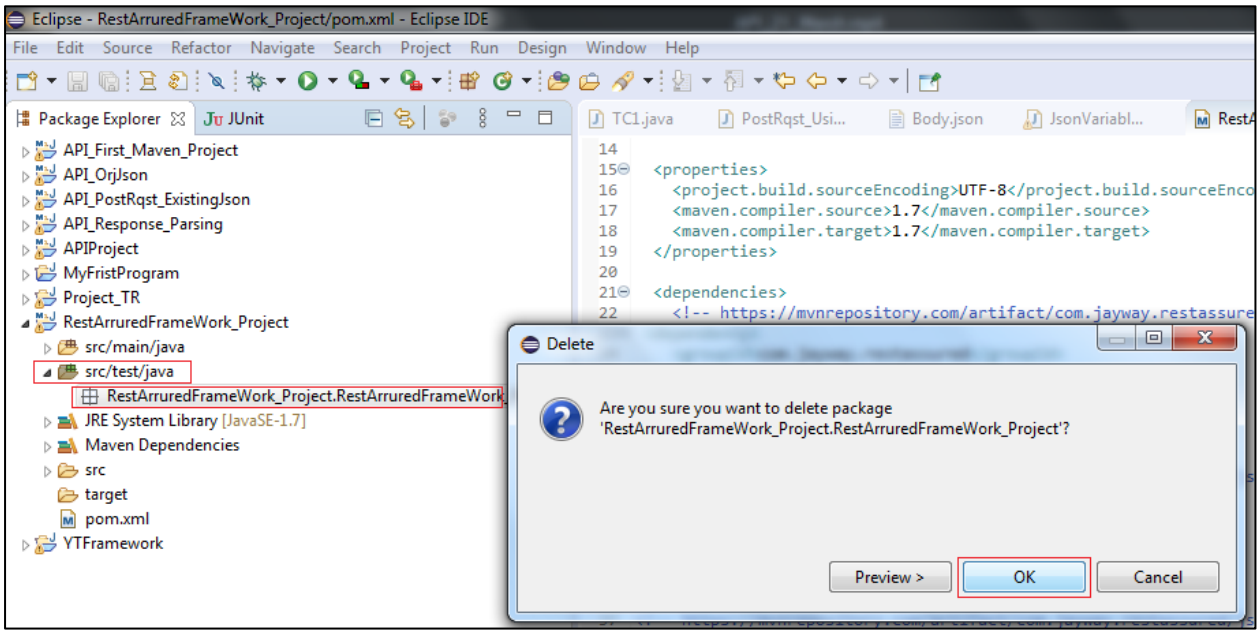
6. A new maven project named RestAssureFrameWork_Project is created successfully.



7. Similarly also delete the default package RestAssuredFrameWork_Project. RestAssuredFrameWork_Project created by java in the project under src/test/java



8. Confirmation message box appears, simply click on OK button

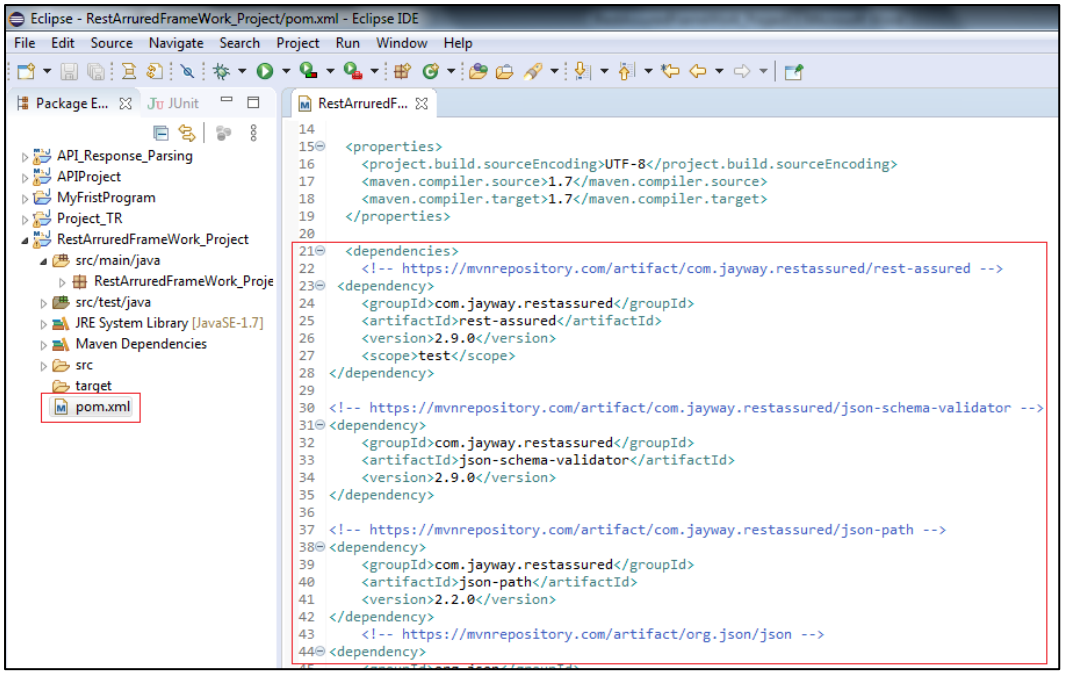


9. The first ever thing is to include Maven dependencies and we achieve that by pasting the following dependencies in our respective project pom.xml under the dependencies tag:

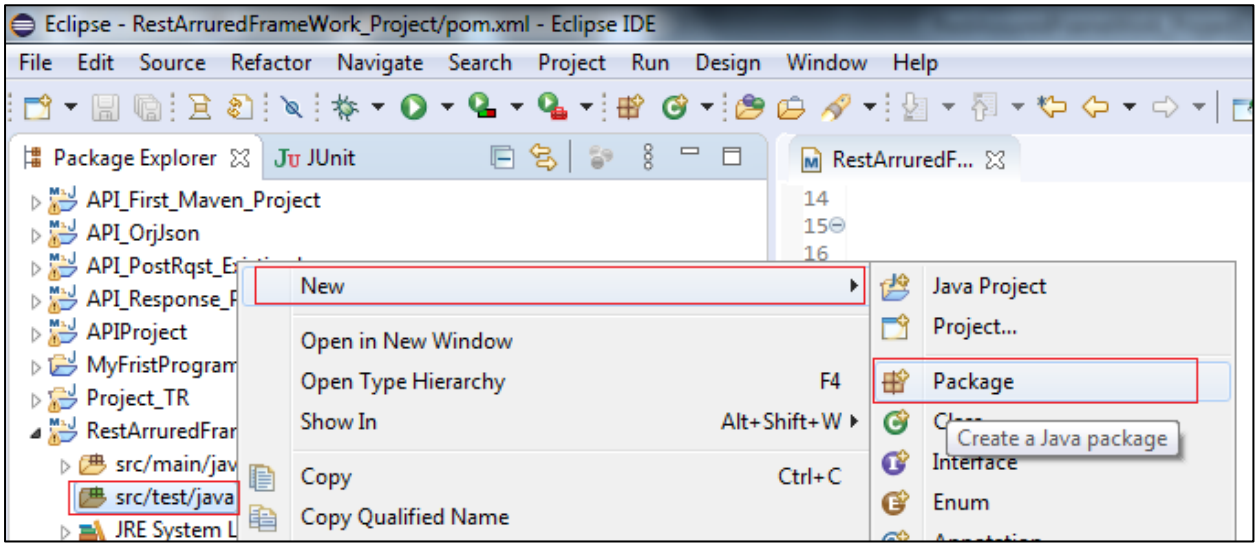
Including The Required REST Assured And Other Dependencies:

a. REST Assured- https://mvnrepository.com/artifact/io.rest-assured/rest-assured
b. TestNG- https://mvnrepository.com/artifact/org.testng/testng
c. JSON- Simple- https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple
d. Apache POI- https://mvnrepository.com/artifact/org.apache.poi/poi
e. Apache POI- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml

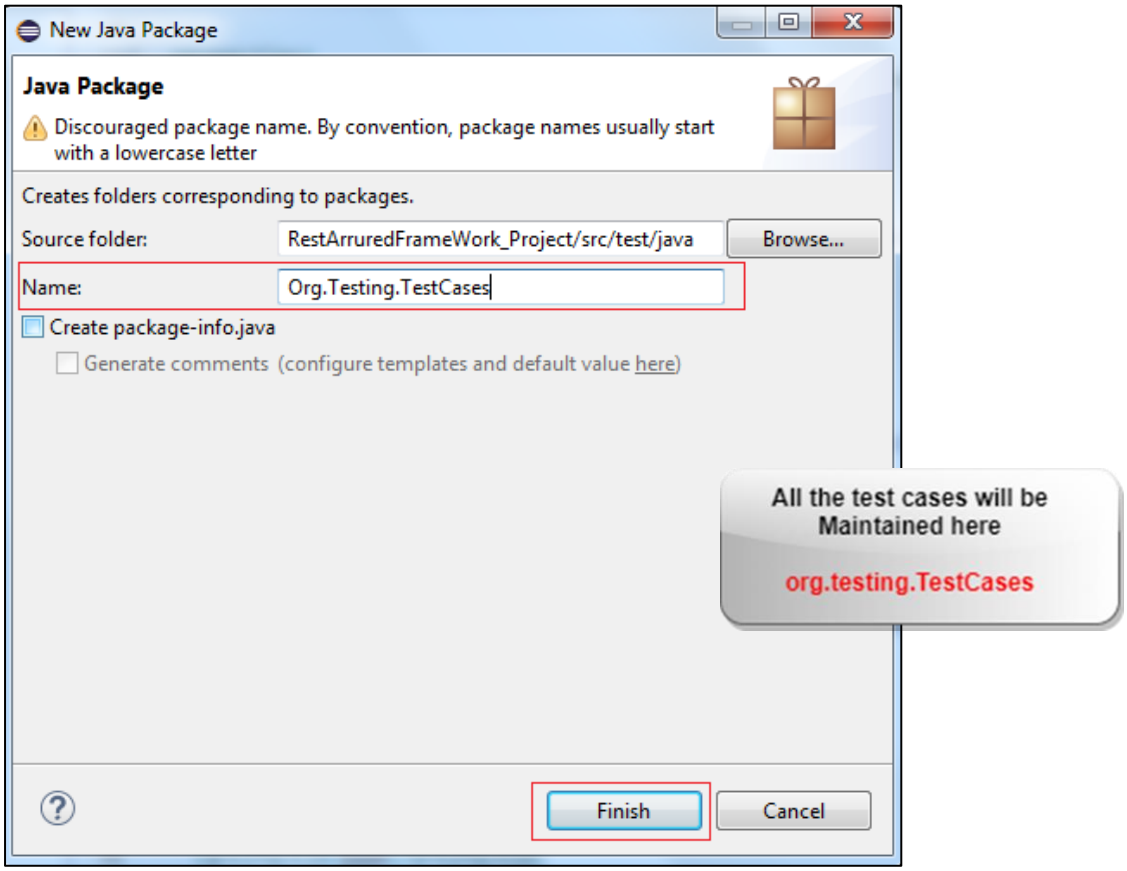
After including the same out pom.xml will look something like as shown below:



10. As we have discussed earlier we will be categorizing and placing every task to be performed in a Within a test case in different packages so, first create the packages:



Enter the Name of the Package Org.Testing.TestCases and click on Finish button.

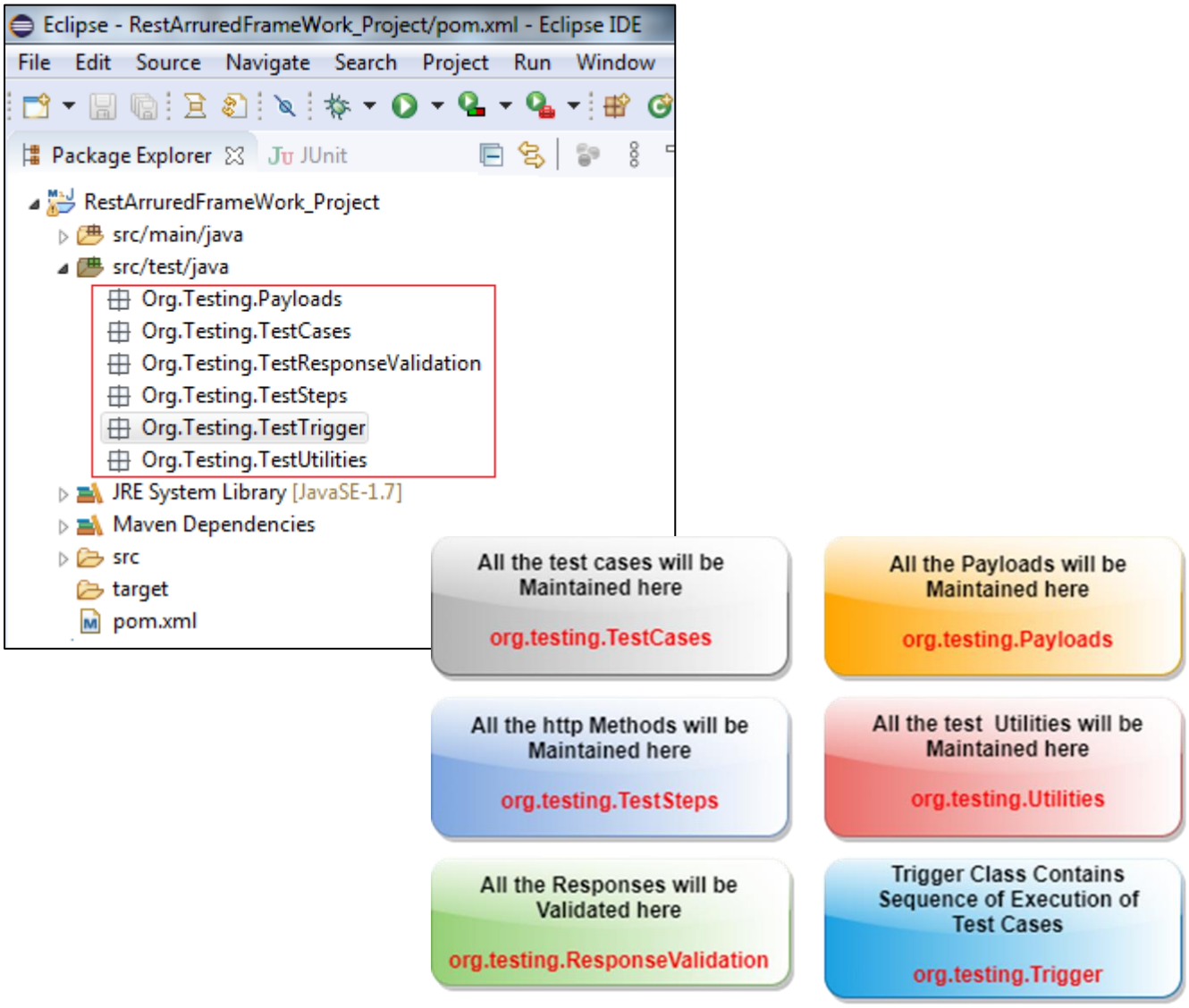


Similarly, we are planning to create the following Packages along with their responsibilities:

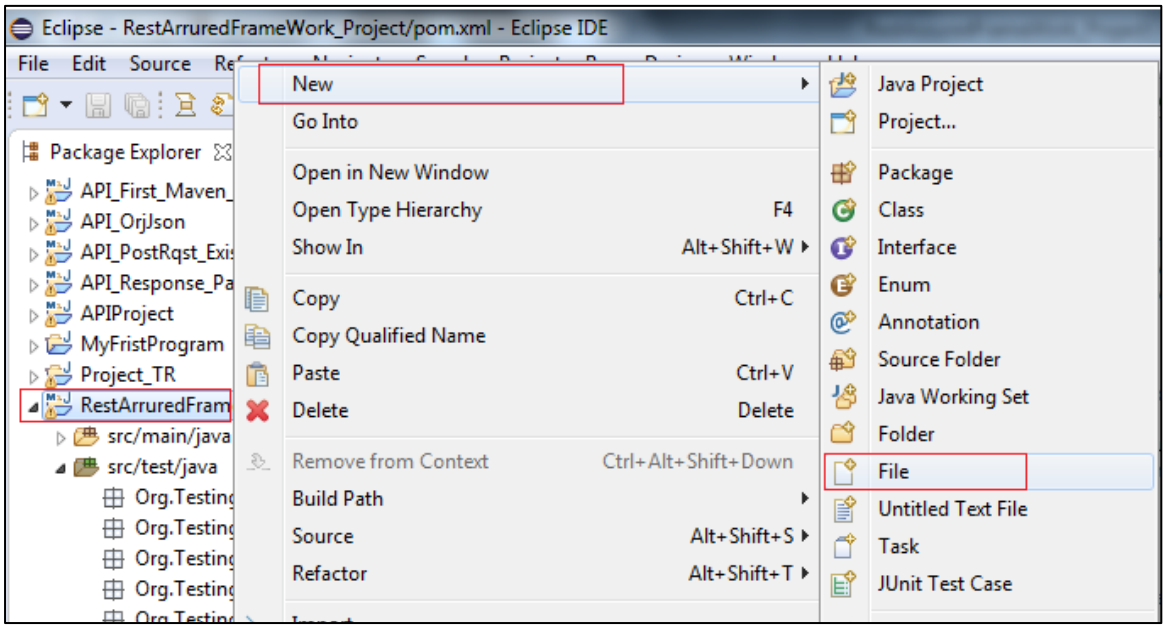
Package Name	Responsibility
Org.Testing.TestCases	Where all the test cases will be maintained
Org.Testing.TestSteps	Where all the https requests will handled
Org.Testing.TestUtilities	Where all the tasks like file reading/writing, parsing etc will be handled
Org.Testing.Payloads	Where all the JSON Body data will be handled for POST,PUT and PATCH https requests.
Org.Testing.TestResponseValidation	Where all the a Response validations will be performed
Org.Testing.TestTrigger	Will trigger the Tests
Env.Properties file	Will hold all the URI's in the form of *Key:Value*
Other file like Existing Json	May or may not be the part framework

Chapter-8 Table 1

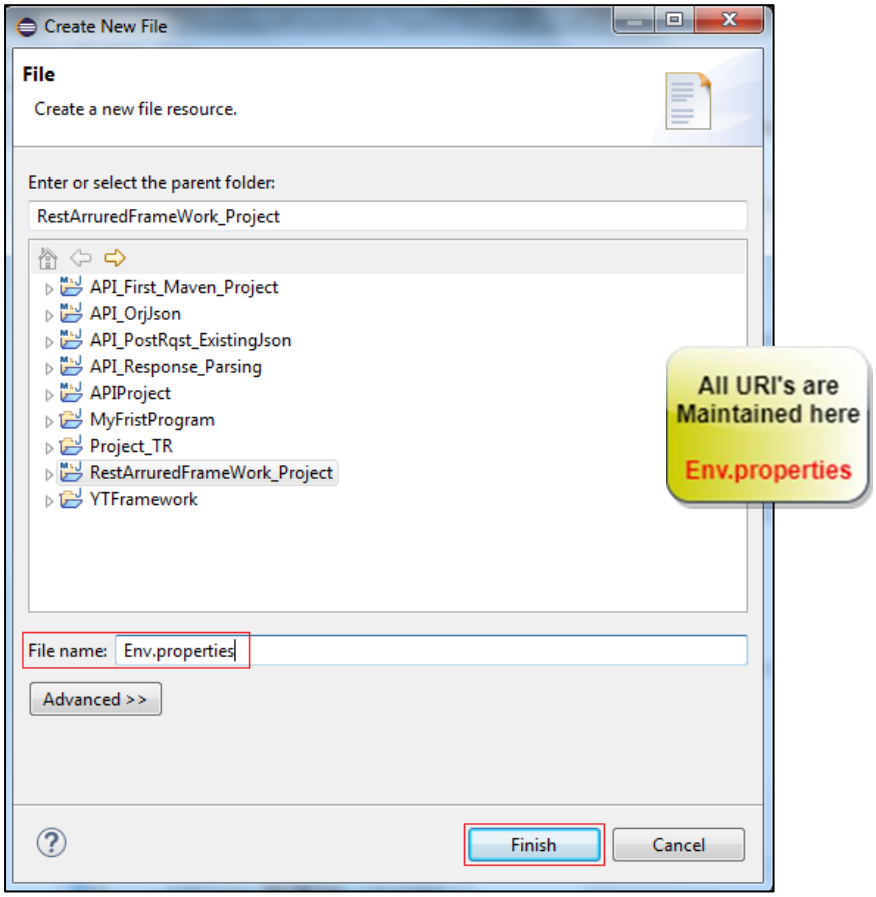
After creating all the packages the package structure under /scr/test/java will look as below:

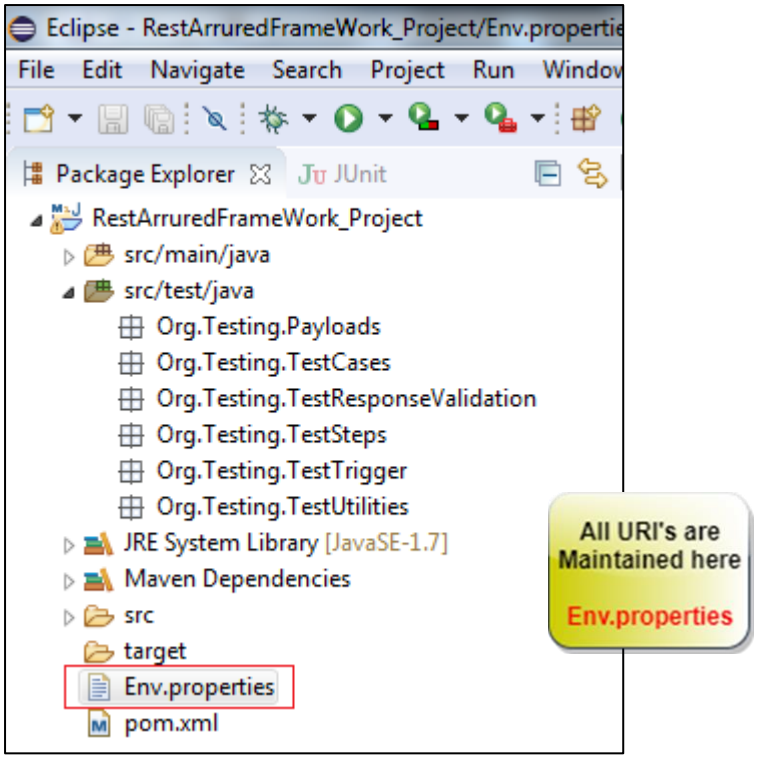


11. Right click on project and create one properties file say Env.properties (please note that the extension of this file must only .properties only)



Enter the file name Env.propeties and click on Finish button



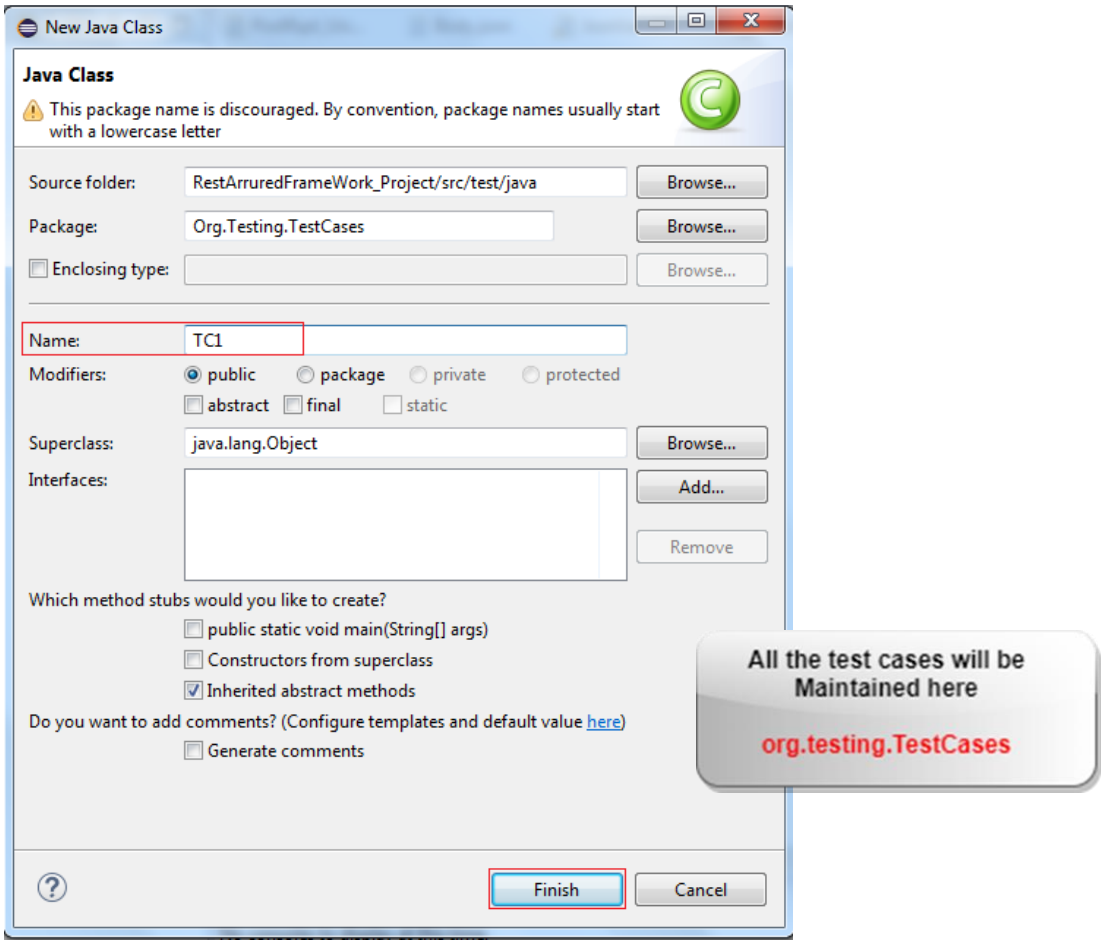


So we all set with our frame work design for at least handling the get() http request.

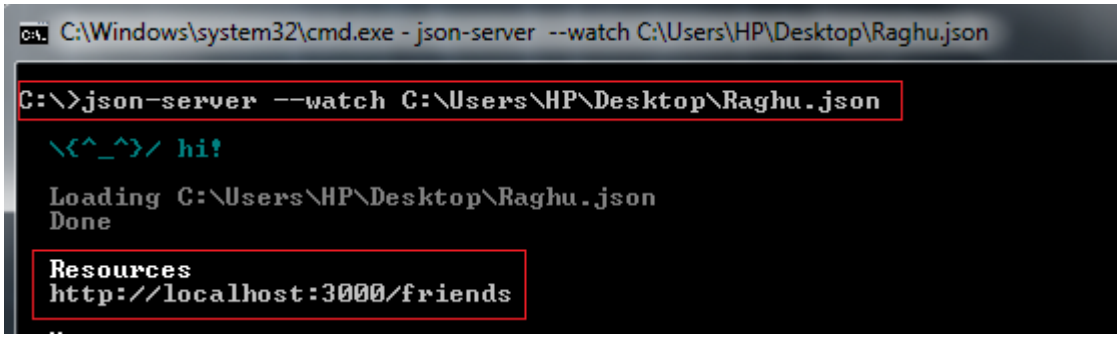
Now We Will Start With Writing Up Test Cases: TC1: To handle the get() Request

	Http Request	Description
1.	**Get**	No data is required for this request. it is used to fetch the data from the given resource location.

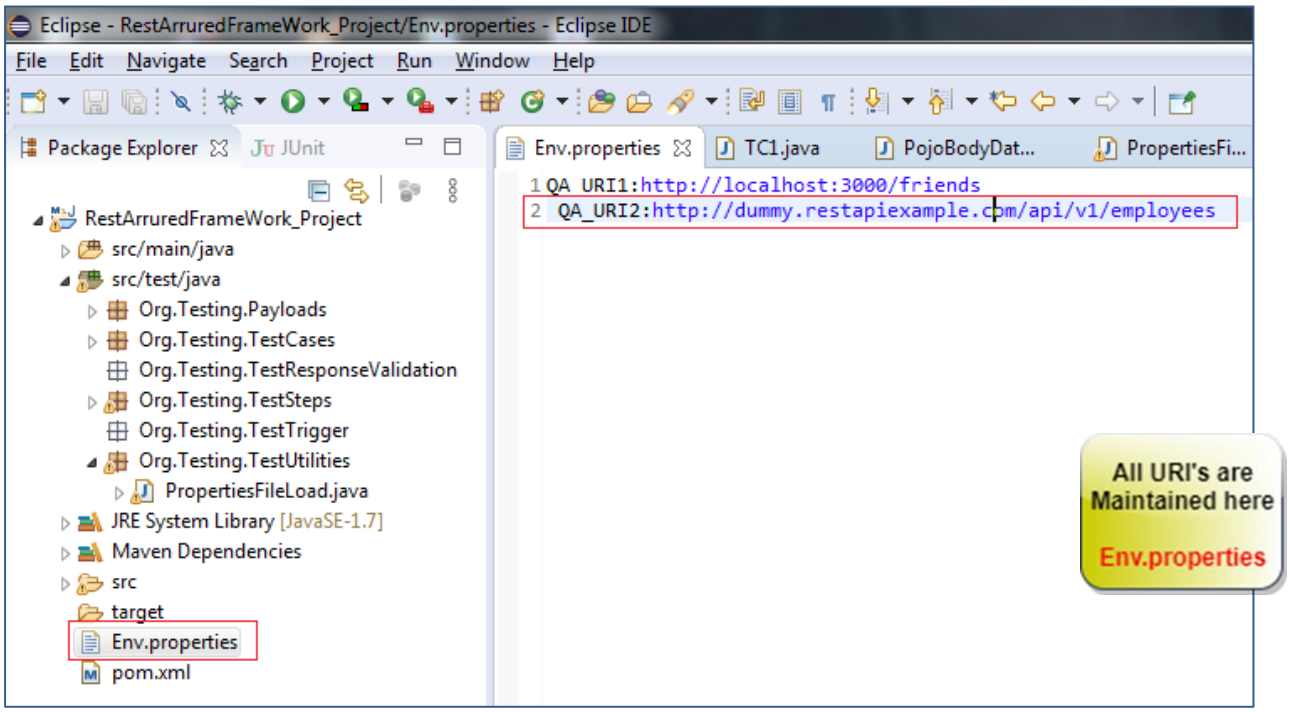
1. Create one class TC1.java under Org.Testing.TestCases package and click on Finish button as we have already decided to maintain all of our test cases under this package.



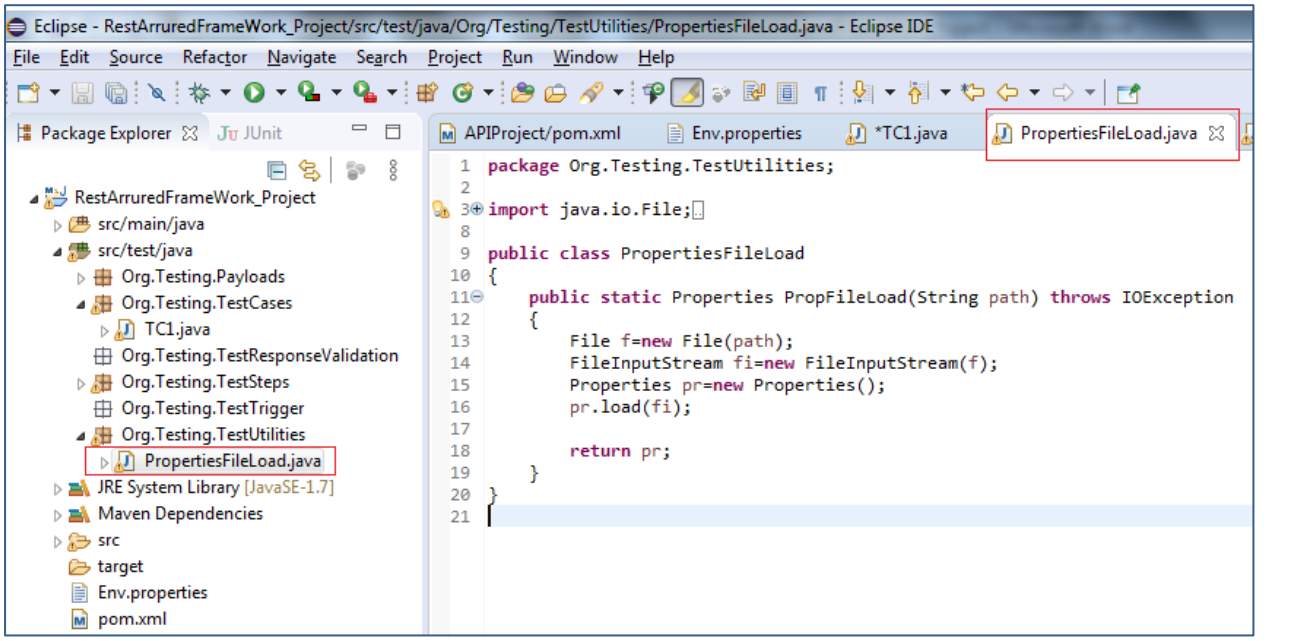
2. Start the json server and copy the URI displayed in your prompt : http://localhost:3000/friends



3. Open Env.properties file and save the copied URI in in the form of key:value
   QA_URI1: http://localhost:3000/friends
   QA_URI2: http://dummy.restapiexample.com/api/v1/employees



4. Create a class say PropetiesFileLoad under Org.Testing.Utilities package



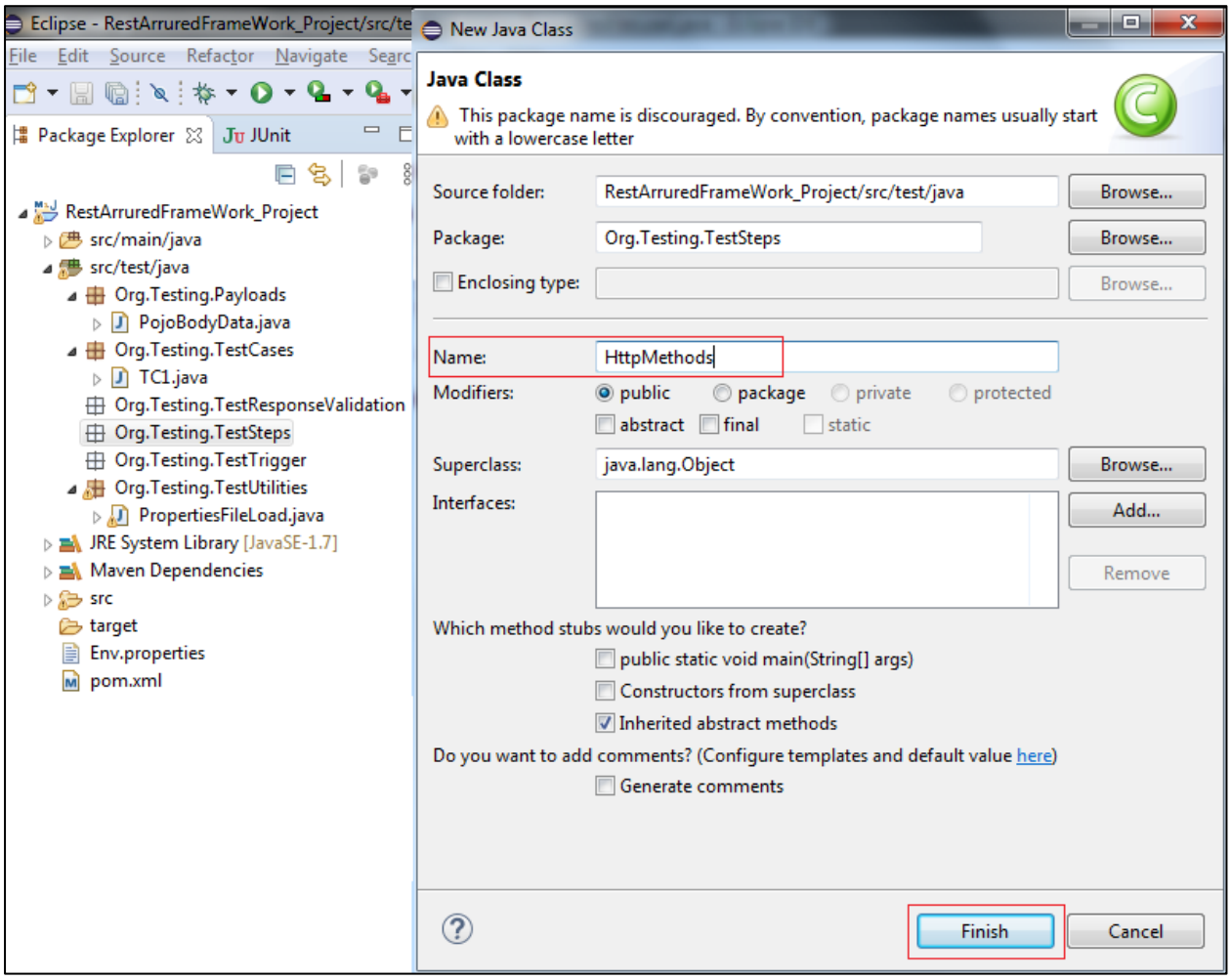
Explanation:

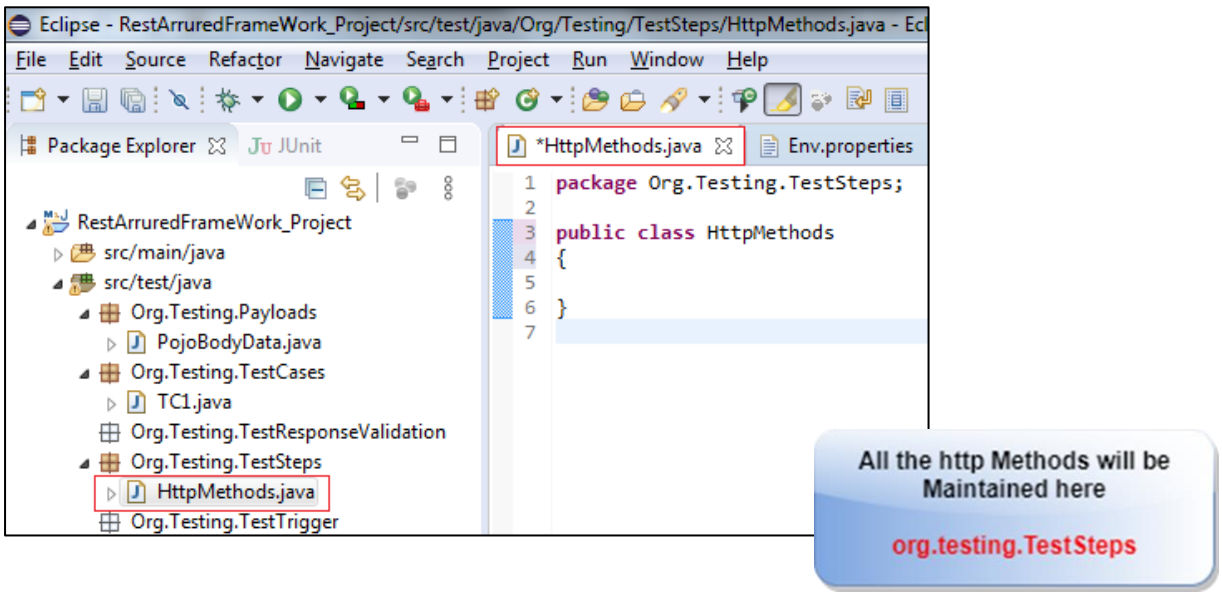
```
public class PropertiesFileLoad
{
	public static Properties PropFileLoad(String path) throws IOException
	{
		File f=new File(path);
		FileInputStream fi=new FileInputStream(f);
		Properties pr=new Properties();
		pr.load(fi);

		return pr;
	}
}
```

We have created a method named PropFileLoad having path as a string argument which will hold path passed to it and return type of this method is of type properties as it is going to return the object of properties class. This is the path of Env.properties file at the moment which will be passed through the TC1 from the calling function.

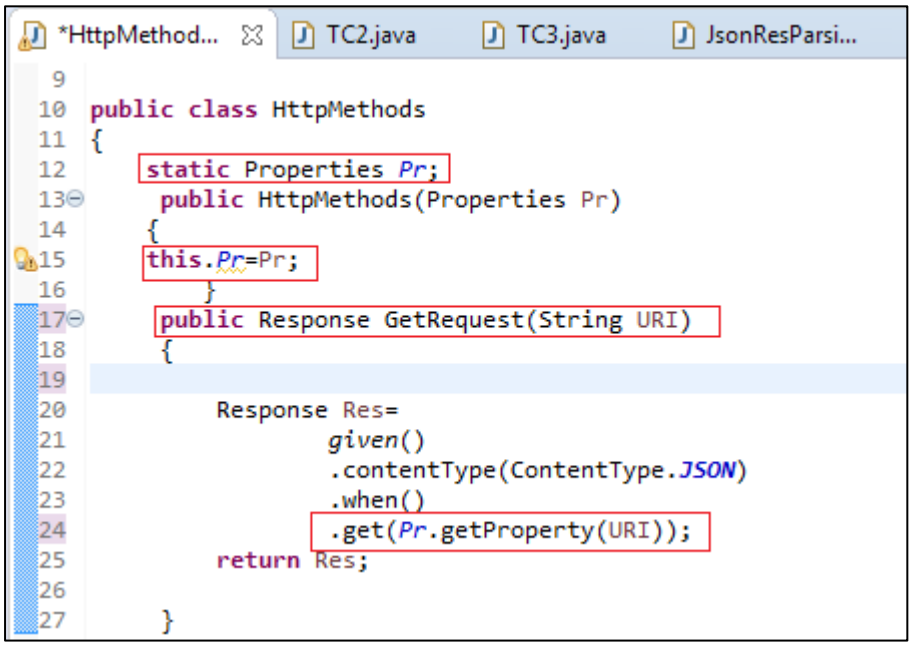
Then we have create a connection to the file `File f=new File(path);` and further to that object of FileInputStream is created for this file so as to read it. Object of properties file is created and the propertiese file is loaded `pr.load(fi);`
Finally, this object will be returned to the calling function `return pr;`

5. Create class named HttpMethods.java under Org.Testing.TestSteps:





Under this class create one method GetRequest() which accepts URI as an argument and will get the property of that URI from the Evn.properties file. At last this method will return the object of the Response class to the calling function from the respective Test Case.



Explanation:

```
public class HttpMethods
{
    static Properties Pr;
     public HttpMethods(Properties Pr)
    {
    this.Pr=Pr;
           }
     public static Response GetRequest(String URI)
     {
            Response Res=
                        given()
                        .contentType(ContentType.JSON)
                        .when()
                        .get(Pr.getProperty(URI));
            return Res;

     }
}
```

As per our framework design plan we will be having HttpMethods class under Org.Testing.TestSteps package that will maintain all the http method definitions and these methods will be called from various test cases residing under the Org.Testing.TestCases package.

Understanding the code deeply, just forget for an instance from where the method

```
public static Response GetRequest(String URI)
```

is being called, we will just focus at the moment what it is exactly doing. As we know that any method have four basic parts: method return type, method name, sequence of arguments it takes and body (set of instructions performing some task)

So, at a first glance it is very clear that method have following:

- Method Return type: Object of Response class
- Method Name: Get()
- Method Arguments: One argument of type String
- And finally Body:

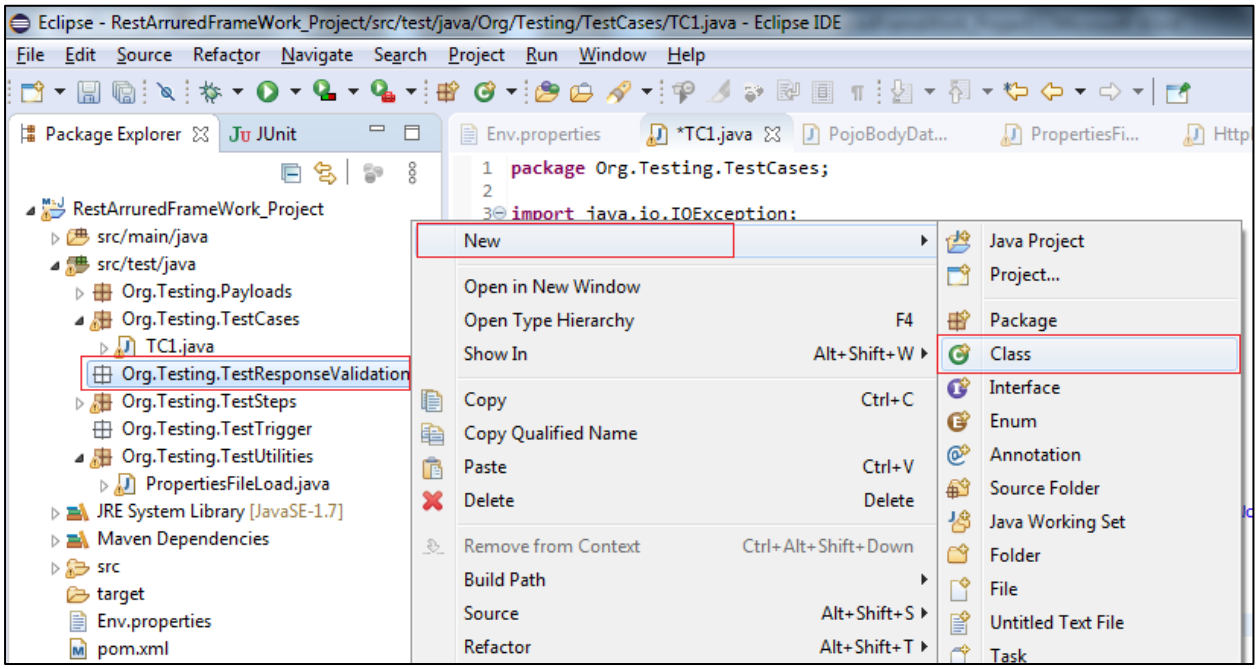
```
Response Res=
                        given()
                        .contentType(ContentType.JSON)
                        .when()
                        .get(Pr.getProperty(URI));
          return Res;
```

Storing the Response of the `.get(Pr.getProperty(URI));` statement in Res Object of Response class. This statement is also retrieving the value of URI string from the Env.properties file with its file handle `Pr` which is an object of Properties class. That's the reason `Pr` is initialized with its constructor at the first.

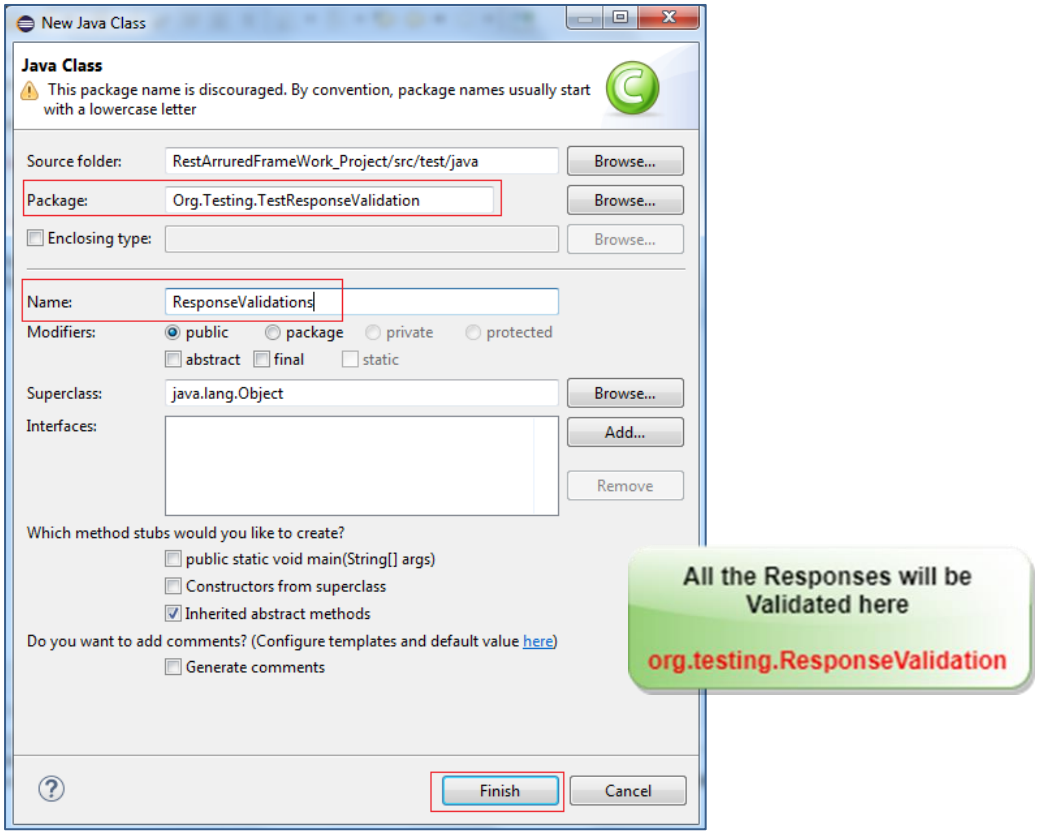
```
static Properties Pr;
    public HttpMethods(Properties Pr)
   {
   this.Pr=Pr;
         }
```

Being `Pr` declared as static object of Properties class, we are assigning it a value through its parameterized constructor `public HttpMethods(Properties Pr)` using this (`this.Pr=Pr;`) keyword and ultimately it will be updated by the every test case calling it according to Test case's own requirement.

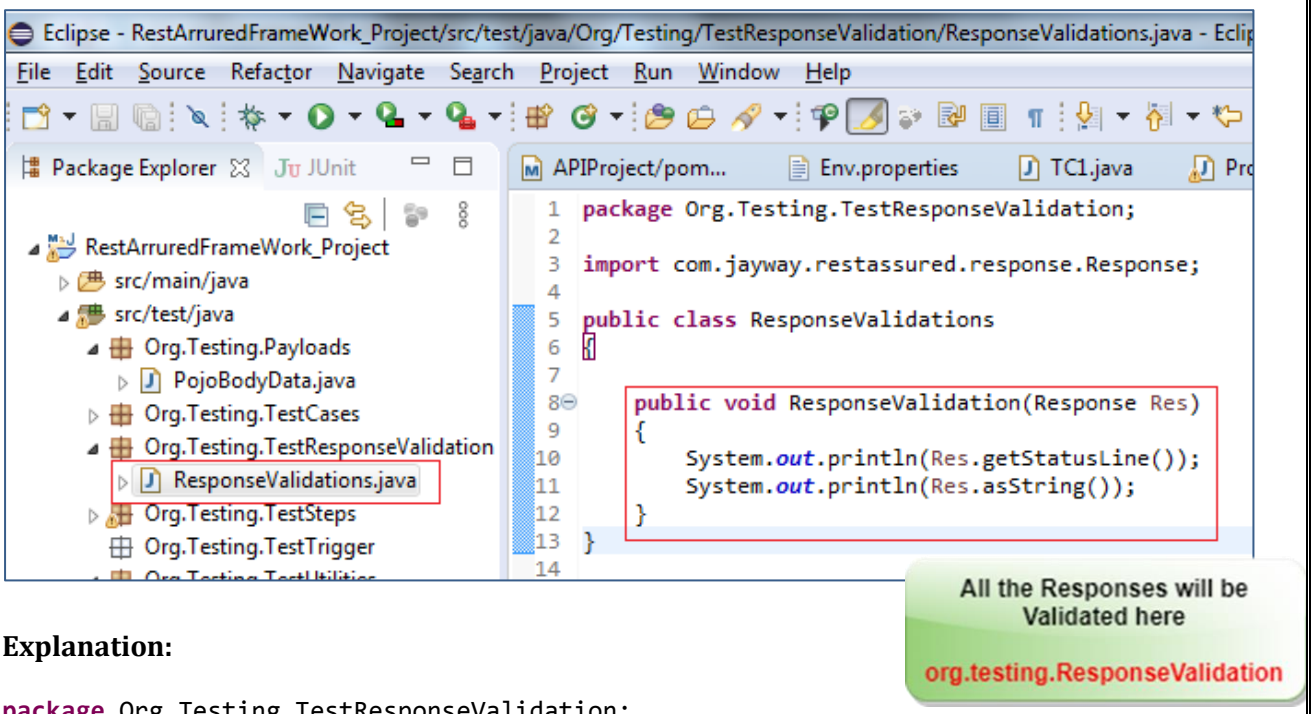
6. Create a class for Response validation say ResposeValidate.java under the package Org.Test.TestResponseValidation



Click on Finish button



7. Write the following code in the class created above:



Explanation:

```
package Org.Testing.TestResponseValidation;
import com.jayway.restassured.response.Response;

public class ResponseValidations
{

    public void ResponseValidation(Response Res)
    {
           System.out.println(Res.getStatusLine());
           System.out.println(Res.asString());
    }
}
```

We have planned the Response Validation is such a way that the response validation will be done in a class ResponseValidations

```
public class ResponseValidations
```

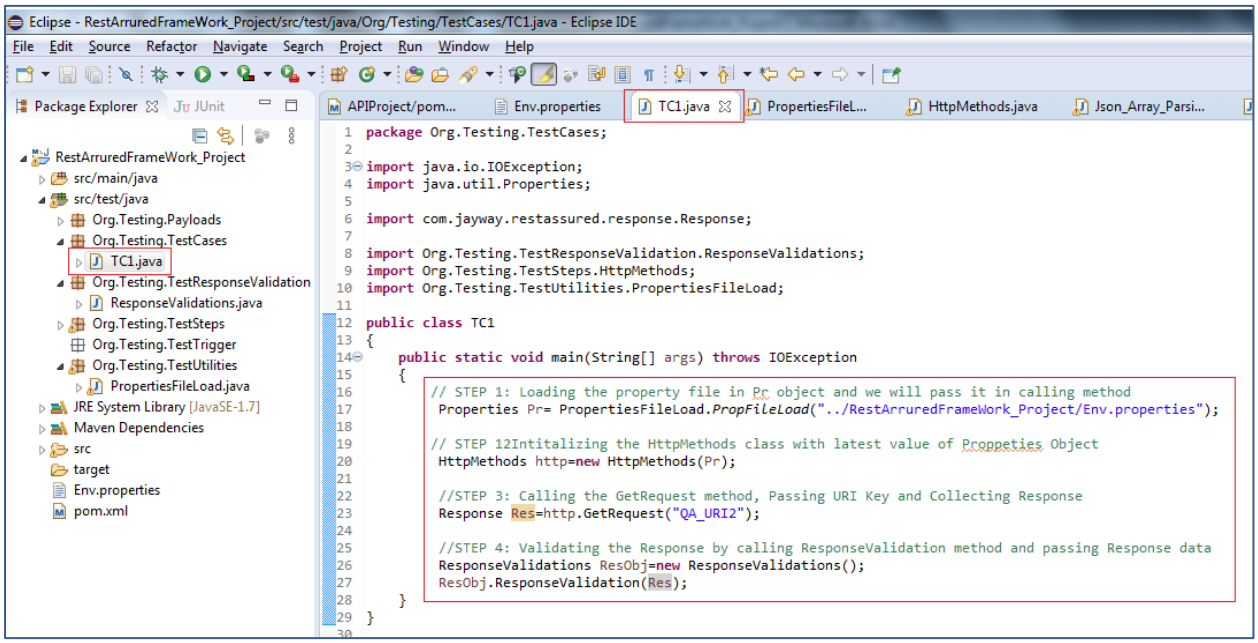
We have created one method under this class which accepts an argument Res of type Response

```
public void ResponseValidation(Response Res)
```

Finally, we print the Status line and response data on console using

```
Res.getStatusLine()
Res.asString()
```

8. Now so far we have created and handled 4 different things in four different packages, lets create a Test Case for get() http request using QA_URI2



Explanation:

```
package Org.Testing.TestCases;

import java.io.IOException;
import java.util.Properties;
import com.jayway.restassured.response.Response;
import Org.Testing.TestResponseValidation.ResponseValidations;
import Org.Testing.TestSteps.HttpMethods;
import Org.Testing.TestUtilities.PropertiesFileLoad;

public class TC1
{
	public static void main(String[] args) throws IOException
	{

		Properties Pr=
		PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.prop
		erties");

		HttpMethods http=new HttpMethods(Pr);

		Response Res=http.GetRequest("QA_URI2");

		ResponseValidations ResObj=new ResponseValidations();
		ResObj.ResponseValidation(Res);
	}
}
```

We have called the PropFileLoad method to the object of file whose path we have sent to it as an argument. It returns the object of file and the same is stored in the object of Properties file.

```
Properties Pr=
PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.prop
erties");
```

Next to that, we have have passed the object of file to HttpMethods class through its constructor so that it should initialize the same in its global variable which can further be used by the methods of its class id required. Properties file is holding the URI as one of the value of key defined in file.

```
HttpMethods http=new HttpMethods(Pr);
```

Then we have called the GetRequest method of HttpMethods class and pass the Key of URI

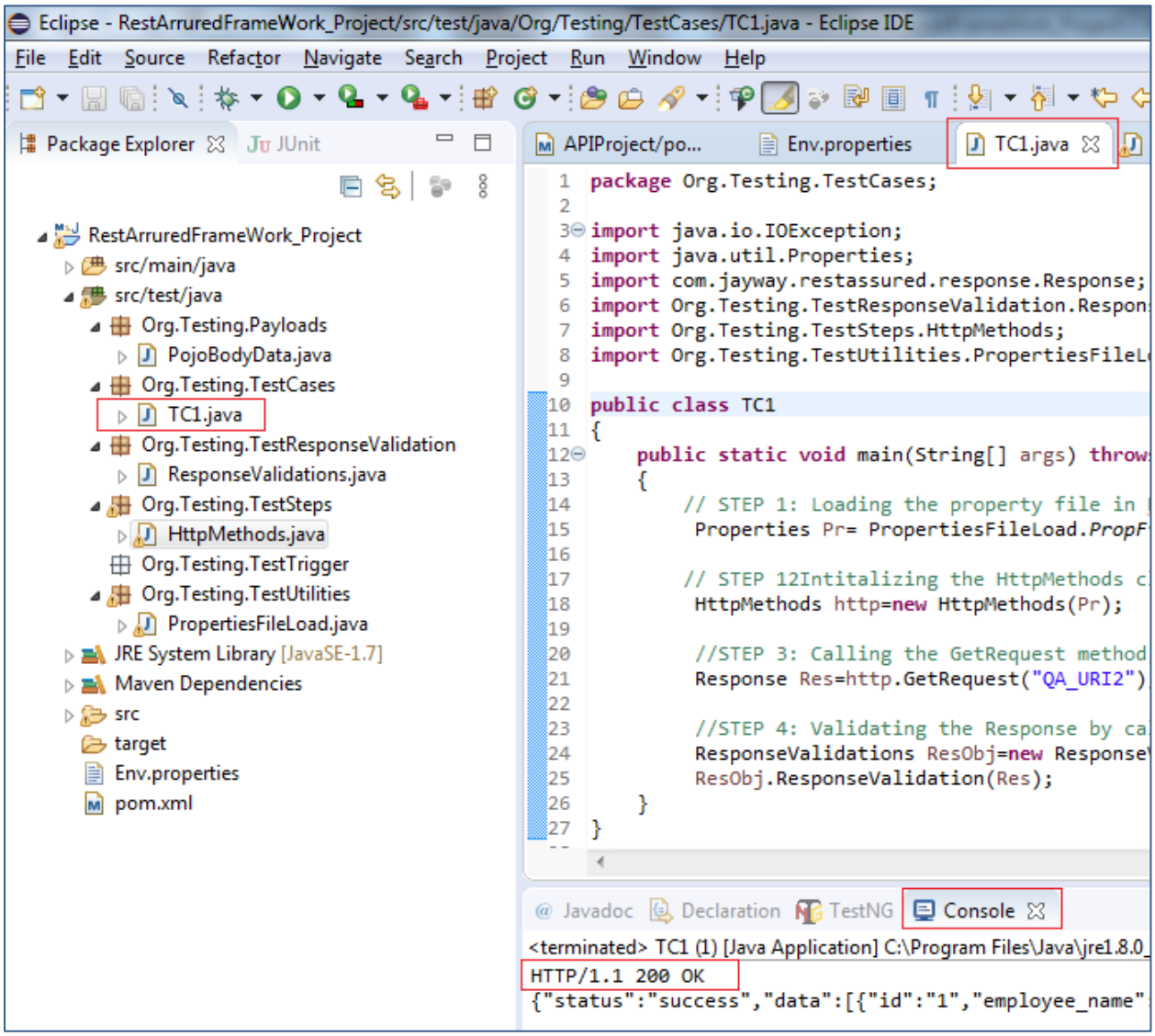
```
Response Res=http.GetRequest("QA_URI2");
```

This method will get the data from Corresponding URI value to this Key: `"QA_URI2"`
And return the entire response data which will be stored in the Res Object of Response class

Finally, this Res object holding the entire Response data is passed to `ResponseValidation` method for validation.

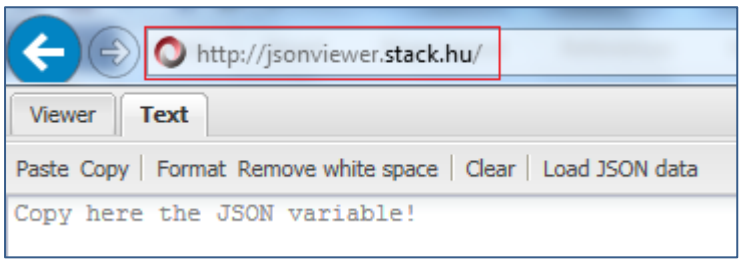
```
ResponseValidations ResObj=new ResponseValidations();
ResObj.ResponseValidation(Res);
```

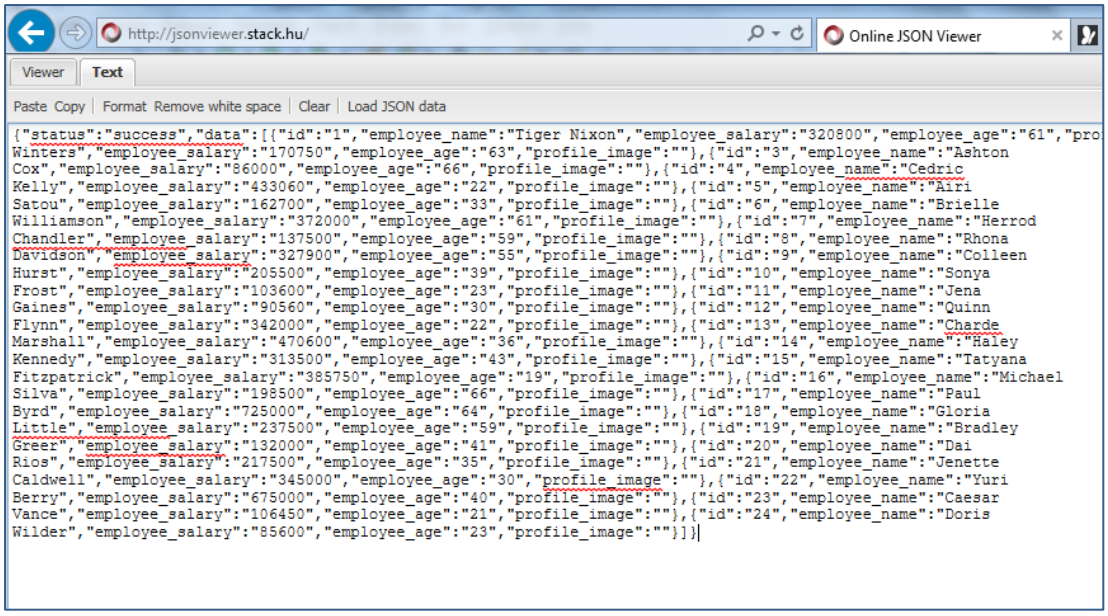
9. Let's execute the Test Case TC1 for the http get() request



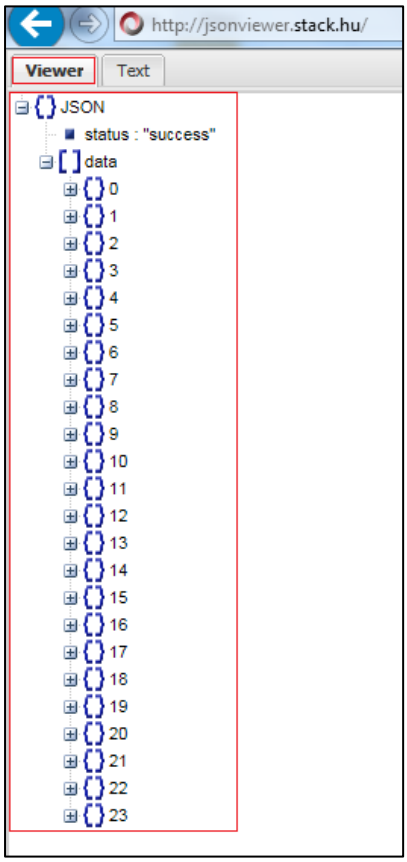
Note the status code in status line is 200 OK which means that our hit request is successfully executed. Apart from that the Json data is also displayed that was retrieved by the get request.

10. Copy the entire data from the console and paste in the online tool https://jsonviewer.stack.hu





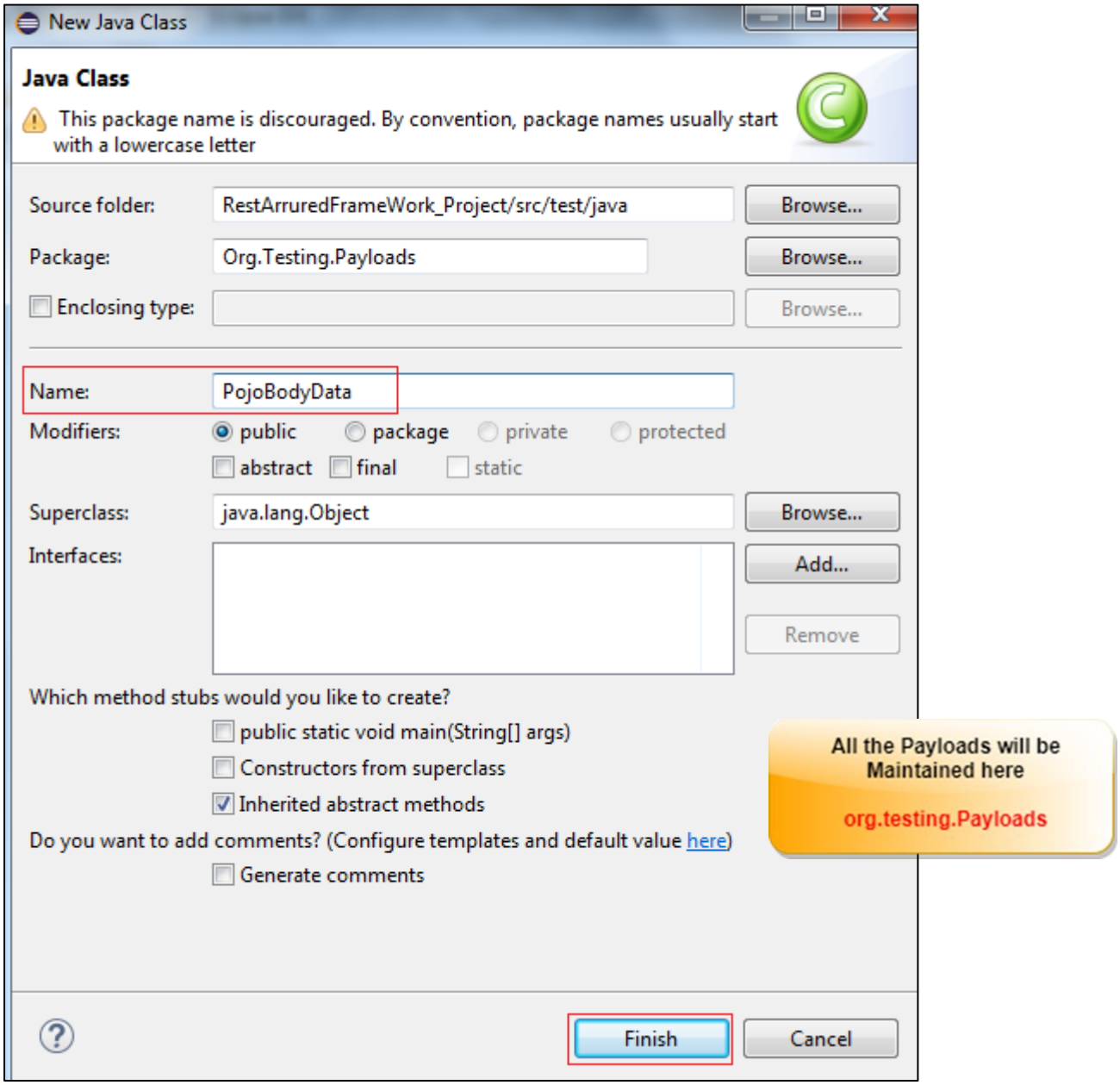
11. Click on viewer tab and here we go…



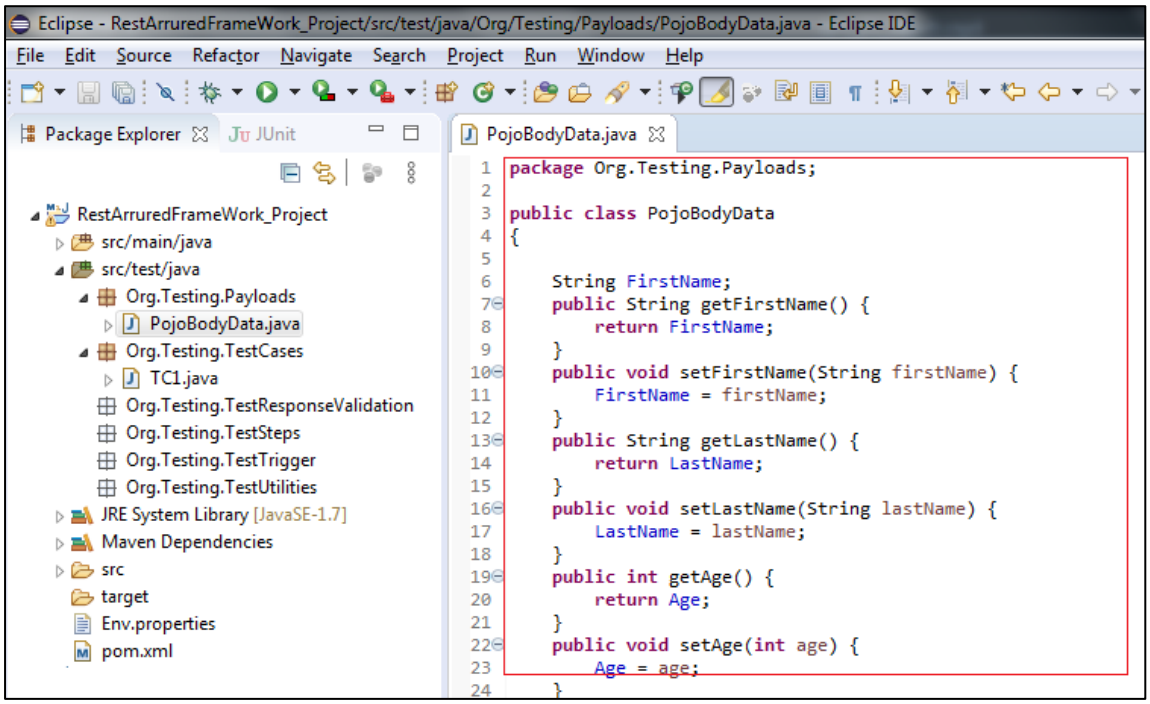
We have successfully automate the one TC1 for http get() request.

Creating Test Case TC2: Http post Request:

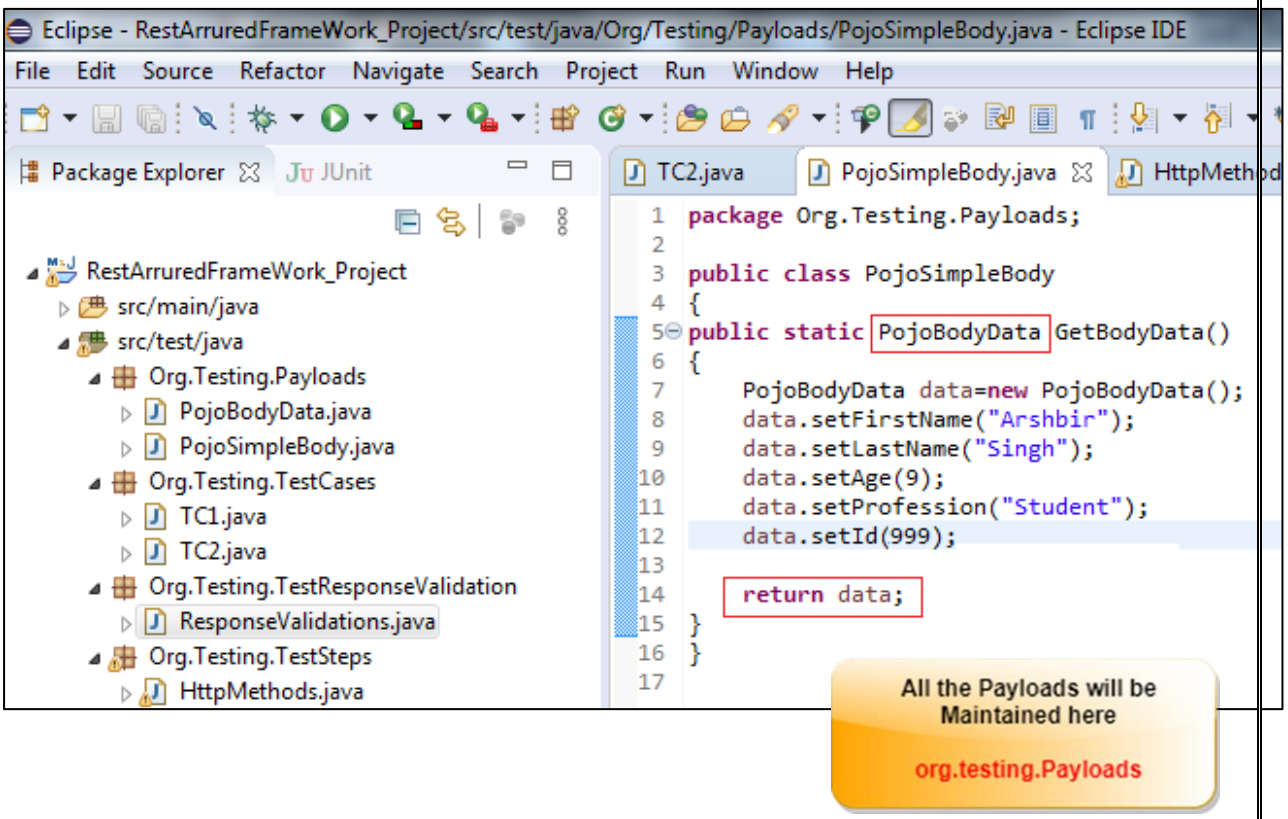
1. Create another class under Org.Testing.Payloads package say PojoBodyData.java in which we will create a body data using simple POJO technique. This data will be used latter on to hit the Post http request.



2. Declare few variables as shown below and generate getter and setter and save the class



3. Create another class PojoSimpleBody and create one method say GetBodyData() for creating the body data under the Org.Testing.Payloads package



Explanation:

```
package Org.Testing.Payloads;

public class PojoSimpleBody
{
public static PojoBodyData GetBodyData()
{
	PojoBodyData data=new PojoBodyData();

	data.setFirstName("Arshbir");
	data.setLastName("Singh");
	data.setAge(9);
	data.setProfession("Student");
	data.setId(999);

	return data;
}
	}
```

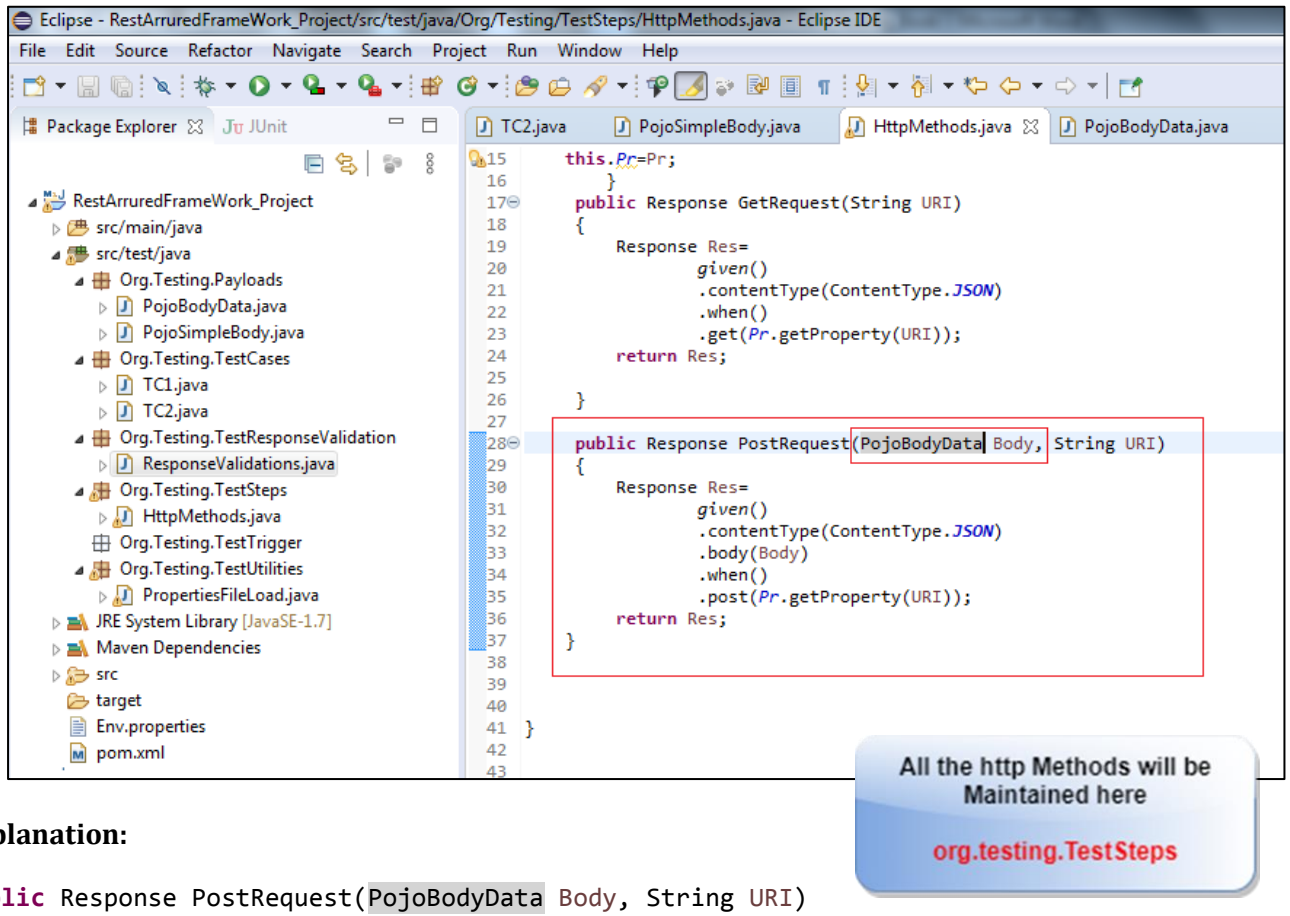
We have already created a few variables and their corresponding getter and setter in class `PojoBodyData` in step 2 for the final creation of the simple json body data. Now, in this class we have created one method `GetBodyData` whose return type `PojoBodyData` and it sets the values to the variables those were defined in step 2.

```
	data.setFirstName("Arshbir");
	data.setLastName("Singh");
	data.setAge(9);
	data.setProfession("Student");
	data.setId(999);
```

Finally, it returns the object of `PojoBodyData` which contain all these data values to the calling function in test case TC2

```
	return data;
```

4. Create Http Method say PostRequest accepting two string arguments one for body data and one for URI under Org.Testing.TestSteps package



Explanation:

```
public Response PostRequest(PojoBodyData Body, String URI)
	{
		Response Res=
				given()
				.contentType(ContentType.JSON)
				.body(Body)
				.when()
				.post(Pr.getProperty(URI));
		return Res;
	}
```

This method `PostRequest(PojoBodyData Body, String URI)` is called by TC2 for posting the data at given URI.

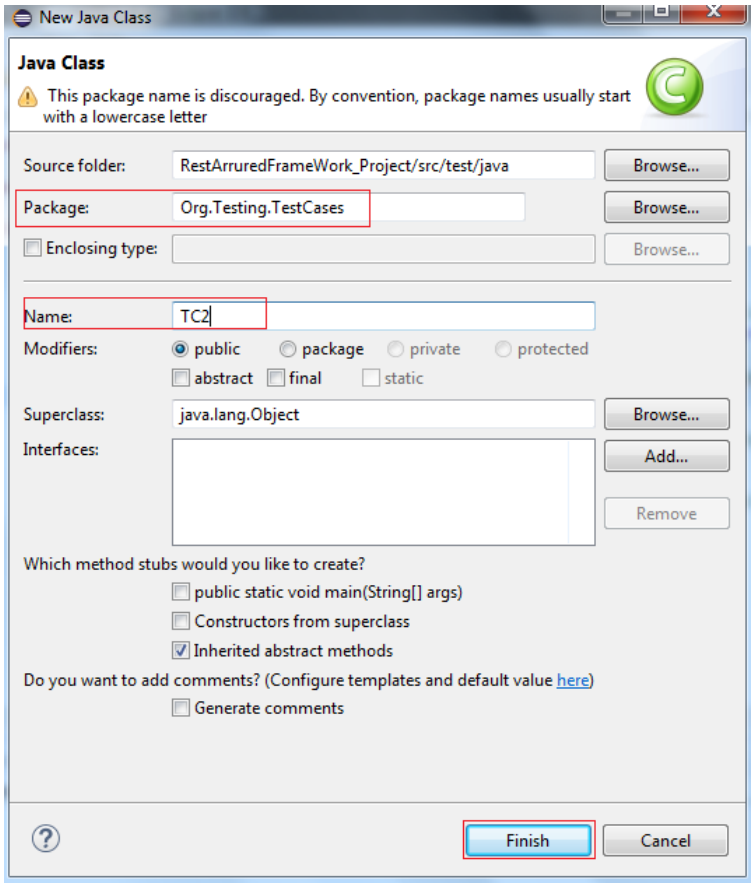
Value of key- URI is fetched from the Env.properties file by the statement

```
Pr.getProperty(URI)
```

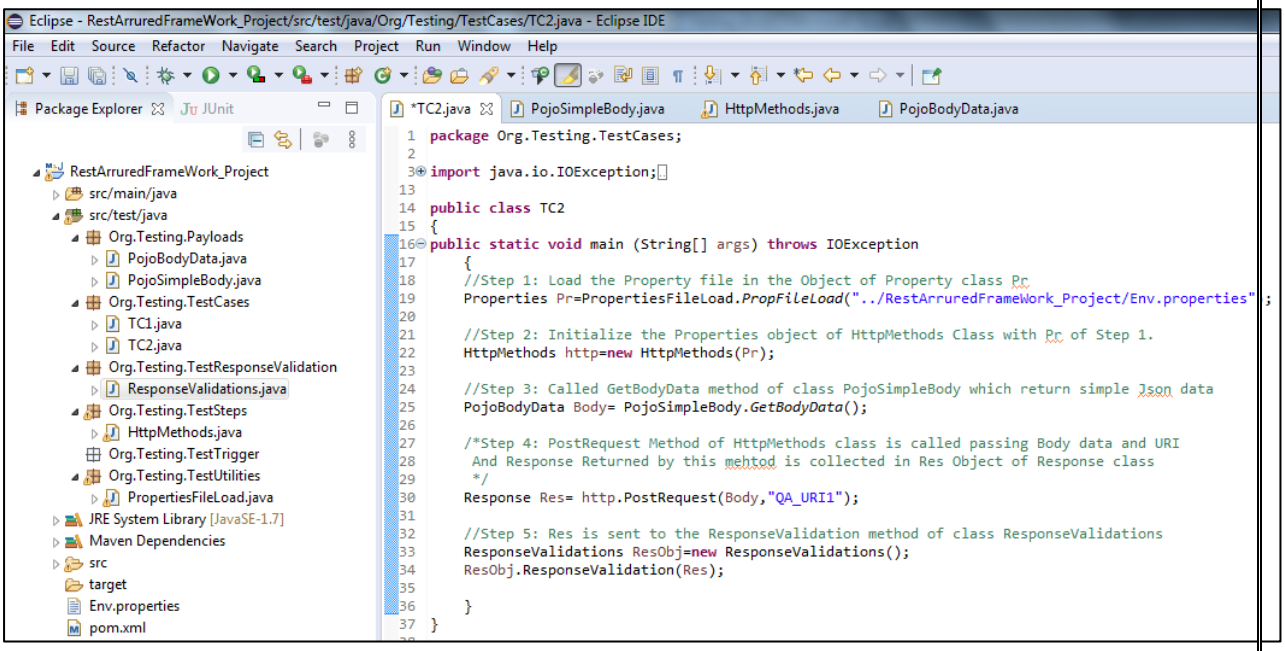
Finall, the Response of the post http request is returned to the calling statement

```
return Res;
```

5. Create class TC2 under the Org.Testing.TestCases package for http post request



6. Write the following code in the TC2 which is explained further:



Explanation:

```
public class TC2
{
public static void main (String[] args) throws IOException
	{

	Properties
Pr=PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
	HttpMethods http=new HttpMethods(Pr);
	PojoBodyData Body= PojoSimpleBody.GetBodyData();
	Response Res= http.PostRequest(Body,"QA_URI1");
	ResponseValidations ResObj=new ResponseValidations();
	ResObj.ResponseValidation(Res);
	}
}
```

With the starting statement, we have loaded the Property file in the Object Pr of Property class

```
Properties
Pr=PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
```

Next to it the Pr is passed to the HttpMethods class through its constructor

```
	HttpMethods http=new HttpMethods(Pr);
```

Further to it, we have called method GetBodyData method of class PojoSimpleBody which return simple Json data

```
	PojoBodyData Body= PojoSimpleBody.GetBodyData();
```

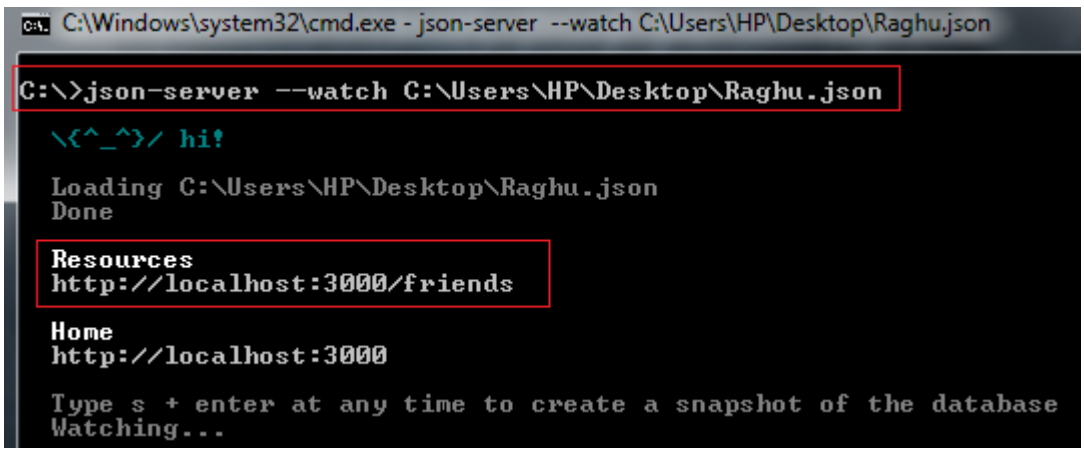
PostRequest Method of HttpMethods class is called passing Body data and URI and Response Returned by this mehtod is collected in Res Object of Response class

```
	Response Res= http.PostRequest(Body,"QA_URI1");
```

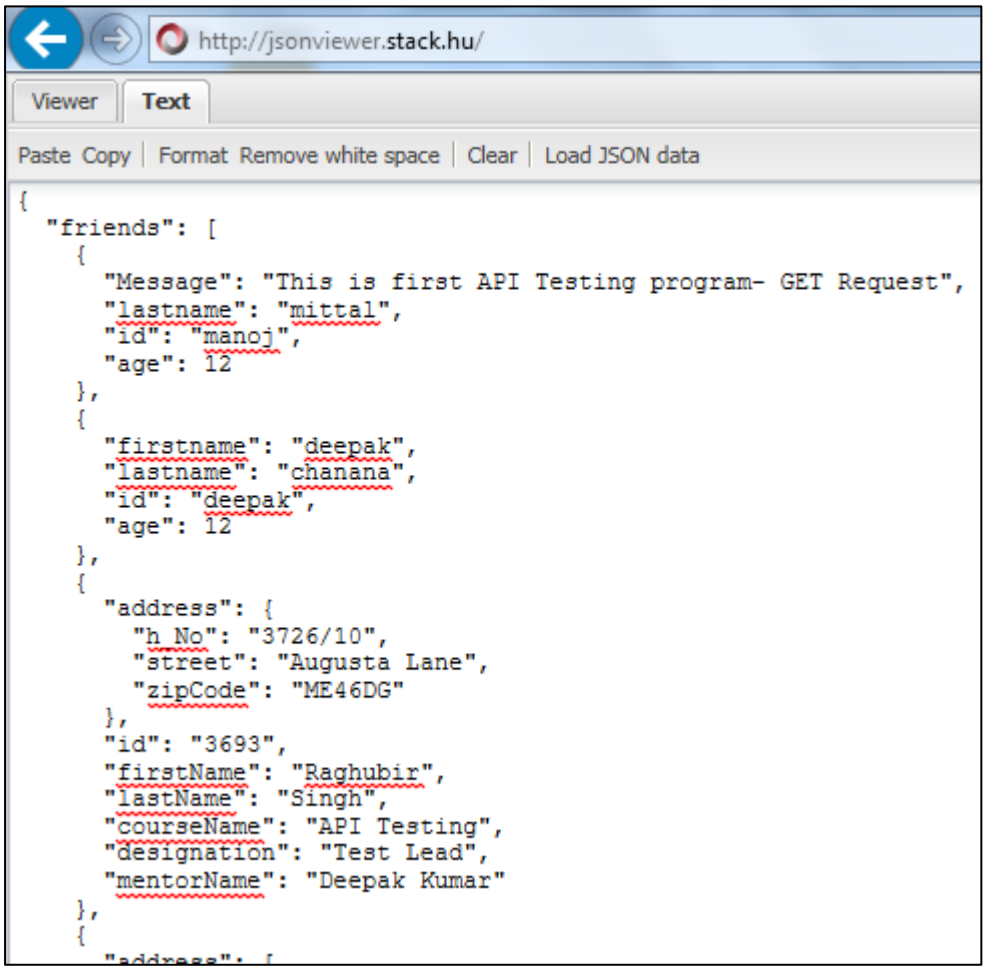
Finally, Object Res is sent to the ResponseValidation method of class ResponseValidations

```
	ResponseValidations ResObj=new ResponseValidations();
	ResObj.ResponseValidation(Res);
```

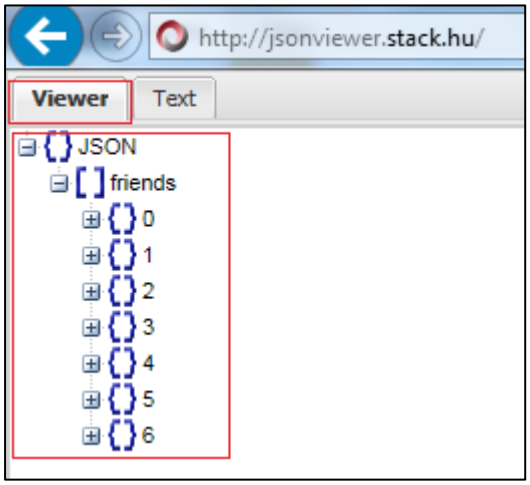
7. Let's start the Json server with respective target Json file



8. Check the details of Json file at URI : http://localhost:3000/friends before hitting the post request. Open the file in Notepad and copy the entire data and paste the data in the Text tab at the online tool http://jsonviewer.stack.hu/

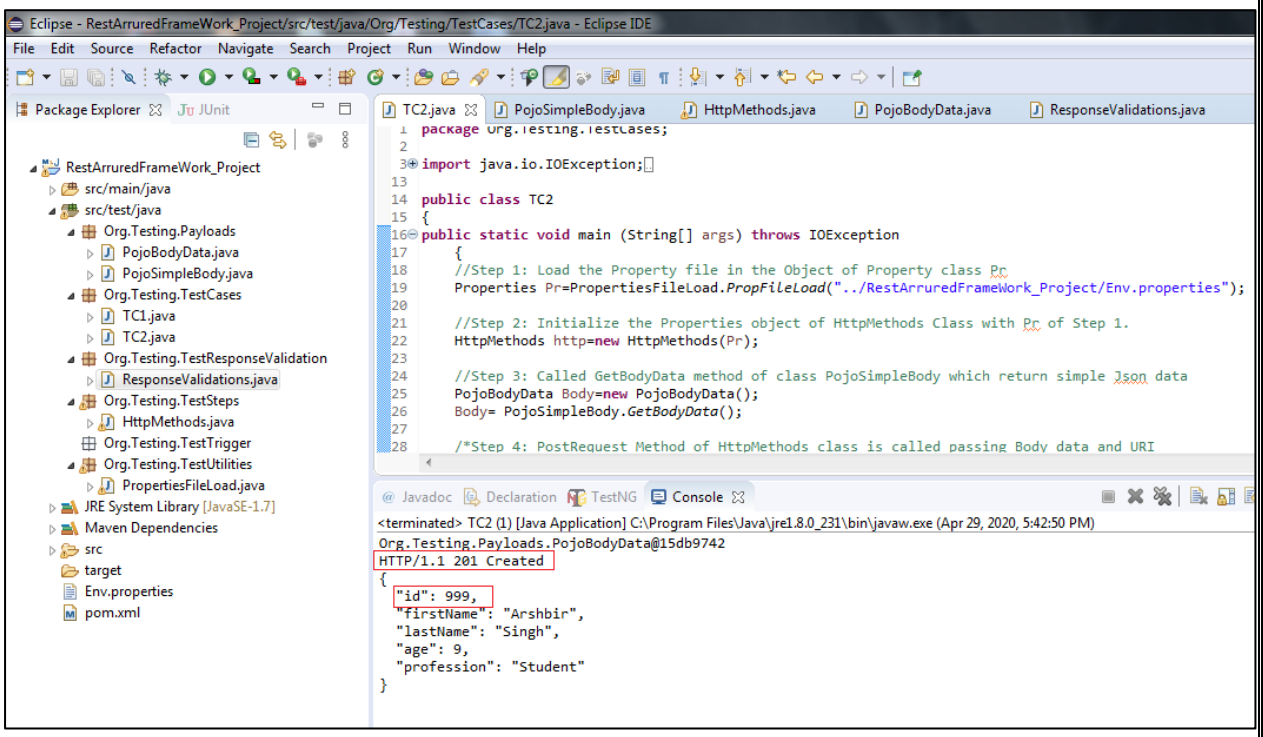


9. Click on the Viewer tab and view the Json structure

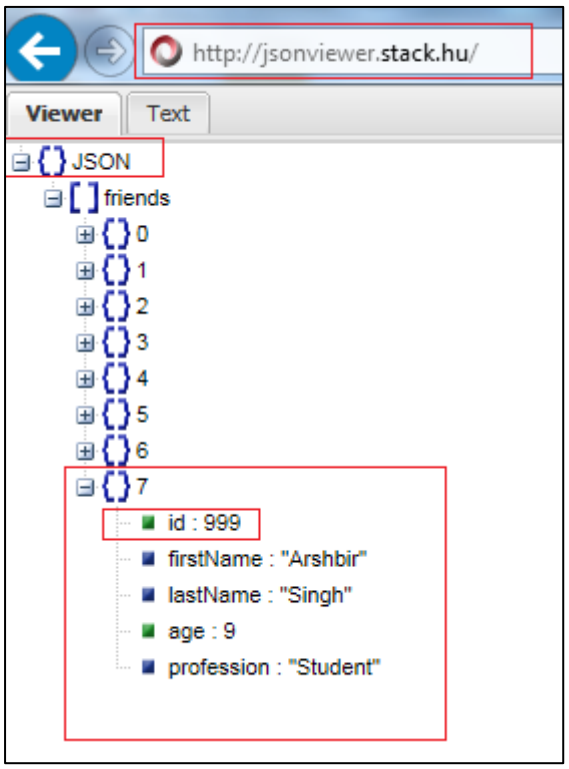


It is observed that under the Json object, there is a 'friends' Array which have 7 objects (0 to 6)

10. Execute the test case TC2:



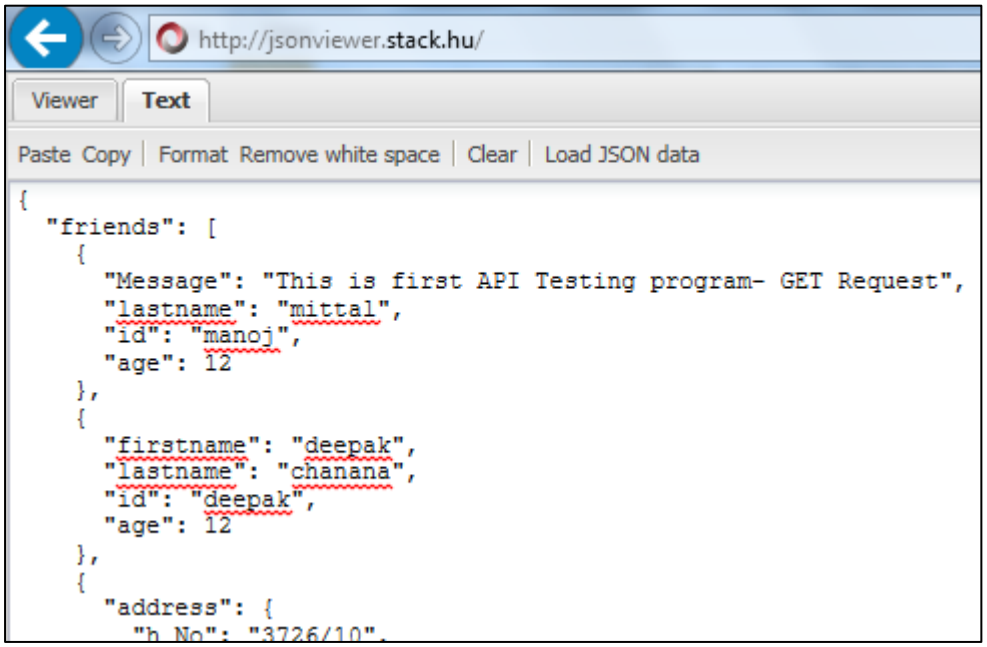
11. Check the details of Json file at URI : http://localhost:3000/friends After hitting the post request. Open the file in Notepad and copy the entire data and paste the data in the Text tab at the online tool http://jsonviewer.stack.hu/ and then click on Viewer tab.



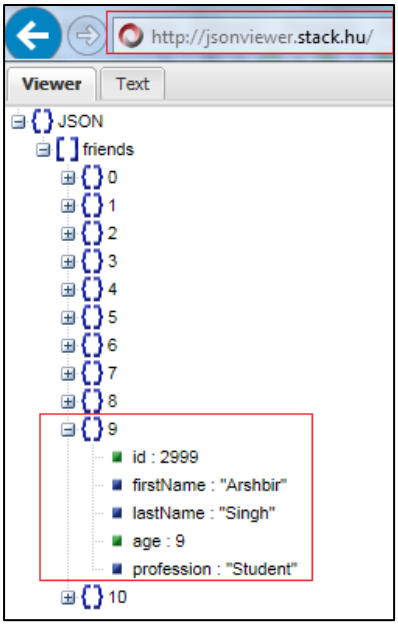
So far so good, we have successfully automate the one TC1 for http post() request.

Creating Test Case TC3: Http delete() Request

Check the details of Json file at URI: http://localhost:3000/friends before hitting the delete request. Open the file in Notepad and copy the entire data and paste the data in the Text tab at the online tool http://jsonviewer.stack.hu/

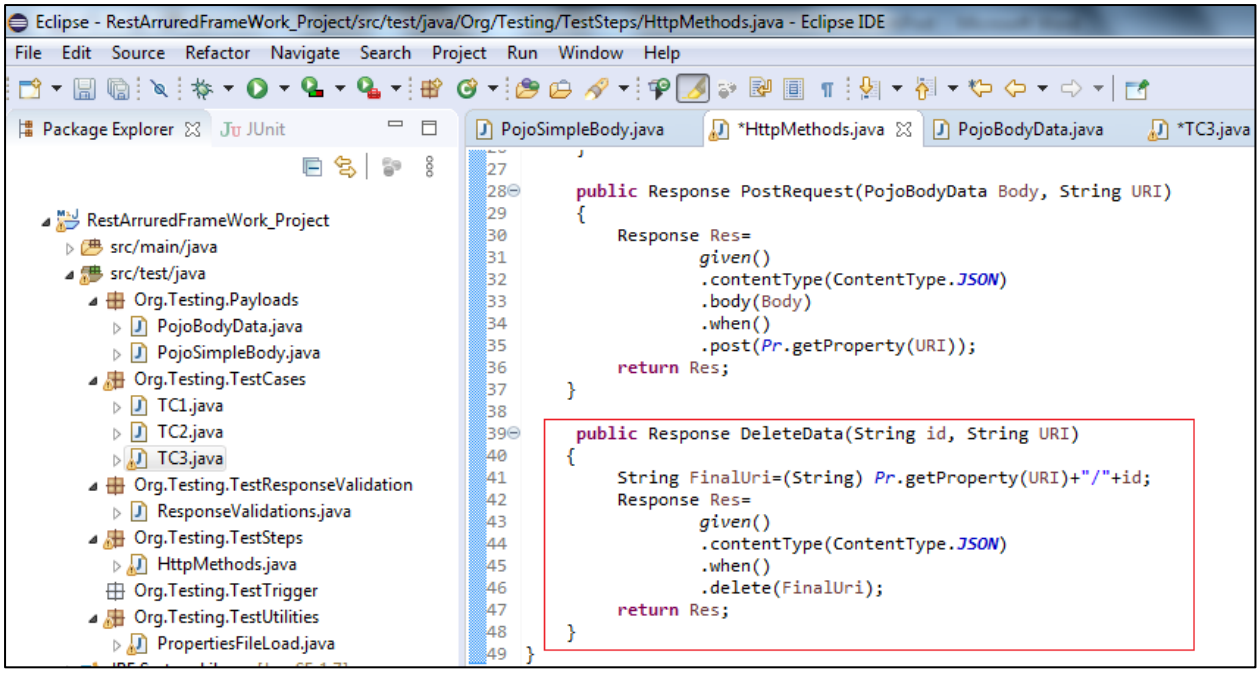


1. Click on the Viewer tab and view the Json structure

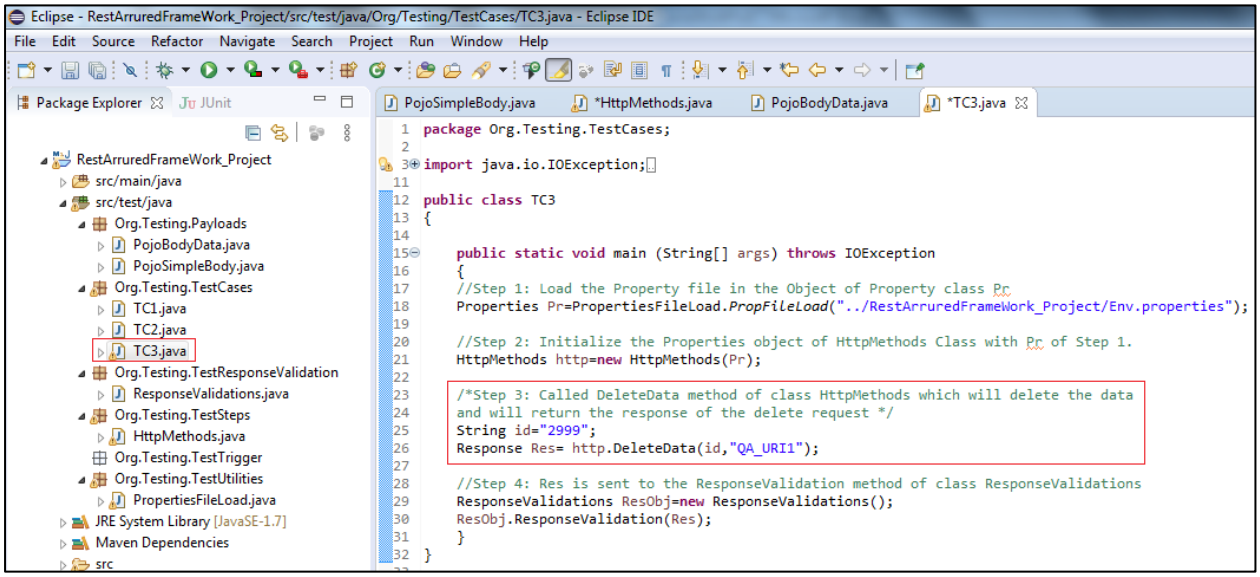


It is observed that under the Json object, there is a 'friends' Arrays which have 11 objects 0 to 10

2. Create the Delete method say DeleteData() which will accept two arguments of a type String – one for id and one for URI



3. Create one class TC3 under the package Org.Testing.TestCases and write the following code



Explanation:

```
public class TC3
{

    public static void main (String[] args) throws IOException
    {

    Properties Pr=PropertiesFileLoad
    .PropFileLoad("../RestArruredFrameWork_Project/Env.properties");

    HttpMethods http=new HttpMethods(Pr);

    String id="2999";
    Response Res= http.DeleteData(id,"QA_URI1");

    ResponseValidations ResObj=new ResponseValidations();
    ResObj.ResponseValidation(Res);
    }
}
```

First we have loaded the property file as in the file we have maintained all the URI's in the form of key:value pairs. The object of properties file returned by `PropFileLoad()` method under the Org.Testing.Utilities package.

```
Properties Pr=PropertiesFileLoad
.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
```

After that, we have initialized the property object inside the HttpMehtods class with the one which was returned from the above step through the constructor of the class HttpMethods.

```
HttpMethods http=new HttpMethods(Pr);
```

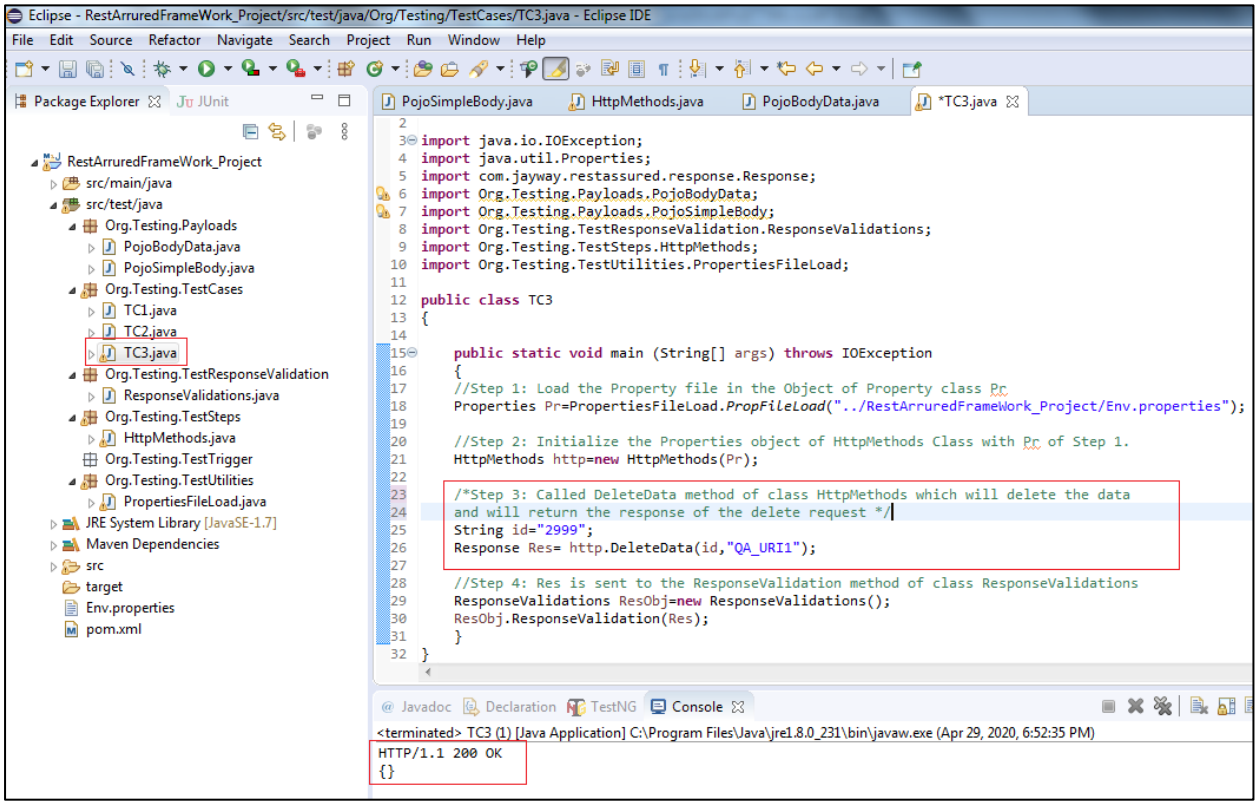
Then we call the method DeleteData() by passing string arguments first one the id and second URI Key. This method returns the Response of the delete() http request which is further assigned to the object Res of Response class

```
Response Res= http.DeleteData(id,"QA_URI1");
```

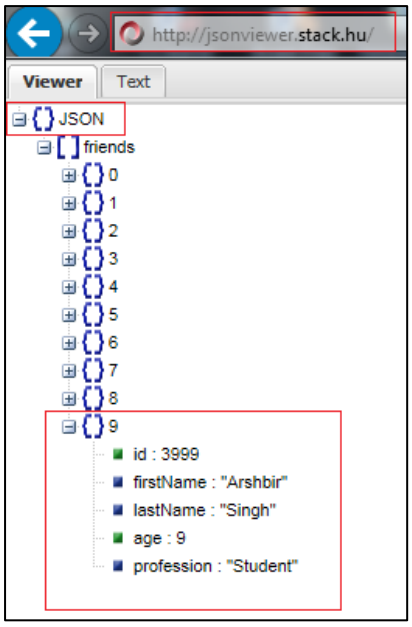
And finally, we performed the Response validation as usual by passing the Res object of Response to the method ResponseValidation() under package Org.Testing.ResponseValidations package

```
ResponseValidations ResObj=new ResponseValidations();
ResObj.ResponseValidation(Res);
```

4. Execute the test case TC3:

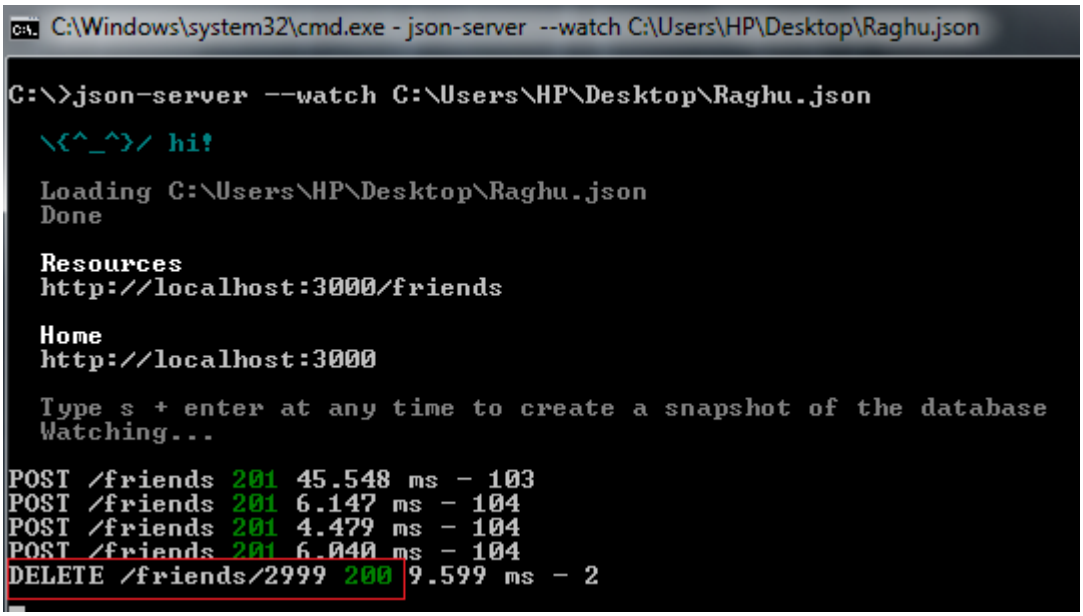


5. Let's check the details of Json file at URI: http://localhost:3000/friends after hitting the delete request. Open the file in Notepad and copy the entire data and paste the data in the Text tab at the online tool http://jsonviewer.stack.hu/ and view in the Viewer tab



Finally, the object with id=2999 was deleted from the target URI :

It is verified from the command prompt also:

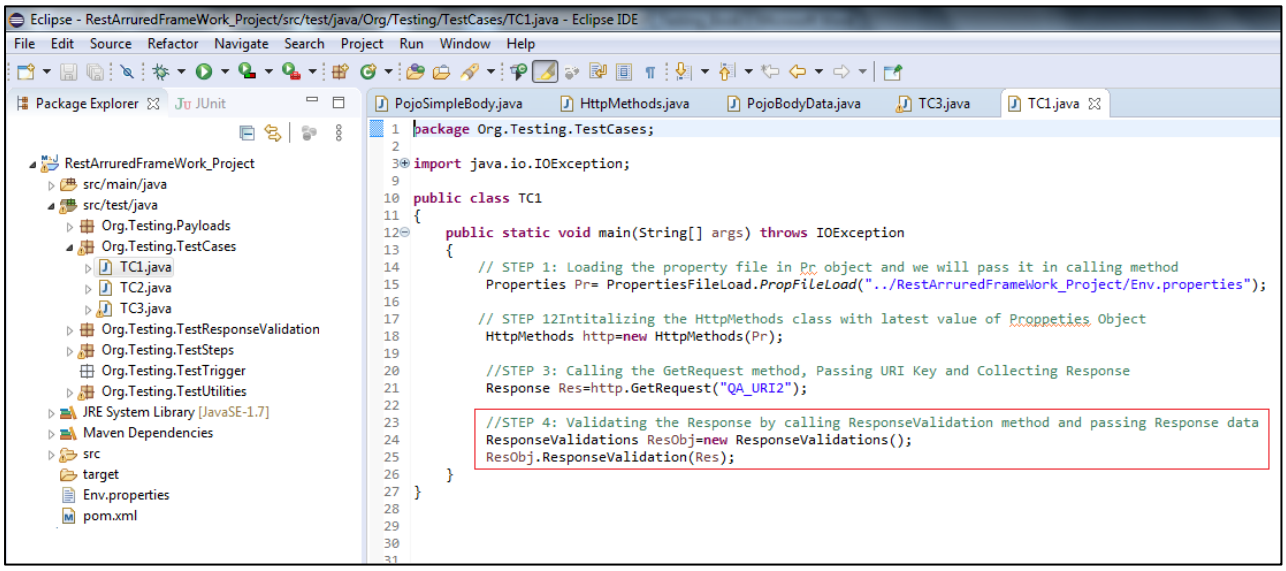


Part- II: Response Validations, Json Response Parsing and API Chaining

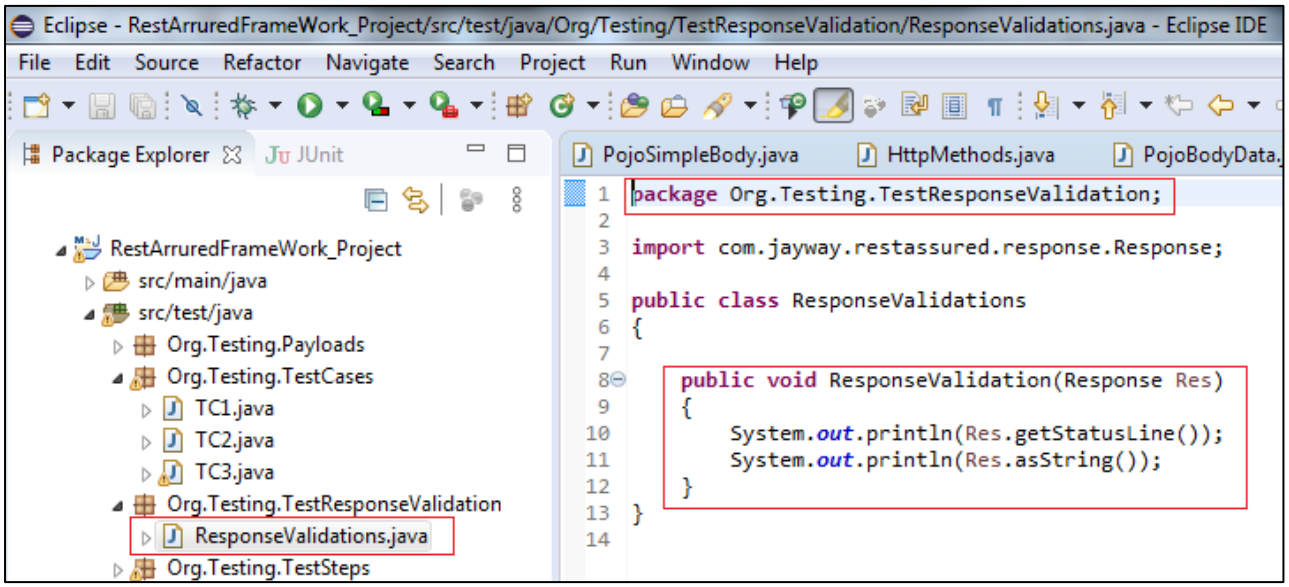
So, far we have created 3 test cases viz: TC1, TC2 and TC3 for the http get(), post() and delete() requests respectively.

We have also created methods for the Response Validation but yet so far we are just printing the Response results on console. To validate the response we need to do some comparisons and these are accomplished using "***Assertions***".

Look at the code for TC1, we have called a method ResponseValidation() and pass Res object of Response class to this method as an argument.

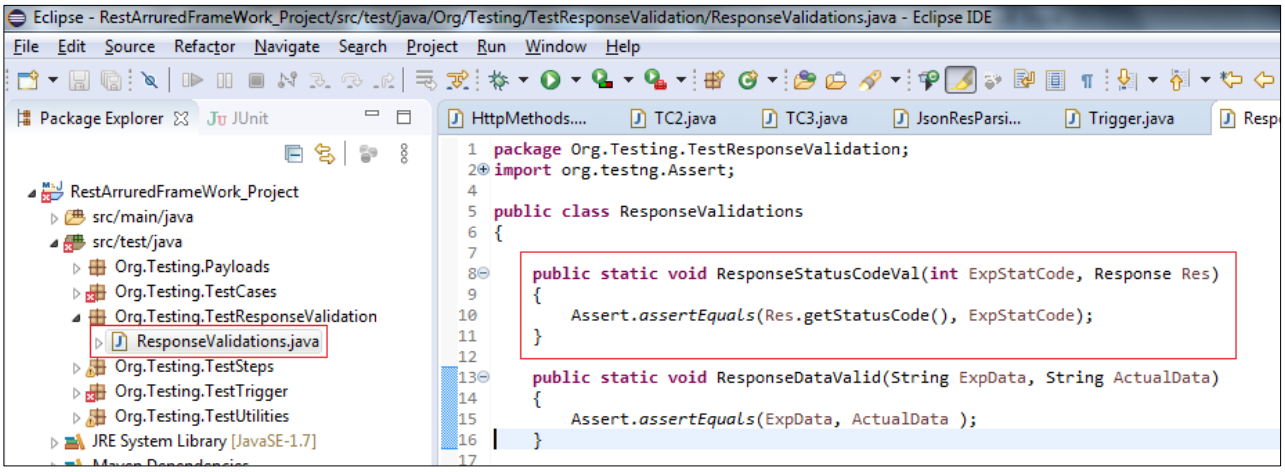


At the moment we are just passing the object of Response to the ResponseValidations section of our framework.



In order to validate response status code/status line we have to pass expected status code/status line (which is already holds up by the Res object of Response for recent request hit) and actual status code/status line where assertions will compare and return the result to the calling method

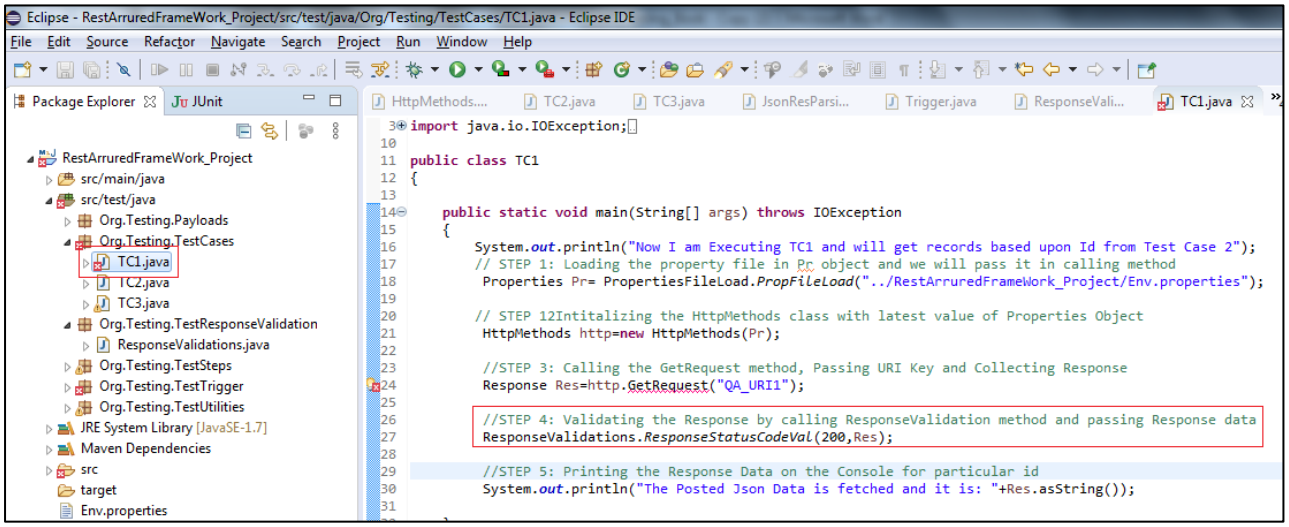
Let's write one assertion in ResponseValidations class under method ResponseStatusCodeVal().



Now, this method is rewritten and has two arguments on of type int which will be holding the expected status code and other of type Response which will hold the object of Response from which the actual status code will be fetched and further it will be compared by the assertion. In the method definition Assertion is applied on response status code

```
public static void ResponseStatusCodeVal(int ExpStatCode, Response Res)
	{
		Assert.assertEquals(Res.getStatusCode(), ExpStatCode);
	}
```

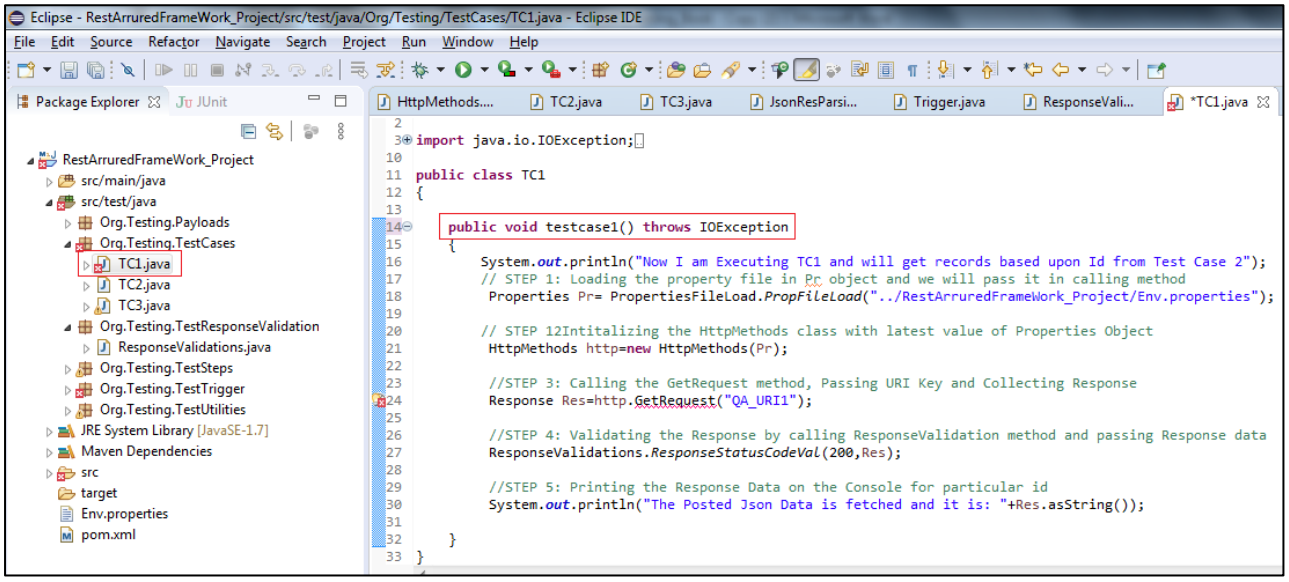
Similarly, the changes must be made in the method ResponseStatusCodeVal() in the TC1 where it is declared. We have to update it so that it should pass actual status code value and object of Response class as arguments:




```
ResponseValidations.ResponseStatusCodeVal(200,Res);
```

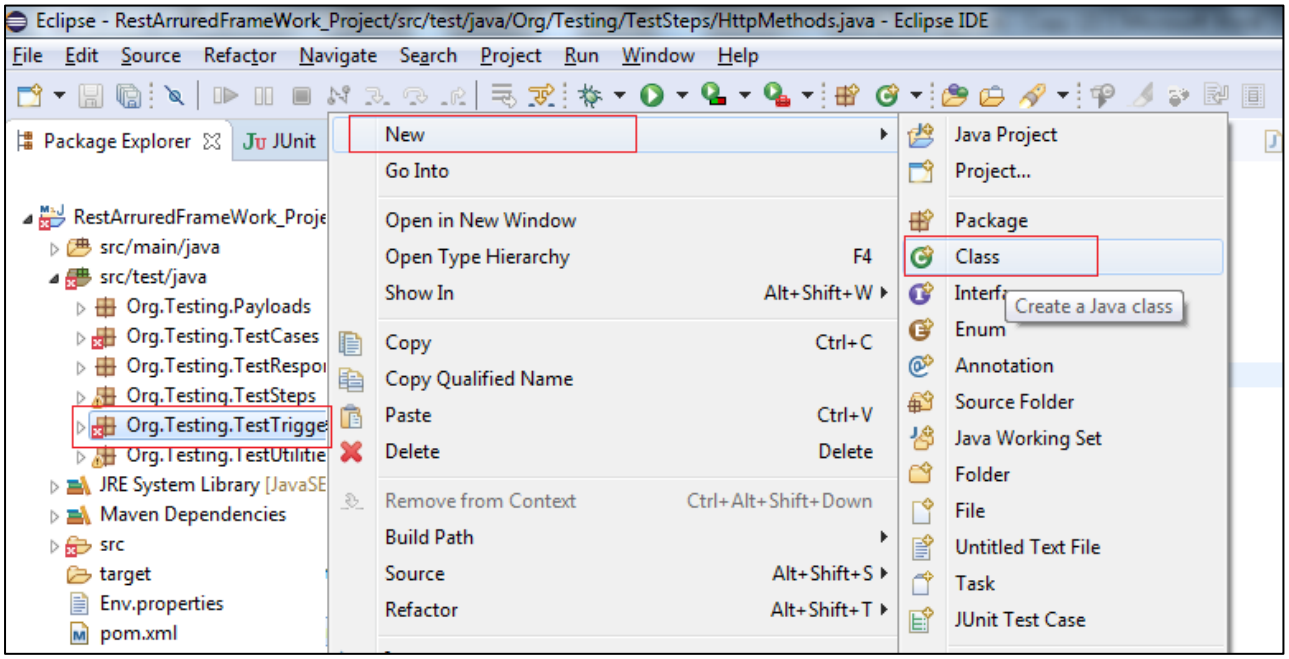
Method ResponseStatusCodeVal() in TC1 is now passing 200 as integer which is the expected value of the get() hit request status code and an object of Response class which in turn is encapsulating the actual status code of the get() request after hit.

Let's also remove the main method from TC1, such as TC1 could be called from anywhere whole as method:

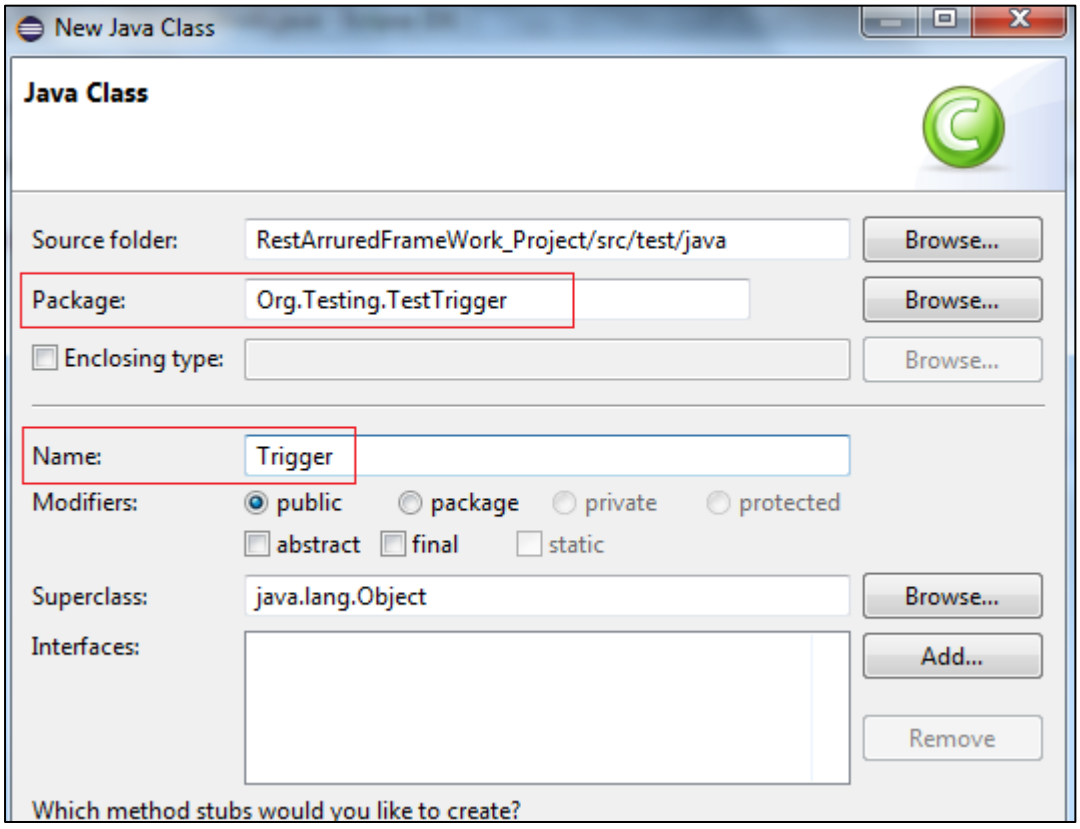


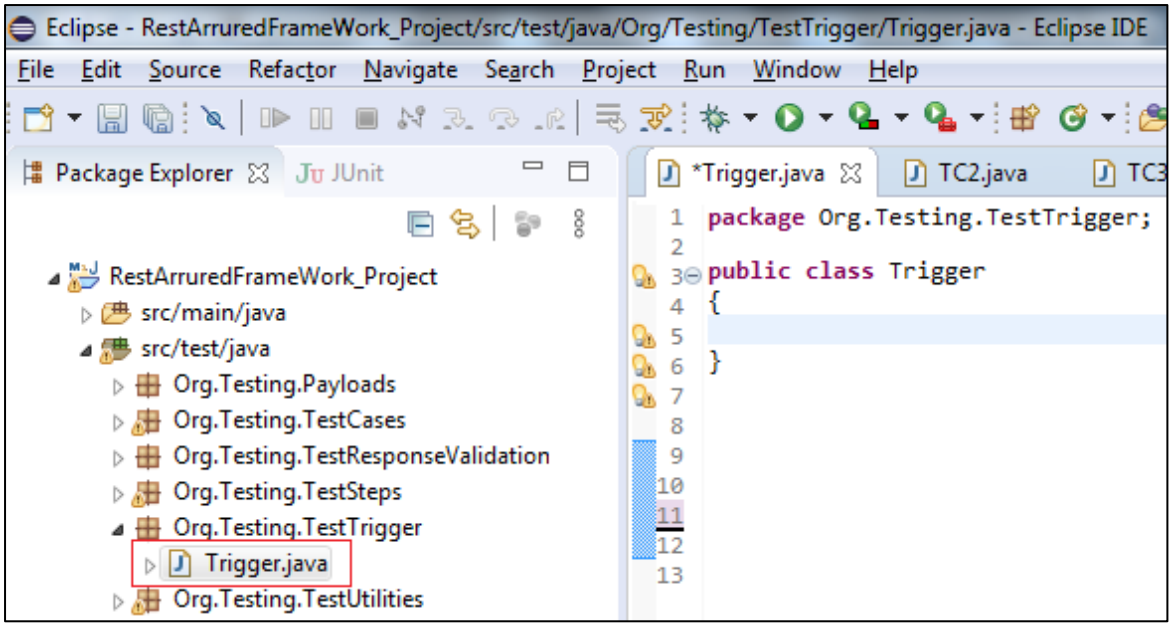
Creating the Trigger class: We need to automate our framework so that things happen automatically one after the other in the predefined sequence. Trigger class is the one where we will define our sequence of automated test cases. Let's create the Trigger class under package Org.Testing.Trigger

1. Right click on package Org.Testing.Trigger and select New> Class



2. Enter the name of class as Trigger and click on Finish Button

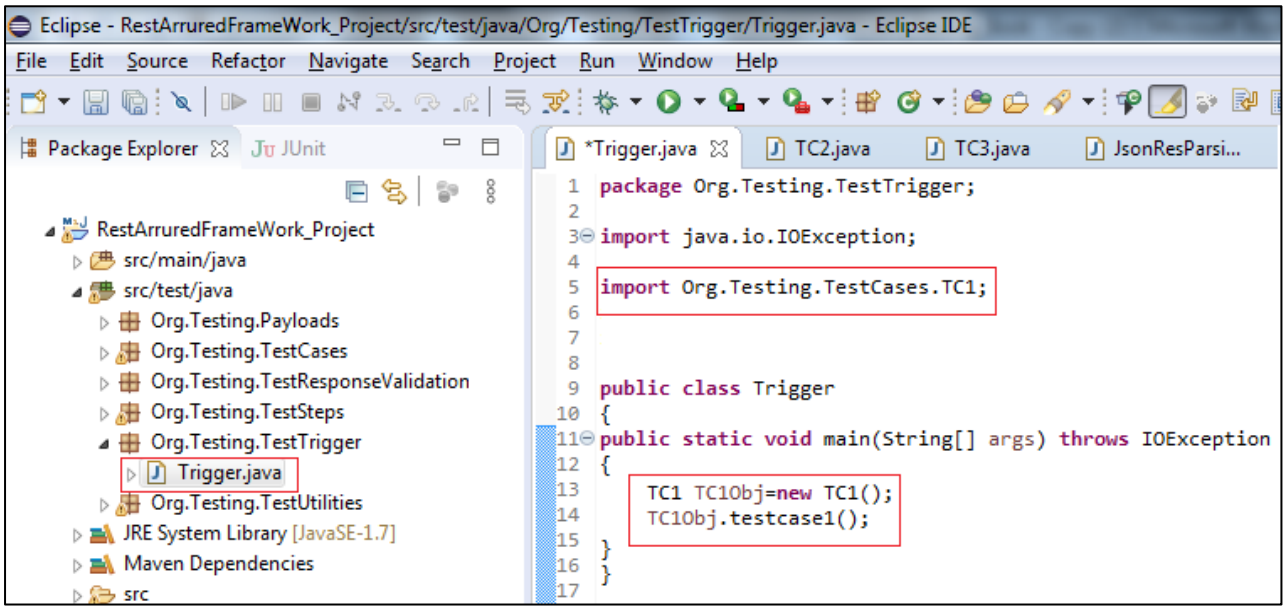




3. Start creating Trigger class with main method

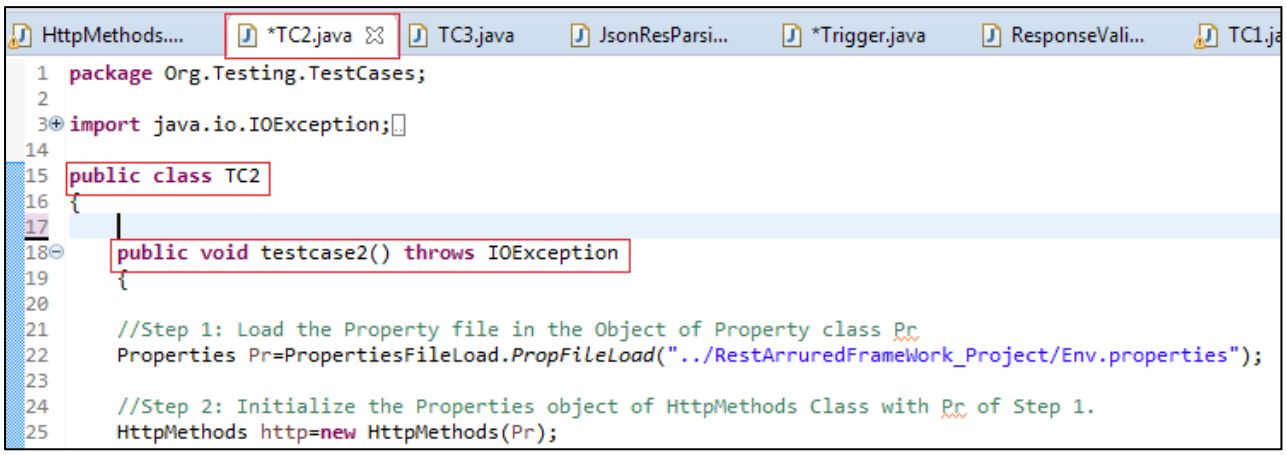
```
public class Trigger
{
public static void main(String[] args)
{

}
}
```

4. Call The TC1 in the Trigger class

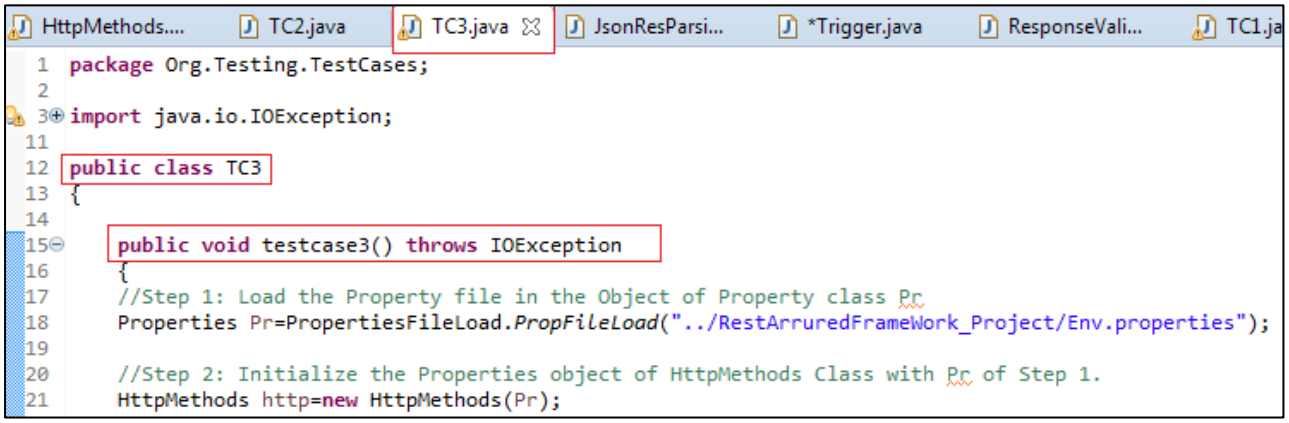


Similarly, we will be changing our other test cases TC2 post() request and TC3 delete() request into methods by removing their respective main methods without disturbing rest of the code:

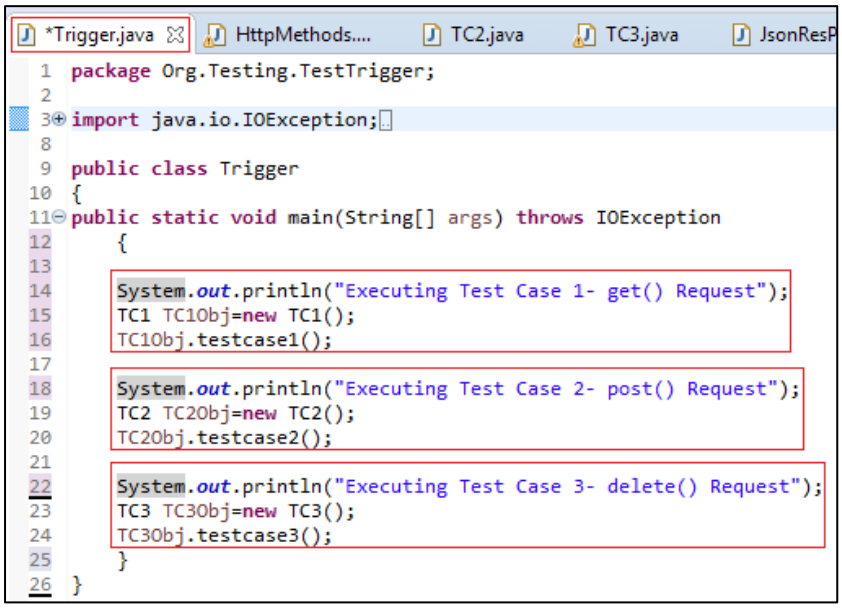
TC2:



TC3:



Now the starting point of our framework is the main method under Trigger class and we will be able to call TC1, TC2 and TC3 in Trigger class and in any sequence like this:



API Chaining

API Chaining is parsing value of key or keys from the Response of one https request (Test Cases in terms of Framework) and use it in other http requests/Test Cases.

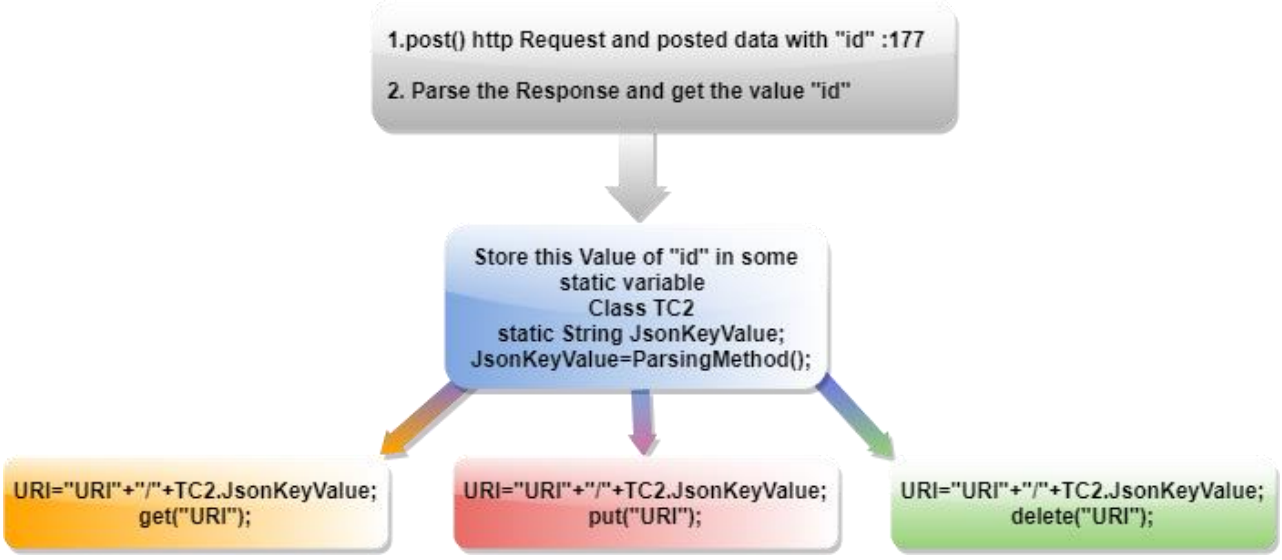
Let's suppose we post some data with the post() request and we fetch its "id" say 177 from the response data and other test case(s) such as get(), put() and delete() uses this value 177 to get, update and delete the posted data by the post() request.

For an instance if URI is *http://localhost:3000/friends* where the data is posted by the post request with ***"id":177*** then the get request URI will look something like this:

get("*http://localhost:3000/friends**/177**"*);
put("*http://localhost:3000/friends**/177**"*);
delete("*http://localhost:3000/friends**/177**"*);

API chaining is achieved by storing the parse "Id":177 in some static variable so that it could be used with the help of its class name.

API Chaining



Chapter -8 Diagram 1

Now we are well versed with the concepts of Response Validations, Json Response Parsing and API Chaining and we are all set to move to the Part-III of the framework design which is focused primarily on achieving some goal out the framework execution.

Part- III: Defining and Accomplishing Task through Framework

We will first see the pictorial representation of the framework design. As explained earlier also there is not hard and fast rule or standardized design of the framework and this varies from project to project and company to company. We will be just practicing here at our own framework design where the requirement is to execute four http request one after the other and the code for these requests will be specifically written in different Test Cases under our package org.testing.TestCases. We will also practice Response validation and API chaining concepts.

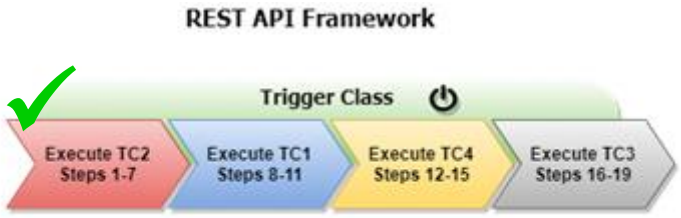
We will maintain execution sequence of our Test Case(s) in the Trigger Class which will be defined under org.testing.Trigger package.

Let's startup with defining task that we expect to accomplish out of our framework:

What is the projects requirement?

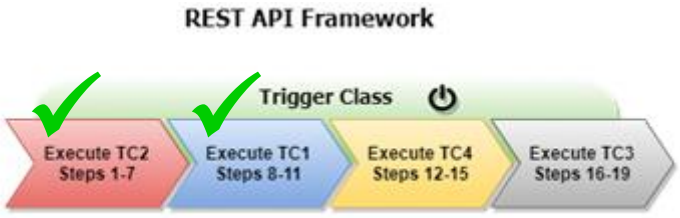
A. Expectations from the post() request:
   1. To hit the post() request
   2. To validate the Response Status Code and Data
   3. To Parse the Response data and fetch value of "id" key
   4. To store that value of "id" key in static variable so that it can be accessed by using its class name by any of the test case in this project.

In our case post request is handled by TC2 so, we will be executing TC2 first under the trigger class.



B. Expectations from the get() request:
   1. To the hit the get() request for fetching that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.
   2. To display the JSON data fetched by get() request for this particular "id" on console
   3. To validate the Response Status Code and Data for this get() request

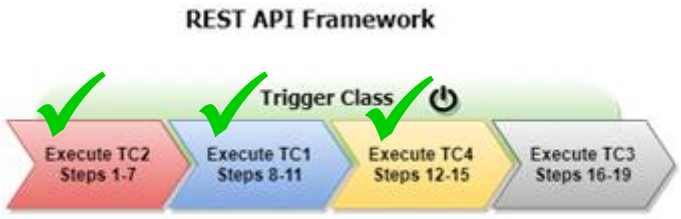
In our case get request is handled by TC1 so, we will be executing TC1 second under the trigger class.



C. Third expectation is from put() request:
   1. To hit the put() request for updating that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" in that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.

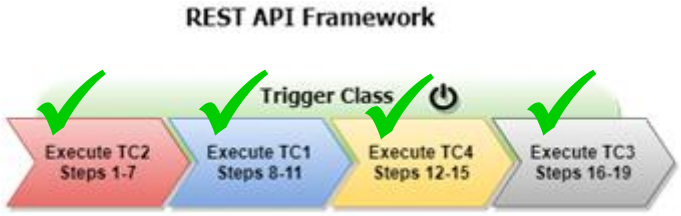
2. To update the JSON data say firstName
3. To validate the updated JSON data by parsing the new updated values of key(s) from Response Data
4. To validate the Response Status Code

In our case put() request is being handled by TC4 so, we will be executing TC4 at third number under the trigger class. *We will design the test case TC4 for put() request latter on as we are just planning our framework task.*

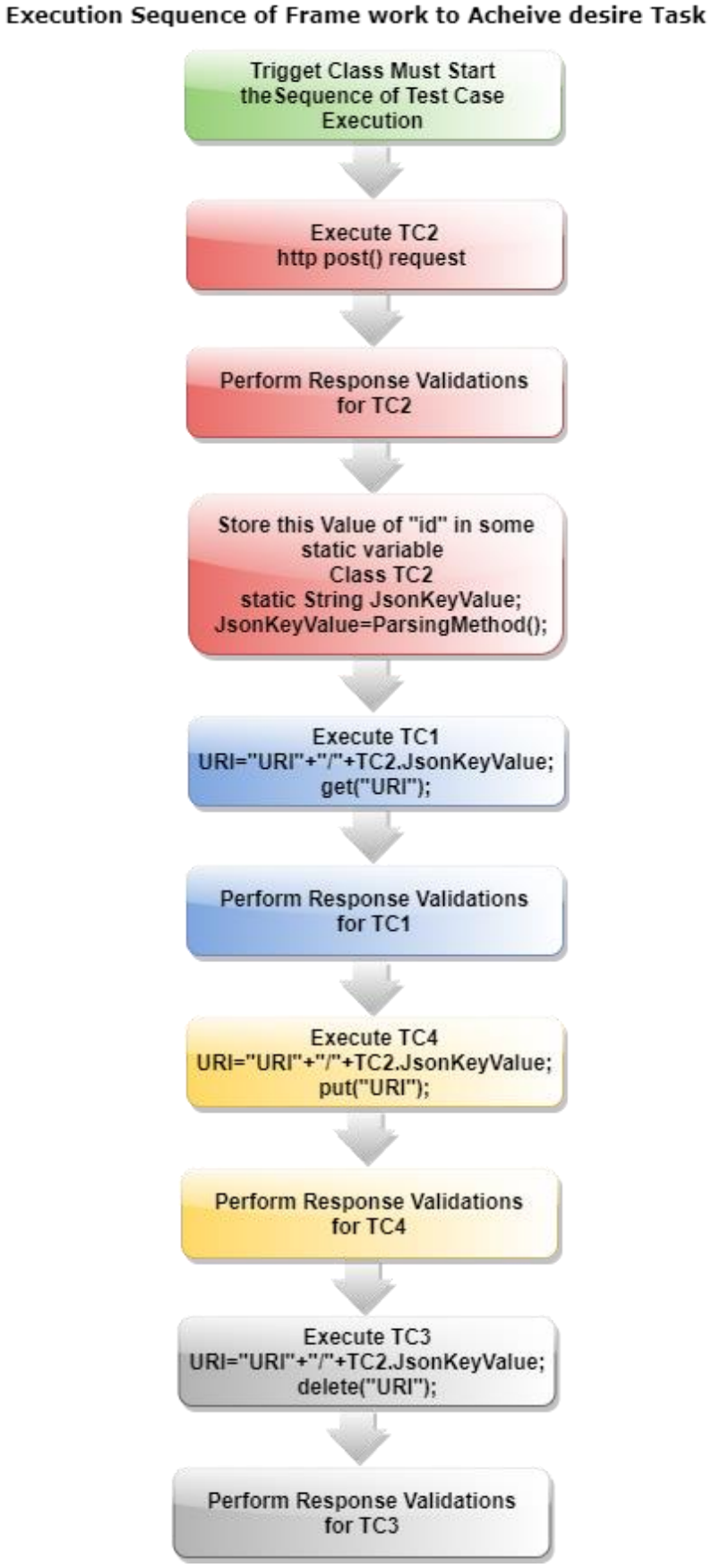


D. Final expectation is from delete() request:
1. To hit the delete() request for deleting that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" in that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.
2. To delete the JSON data
3. To validate the Response Status Code

In our case delete() request is being handled by TC3 so, we will be executing TC3 at the last under the trigger class.



Moving further, the flow chart of our task in hand will look something like this:

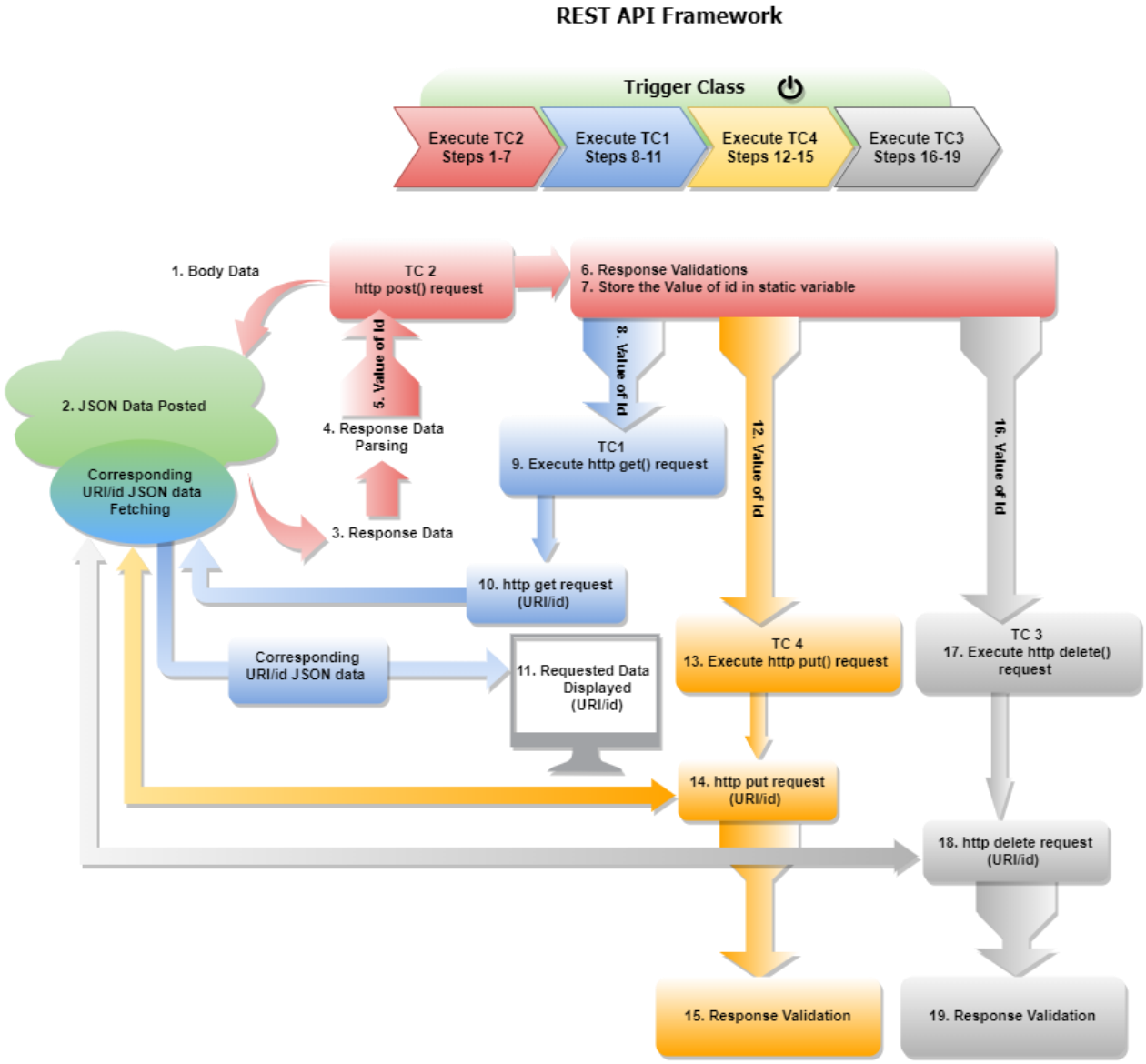


Chapter-8 Flowchart 1

Implementation

Now, just look at the framework workflow as shown below. You should follow first the color coding of the components of the particular Test Case like pink for TC2 and sky blue for TC1 and then follow the step numbers in sequence.

We will be start with the trigger class in which we will be defining the sequence of our test cases as per our requirements A,B,C and D discussed above.



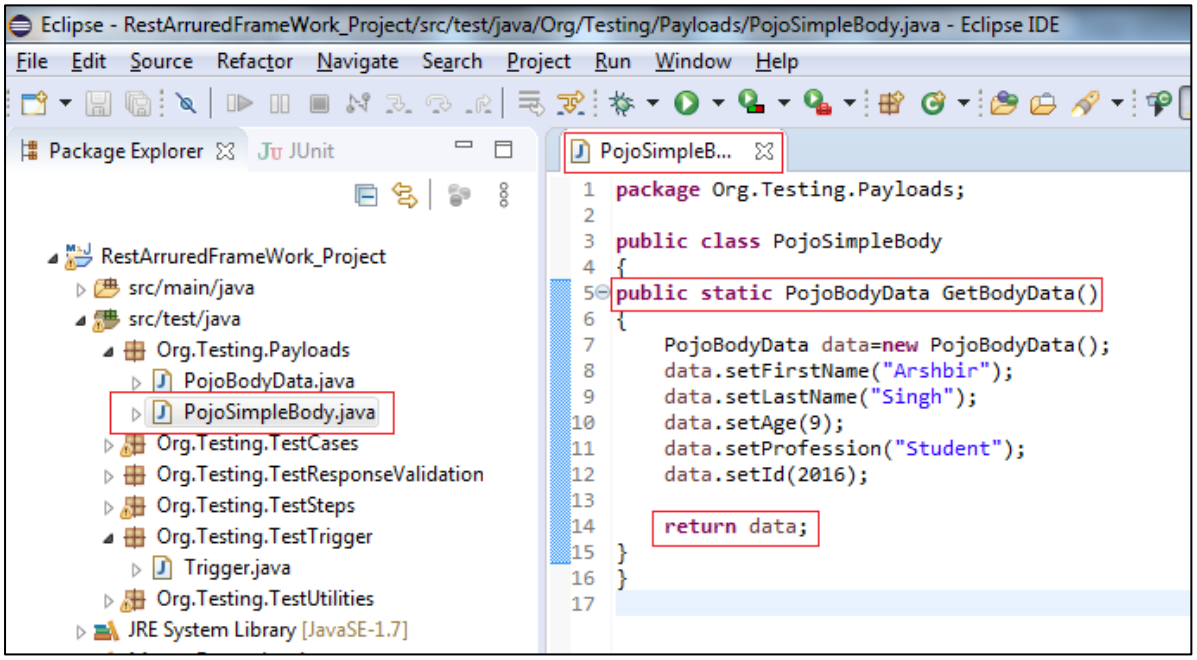
Chapter-8 Diagram 2

Implementing The Requirements 'A'

The project will start its execution from TC2 and will execute the steps 1-7 in which all our requirements of "*A. Expectations from the post() request*" will be completed. Let's began with implementing TC2 requirements:

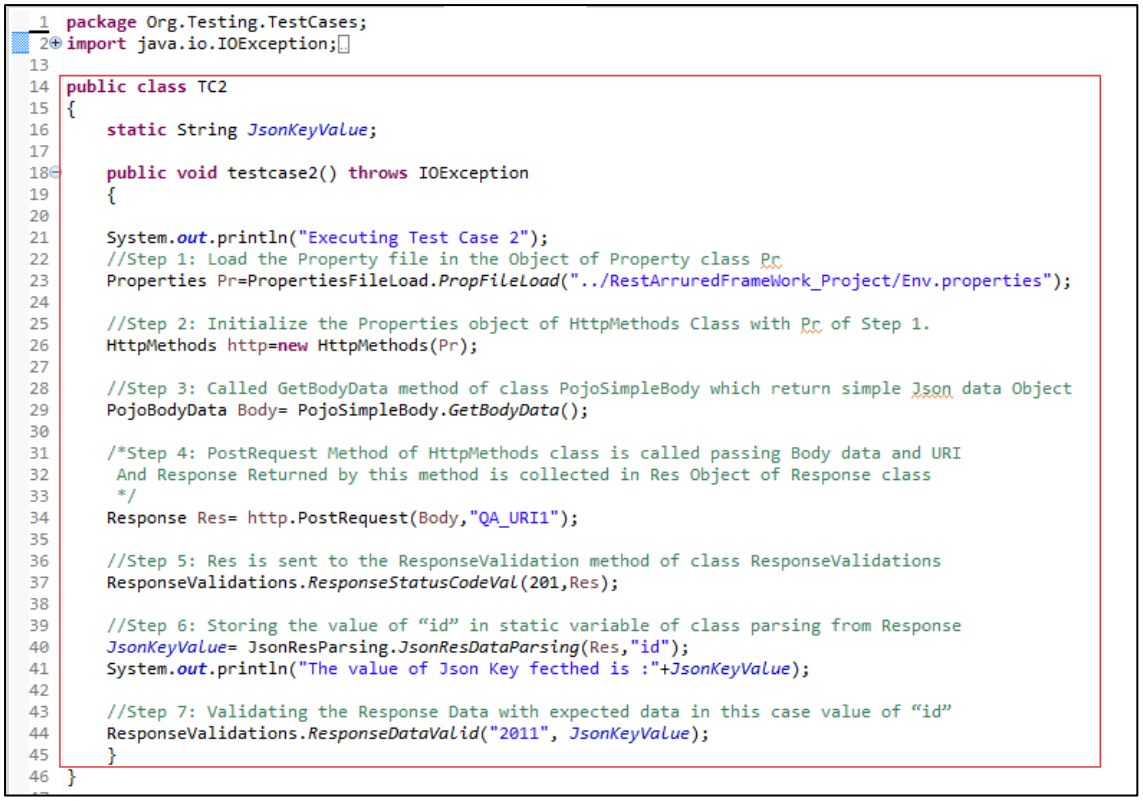
1. To hit the post() request
2. To validate the Response Status Code and Data
3. To Parse the Response data and fetch value of “id” key
4. To store that value of “id” key in static variable so that it can be accessed by using its class name by any of the test case in this project.

We have already created a payload, we will be using this method to create simple json data and post it through post request:



Please note the value of id=2011.

Update the code according the expectations that we have from TC2



```
1  package Org.Testing.TestCases;
2  import java.io.IOException;
13
14 public class TC2
15 {
16     static String JsonKeyValue;
17
18     public void testcase2() throws IOException
19     {
20
21     System.out.println("Executing Test Case 2");
22     //Step 1: Load the Property file in the Object of Property class Pr
23     Properties Pr=PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
24
25     //Step 2: Initialize the Properties object of HttpMethods Class with Pr of Step 1.
26     HttpMethods http=new HttpMethods(Pr);
27
28     //Step 3: Called GetBodyData method of class PojoSimpleBody which return simple Json data Object
29     PojoBodyData Body= PojoSimpleBody.GetBodyData();
30
31     /*Step 4: PostRequest Method of HttpMethods class is called passing Body data and URI
32      And Response Returned by this method is collected in Res Object of Response class
33      */
34     Response Res= http.PostRequest(Body,"QA_URI1");
35
36     //Step 5: Res is sent to the ResponseValidation method of class ResponseValidations
37     ResponseValidations.ResponseStatusCodeVal(201,Res);
38
39     //Step 6: Storing the value of "id" in static variable of class parsing from Response
40     JsonKeyValue= JsonResParsing.JsonResDataParsing(Res,"id");
41     System.out.println("The value of Json Key fecthed is :"+JsonKeyValue);
42
43     //Step 7: Validating the Response Data with expected data in this case value of "id"
44     ResponseValidations.ResponseDataValid("2011", JsonKeyValue);
45     }
46 }
```

Explanation:

```
public class TC2
{
	static String JsonKeyValue;

	public void testcase2() throws IOException
	{

	System.out.println("Executing Test Case 2");
	//Step 1: Load the Property file in the Object of Property class Pr
	Properties
Pr=PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");

//Step 2: Initialize the Properties object of HttpMethods Class with Pr of Step 1.
	HttpMethods http=new HttpMethods(Pr);

//Step 3: Called GetBodyData method of class PojoSimpleBody which return simple Json
data Object
	PojoBodyData Body= PojoSimpleBody.GetBodyData();

/*Step 4: PostRequest Method of HttpMethods class is called passing Body data and URI
And Response Returned by this method is collected in Res Object of Response class*/
	Response Res= http.PostRequest(Body,"QA_URI1");

//Step 5: Res is sent to the ResponseValidation method of class ResponseValidations
	ResponseValidations.ResponseStatusCodeVal(201,Res);
	System.out.println("Status code checked and found Ok");

//Step 6: Storing the value of "id" in static variable of class parsing from Response
	JsonKeyValue= JsonResParsing.JsonResDataParsing(Res,"id");
	System.out.println("The value of Json Key fecthed is :"+JsonKeyValue);
```

```
//Step 7: Validating the Response Data with expected data in this case value of "id"
        ResponseValidations.ResponseDataValid("2016", JsonKeyValue);
        }
}
```

We are already familiar with Steps:1 to Steps: 5 are already, lets discuss Steps: 5,6 and Step 7

Explanation Step 5:

```
ResponseValidations.ResponseStatusCodeVal(201,Res);
```

We are validating the Status code by passing the value of expected and actual status code of the request to the ResponseStatusCodeVal method of ResponseaValidations class which we have updated in the Part-II section of this chapter.

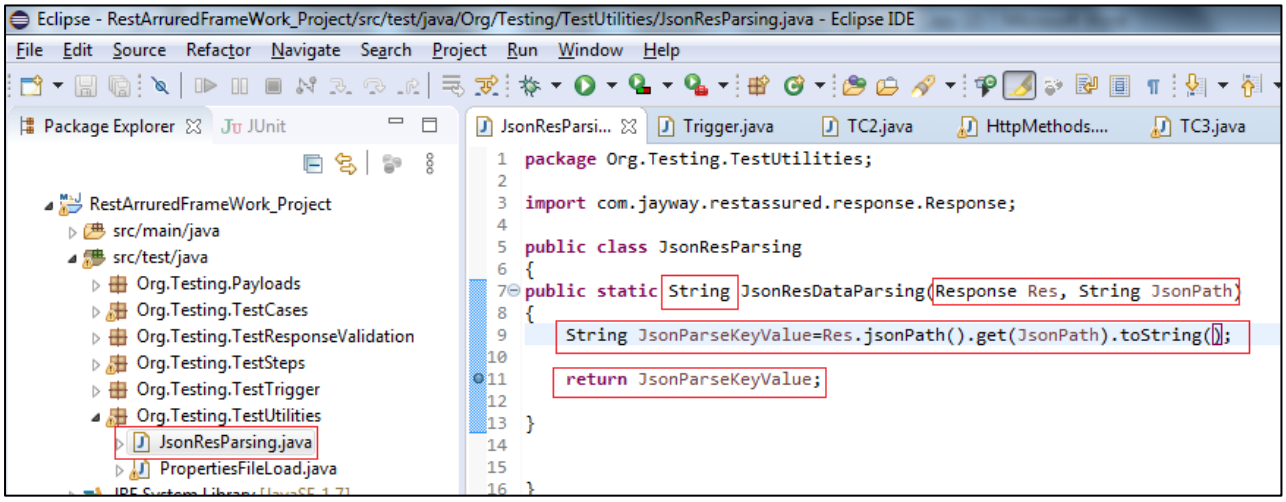
Explanation Step 6:

```
JsonKeyValue= JsonResParsing.JsonResDataParsing(Res,"id");
System.out.println("The value of Json Key fecthed is :"+JsonKeyValue);
```

In this step JsonResDataParsing() method is called passing two arguments Object Res of Response and Json path of the key whose value is expected to be parsed from the response data. In the next instruction value of parsed data is displayed.

Let's create the definition of this method also:

Create one class JsonResParsing under org.testing.Utilities package and create one method JsonResDataParsing() method. We also have to declare two instances one object of type Response to hold the Response data and other of type String to hold the Json Path



Finally we have to return the data corresponding to the Json path and we know that the data in Json is in the form of String and hence so the return type of this method is set as String.

Explanation Step 7:

We have to validate our response data too and for this we have to pass expected data and actual data to some function that must do this for us whenever it is being called from any of the test case.

Under ResponseValidations class, create one method ResponseDataValid() which passes the expected data and actual data (actual data must be parsed from response and then pass these values) as an arguments. In our scenario we have parsed the value of "id" and stored in static variable so let's we will be validate the value of "id" only.

```
ResponseValidations.ResponseDataValid("2016", JsonKeyValue);
```

Let's see the definition of this method:

```
1  package Org.Testing.TestResponseValidation;
2  import org.testng.Assert;
4
5  public class ResponseValidations
6  {
7
8      public static void ResponseStatusCodeVal(int ExpStatCode, Response Res)
9      {
10         Assert.assertEquals(Res.getStatusCode(), ExpStatCode);
11         System.out.println("Response Status Validated");
12     }
13
14     public static void ResponseDataValid(String ExpData, String ActualData)
15     {
16         Assert.assertEquals(ExpData, ActualData );
17         System.out.println("Response Data Validated");
18     }
19 }
20
```

Explanation:

```
public static void ResponseStatusCodeVal(int ExpStatCode, Response Res)
	{
		Assert.assertEquals(Res.getStatusCode(), ExpStatCode);
		System.out.println("Response Status Validated");
	}
```

In the first method `ResponseStatusCodeVal(int ExpStatCode, Response Res)` two parameters, `int ExpStatCode` holding expected Status Code and object of Response class `Response Res` are accepted by the method and are compared using Assert. On confirmation, message gets displayed.

```
public static void ResponseDataValid(String ExpData, String ActualData)
	{
		Assert.assertEquals(ExpData, ActualData );
		System.out.println("Response Data Validated");
	}
```

In the second method `ResponseDataValid(String ExpData, String ActualData)` two parameters, `String ExpData` holding expected data and `String ActualData` holding actual data are accepted by the method and are compared using Assert. On confirmation, message gets displayed.

Implementing The Requirements 'B'

Moving further, the execution of TC2, the Trigger Class will start its executing TC1 and will execute the steps 8-15 in which all our requirements of "*B. Expectations from the get() request*" will be completed. Let's began with implementing TC1 requirements:

1. To the hit the get() request for fetching that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.
2. To display the JSON data fetched by get() request for this particular "id" on console
3. To validate the Response Status Code and Data for this get() request

```
1  package Org.Testing.TestCases;
2
3  import java.io.IOException;
10
11 public class TC1
12 {
13
14     public void testcase1() throws IOException
15     {
16
17         System.out.println("Executing Test Case 1");
18         // STEP 1: Loading the property file in Pr object and we will pass it in calling method
19         Properties Pr= PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
20
21         // STEP 2: Initializing the HttpMethods class with latest value of Properties Object
22         HttpMethods http=new HttpMethods(Pr);
23
24         //STEP 3: Calling the GetRequest method, Passing URI Key and Collecting Response
25         Response Res=http.GetRequest(TC2.JsonKeyValue,"QA_URI1");
26
27         //STEP 4: Validating the Response by calling ResponseValidation method and passing Response data
28         ResponseValidations.ResponseStatusCodeVal(200,Res);
29
30         //STEP 5: Printing the Response Data on the Console for particular id
31         String ParseData=JsonResParsing.JsonResDataParsing(Res,"firstName");
32         System.out.println("The value of Json Key fecthed is :"+ParseData);
33         System.out.println(Res.asString());
34         ResponseValidations.ResponseDataValid("Arshbir", ParseData);
35     }
36 }
37
```

Explanation:

```
public void testcase1() throws IOException
	{

		System.out.println("\nExecuting Test Case 1");
	// STEP 1: Loading the property file in Pr object and we will pass it in calling
	method
	Properties Pr=
PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");

	// STEP 12Intitalizing the HttpMethods class with latest value of Properties
	Object
		HttpMethods http=new HttpMethods(Pr);

	//STEP 3: Calling the GetRequest method, Passing URI Key and Collecting Response
		Response Res=http.GetRequest(TC2.JsonKeyValue,"QA_URI1");

	 //STEP 4: Validating the Response by calling ResponseValidation method and
	 passing Response data
		ResponseValidations.ResponseStatusCodeVal(200,Res);

	//STEP 5: Printing the Response Data on the Console for particular id
		String ParseData=JsonResParsing.JsonResDataParsing(Res,"firstName");
		System.out.println("The value of Json Key fecthed is :"+ParseData);
		ResponseValidations.ResponseDataValid("Arshbir", ParseData);

		System.out.println(Res.asString()+"\n");
	}
```

We are already familiar with steps 1 and 2 and lets discuss steps 3, 4 and 5

Explanation Step 3:

```
Response Res=http.GetRequest(TC2.JsonKeyValue,"QA_URI1");
```

Method `GetRequest(TC2.JsonKeyValue,"QA_URI1")` is called and this method passes two arguments, the first argument is static variable of TC2 holding the parse data and is used to fetch Json data that corresponds to the value passed by `TC2.JsonKeyValue` (value of key "id" in our case) and second is the URI string

Explanation Step 4:

```
ResponseValidations.ResponseStatusCodeVal(200,Res);
```

We are validating the Status code by passing the value of expected and actual status code of the request to the ResponseStatusCodeVal method of ResponseaValidations class which we have updated in the Part-II section of this chapter.

Explanation Step 5:

```
String ParseData=JsonResParsing.JsonResDataParsing(Res,"firstName");
System.out.println("The value of Json Key fecthed is :"+ParseData);
```

In step JsonResDataParsing() method is called passing two arguments Object Res of Response and Json path of the key whose value is expected to be parsed from the response data. In the next instruction value of parsed data is displayed.

Parse data is validated with below method called in which two arguments are passed. The first one is the expected data string and second is the parse data that is to be validated

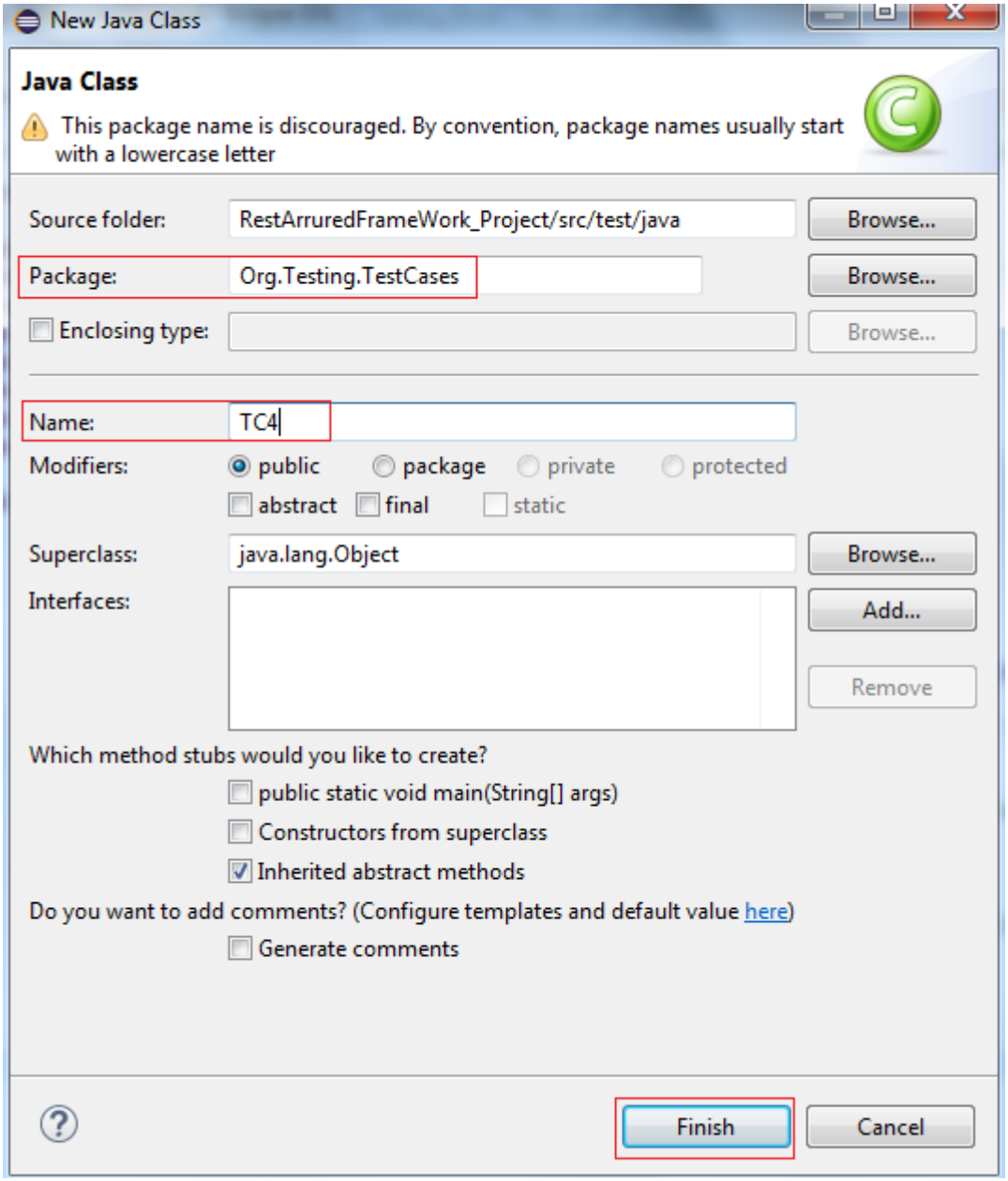
```
ResponseValidations.ResponseDataValid("Arshbir", ParseData);
```

Finally, the response data is displayed

```
System.out.println(Res.asString()+"\n");
```

Implementing The Requirements 'C'

Create test case TC4 under package Org.Testing.TestCases for implementing the requirements.



Next to that the control in the Trigger class moves to the next instruction which is execution of the TC4 for the put() request and following tasks are to be implemented in TC4:

1. To hit the put() request for updating that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" in that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.
2. To update the JSON data say firstName
3. To validate the updated JSON data by parsing the new updated values of key(s) from Response Data
4. To validate the Response Status Code

```
1  package Org.Testing.TestCases;
2
3  import java.io.IOException;
12
13 public class TC4
14 {
15     public void testcase4() throws IOException
16     {
17      System.out.println("Executing Test Case 4");
18      // STEP 1: Loading the property file in Pr object and we will pass it in calling method
19      Properties Pr= PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");
20
21      // Step 2: Called GetUpdatedBodyData method of class PojoSimpleUpdatedData which return simple Json data Object
22      PojoBodyData Body= PojoSimpleUpdatedData.GetUpdatedBodyData();
23
24      // STEP 3: Initializing the HttpMethods class with latest value of Properties Object
25      HttpMethods http=new HttpMethods(Pr);
26
27      //STEP 4: Calling the put() Request method, Passing URI Key, updated data and Collecting Response
28      Response Res=http.PutRequest(Body,"QA_URI1",TC2.JsonKeyValue);
29      System.out.println("Data Updated by put() http request");
30
31      //STEP 5: Validating the Response by calling ResponseValidation method and passing Response data
32      ResponseValidations.ResponseStatusCodeVal(200,Res);
33      String UpdatedData= JsonResParsing.JsonResDataParsing(Res,"lastName");
34      System.out.println("The value of Updated Json Key fecthed is :"+UpdatedData);
35
36      ResponseValidations.ResponseDataValid("Singh Bhatti", UpdatedData);
37     }
38 }
```

Explanation:

```
        public void testcase4() throws IOException
        {
        System.out.println("Executing Test Case 4");
         // STEP 1: Loading the property file in Pr object and we will pass it in
         calling method
         Properties Pr=
PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");

         //    Step 2: Called GetUpdatedBodyData method of class PojoSimpleUpdatedData
        which return simple Json data Object
         PojoBodyData Body= PojoSimpleUpdatedData.GetUpdatedBodyData();

        // STEP 3: Initializing the HttpMethods class with latest value of Properties
        Object
         HttpMethods http=new HttpMethods(Pr);

        //STEP 4: Calling the put() Request method, Passing URI Key, updated data and
        Collecting Response
         Response Res=http.PutRequest(Body,"QA_URI1",TC2.JsonKeyValue);
         System.out.println("Data Updated by put() http request");
```

```
        //STEP 5: Validating the Response by calling ResponseValidation method and
       passing Response data
        ResponseValidations.ResponseStatusCodeVal(200,Res);
        String UpdatedData= JsonResParsing.JsonResDataParsing(Res,"lastName");
        System.out.println("The value of Updated Json Key fecthed is :"+UpdatedData);

        ResponseValidations.ResponseDataValid("Singh Bhatti", UpdatedData);

       }
```

Explanation:

Step No:1,2,3 and 4 are very much the same as that of the other test case just the point here to be noted is the Body object of the PojoBodydata stores the update information which we have stored in the new PojoSimpleUpdatedData created under the org.testing.Payloads package. The record to be upated is targeted by the value of TC2.JsonKeyValue which holds the value of "id" parsed in TC2

```
Response Res=http.PutRequest(Body,"QA_URI1",TC2.JsonKeyValue);
```

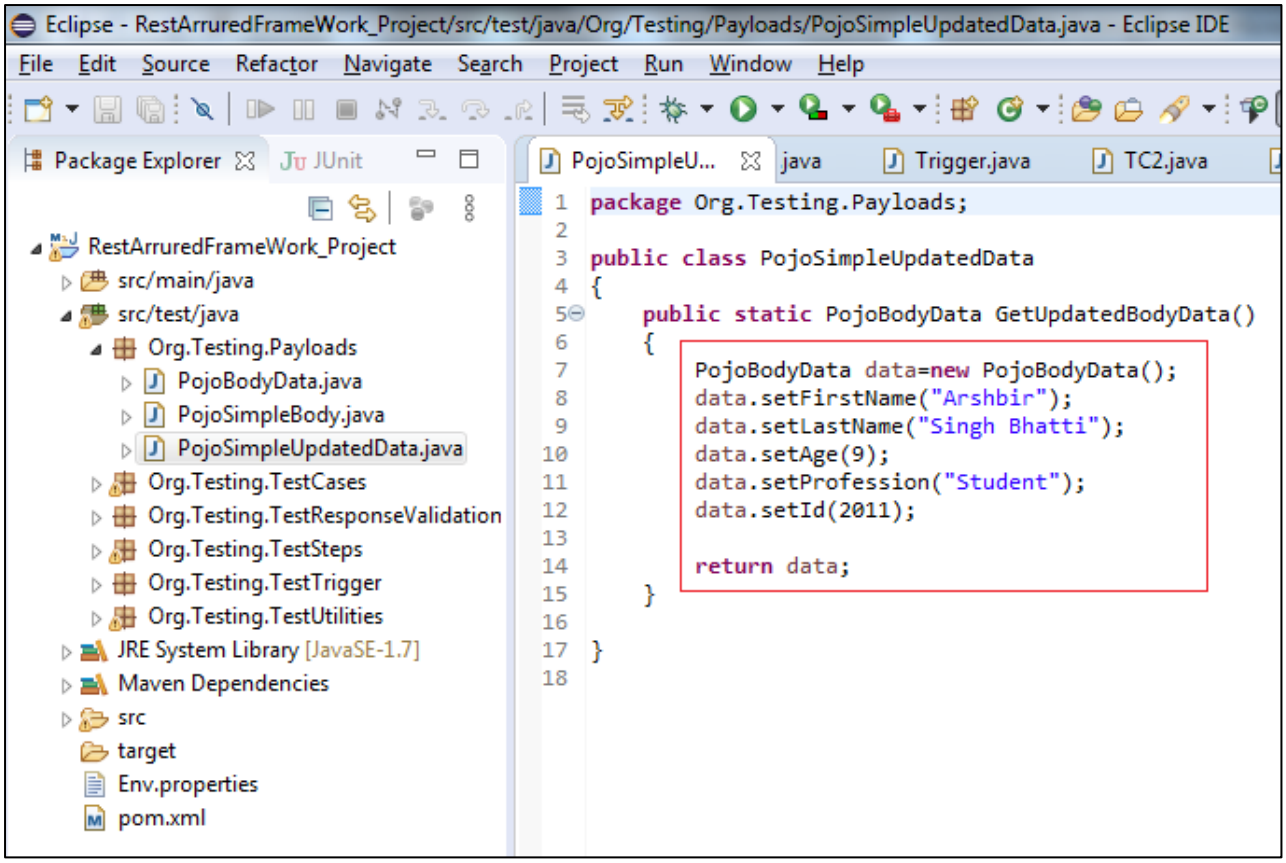
And the first argument in the method `PutRequest` is a URI key which is passed to the definition of this method in the HttpMethods class as shown below:

```
public Response PutRequest(PojoBodyData Body, String URI, String id)
   {
        String FinalURI=(String) Pr.getProperty(URI)+"/"+id;
        Response Res=
                given()
                .contentType(ContentType.JSON)
                .body(Body)
                .when()
                .put(FinalURI);
        return Res;
   }
```

Here, the final URI is created by concatenating the URI that is fetched from the Env.properties file corresponding to the value of key "QA_URI1" with the value of Id that is being fetched from the response data in TC2 that is `TC2.JsonKeyValue` and is passed here by the method declaration as discussed above also.

To get the updated data from we have created another class named: PojiSimpleUpdatedData and under this class we have defined one method GetUpdatedBodyData() which returns an object of PojoBodyData class.

```
PojoBodyData Body= PojoSimpleUpdatedData.GetUpdatedBodyData();
```



Data is created just as we before created with the class PojoSimpleBody.java which is called in TC2 to post data but here the value of lastName is changed so as to updated the data and so practice the put() request.

In explanation to step 6:

```
ResponseValidations.ResponseStatusCodeVal(200,Res);
```

We are validating the response status code by calling the ResponseStatusCodeVal method as we did in earlier test cases.

```
String UpdatedData= JsonResParsing.JsonResDataParsing(Res,"lastName");
```

In the above instruction we are parsing the response data at the ket lastName and this is being stored in a String Variable UpdatedData and simply this data is printer on the console with below statement.

```
System.out.println("The value of Updated Json Key fecthed is :"+UpdatedData);
```

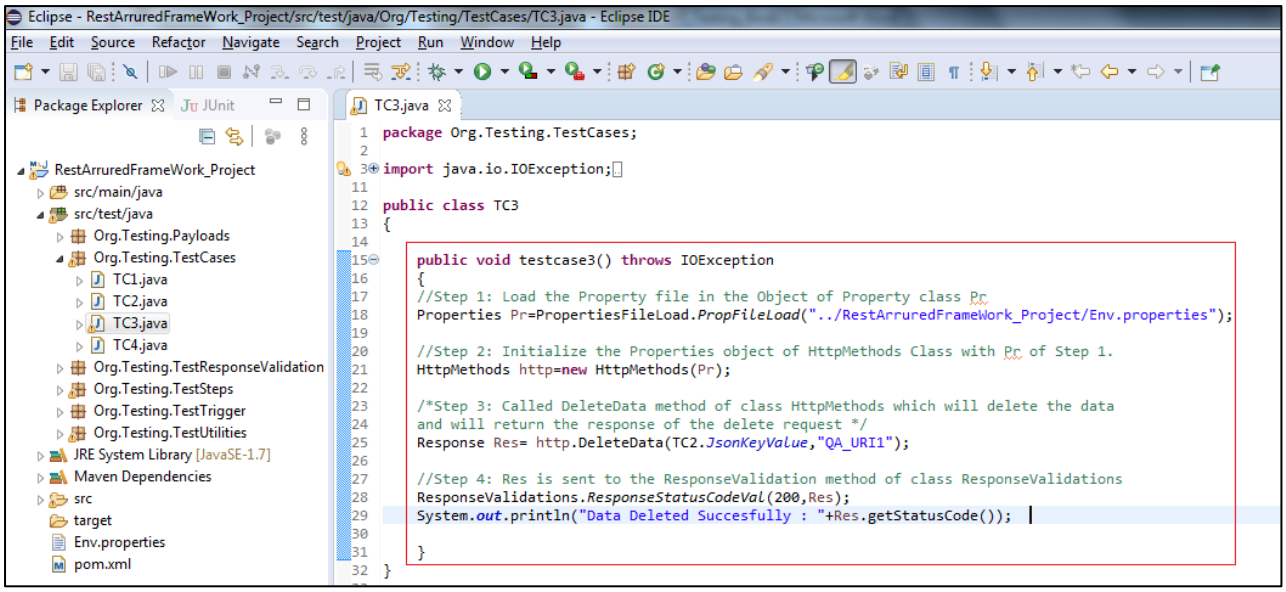
Finally, we validate updated response data for the value of key by calling ResponseDataValid method of the class in which we are passing two arguments first one as a String (which is updated data) and the second is the parsed updated response data

```
ResponseValidations.ResponseDataValid("Singh Bhatti", UpdatedData);
```

Implementing the Requirements 'D'

Finally, the control passes to the last step in the Trigger Class that is executing the TC3 the delete() request. TC3 is very much the same we have already created in the Part-I of this chapter but with few updations. Let's implement the requirements of TC3

1. To hit the delete() request for deleting that particular JSON data that was posted by post request in TC2. It will be done by using the value of "id" in that was already parsed from response data in TC2 and is stored in one of the static variable in TC2.
2. To delete the JSON data
3. To validate the Response Status Code



Explanation:

```
public class TC3
{
	public void testcase3() throws IOException
	{
//Step 1: Load the Property file in the Object of Property class Pr
	Properties
Pr=PropertiesFileLoad.PropFileLoad("../RestArruredFrameWork_Project/Env.properties");

//Step 2: Initialize the Properties object of HttpMethods Class with Pr of Step 1.
	HttpMethods http=new HttpMethods(Pr);

/*Step 3: Called DeleteData method of class HttpMethods which will delete the data
and will return the response of the delete request */

	Response Res= http.DeleteData(TC2.JsonKeyValue,"QA_URI1");

//Step 4: Res is sent to the ResponseValidation method of class ResponseValidations

	ResponseValidations.ResponseStatusCodeVal(200,Res);
	System.out.println("Data Deleted Succesfully : "+Res.getStatusCode());
	}
}
```

Here, all the steps are seems to be very much familiar and are in the sequence with the requirements 'D' of the implementation. The main part to discuss here is about the instruction is the method:

```
Response Res= http.DeleteData(TC2.JsonKeyValue,"QA_URI1");
```

DeleteData method of http class is having two arguments, the first one is the value of id that was parsed in TC2 and is accessed using `TC2.JsonKeyValue` and the second argument is the key against the URI value stored in the file Env.properties

The definition of the DeleteData() method under the class HttpMethods() looks something like this:

```
public Response DeleteData(String id, String URI)
{
    String FinalURI=(String) Pr.getProperty(URI)+"/"+id;
    Response Res=
            given()
            .contentType(ContentType.JSON)
            .when()
            .delete(FinalURI);
    return Res;
}
```

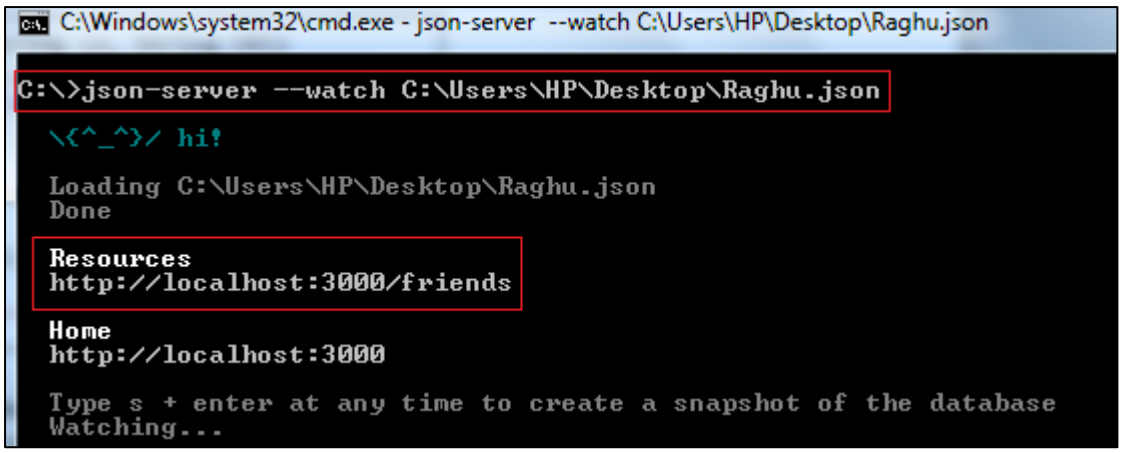
Here, the final URI is created by concatenating the URI that is fetched from the Env.properties file corresponding to the value of key "QA_URI1" with the value of Id that is being fetched from the response data in TC2 that is `TC2.JsonKeyValue` and is passed here by the method declaration as discussed above also.

The last part of implementation id to update the Trigger Class with this sequence TC2, TC1, TC4 and TC3 and the below code is very much self-explanatory:

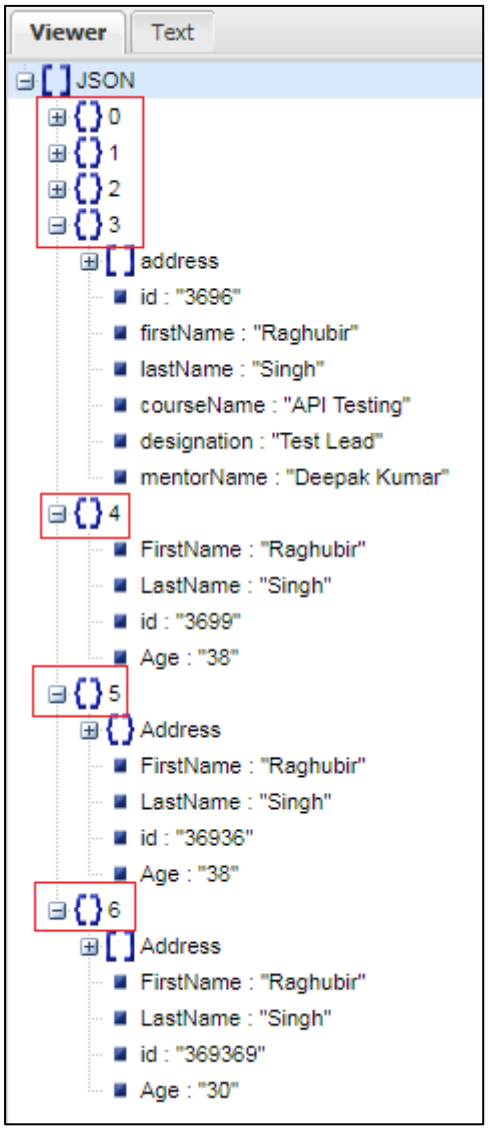
```
1  package Org.Testing.TestTrigger;
2
3  import java.io.IOException;
9
10 public class Trigger
11 {
12 public static void main(String[] args) throws IOException
13 {
14
15     System.out.println("Called Test Case 2- post() Request");
16     TC2 TC2Obj=new TC2();
17     TC2Obj.testcase2();
18     System.out.println("--------------End of Test Case 2 the post() Request--------------");
19     System.out.println("\n");
20
21     System.out.println("Called Test Case 1- get() Request");
22     TC1 TC1Obj=new TC1();
23     TC1Obj.testcase1();
24     System.out.println("--------------End of Test Case 1 the get() Request--------------");
25     System.out.println("\n");
26
27     System.out.println("Called Test Case 4- put() Request");
28     TC4 TC4Obj=new TC4();
29     TC4Obj.testcase4();
30     System.out.println("--------------End of Test Case 4 the put() Request--------------");
31     System.out.println("\n");
32
33     System.out.println("Called Test Case 3- delete() Request");
34     TC3 TC3Obj=new TC3();
35     TC3Obj.testcase3();
36     System.out.println("--------------End of Test Case 3 the delete() Request--------------");
37 }
38 }
```

Now let's start the execution of Trigger Class:

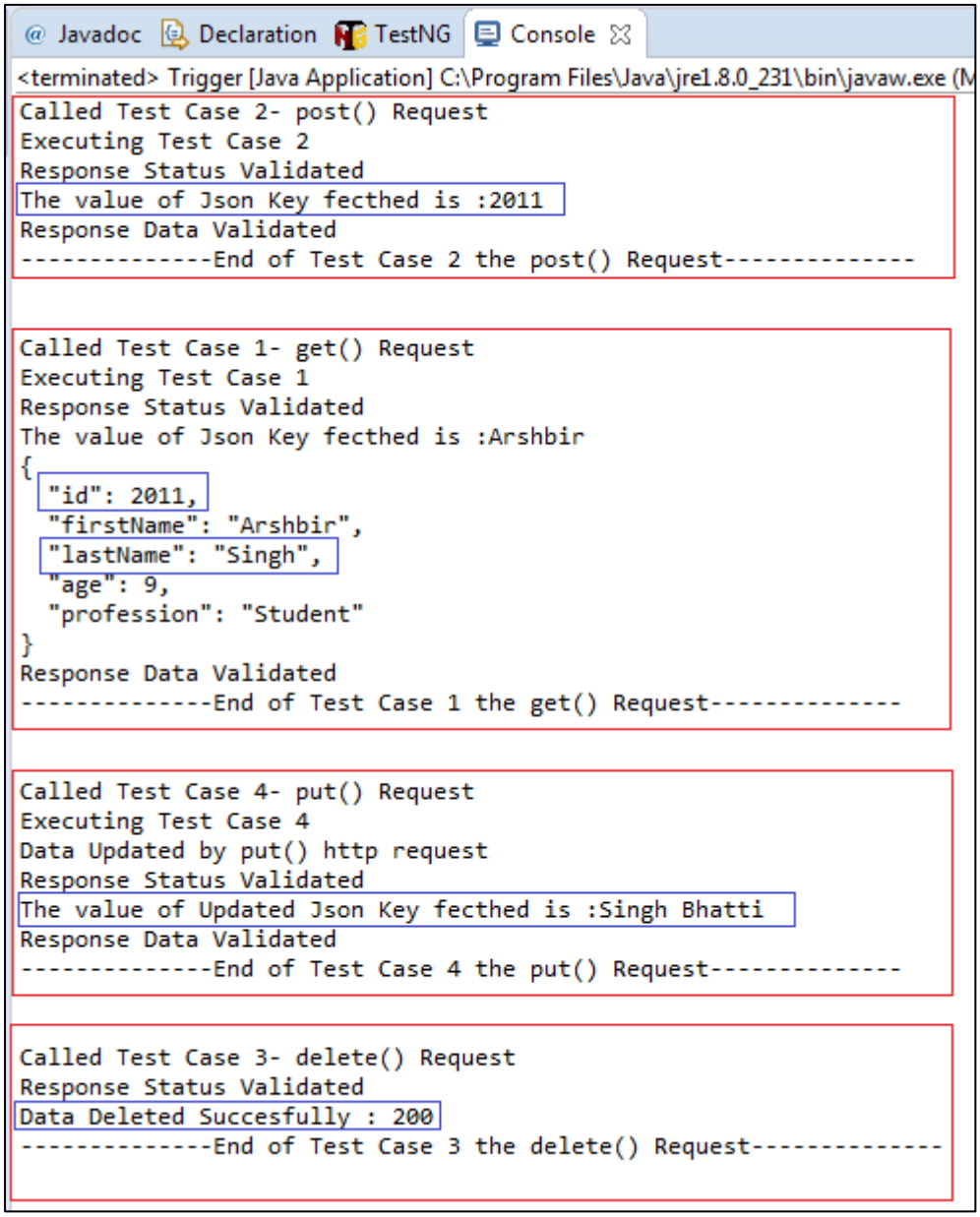
1. Start the Json Server with Json file in which data is to posted, fetched, updated and deleted



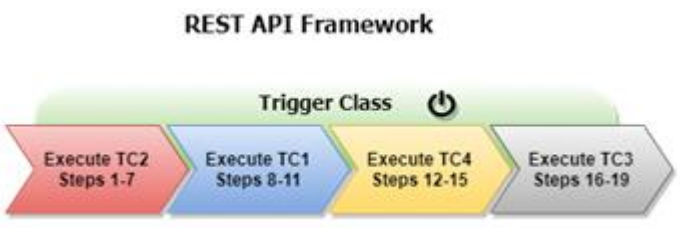
2. Check the data in the data at the URI file using online tool at: http://jsonviewer.stack.hu/ by copy the entire data and pasting it in Text tab and then clicking the Viewer tab as shown below:



Let's run the Trigger class and here we go:



The output is exactly what we have implemented



Chapter -9: Introduction to TestNG

TestNG: Annotations, Framework with REST API using Java

Before moving to TestNG it is very important to note that by just adding maven dependency or corresponding .jar file of TestNG is not sufficient. Setup for the TestNG package is also required for using the TestNG framework. Details of TestNG details are out of the scope of this book.

Introduction: TestNG is an automation testing framework where NG stands for "Next Generation". TestNG is the advanced influencer of JUnit and uses annotations (@). TestNG overcomes the limitations of JUnit.

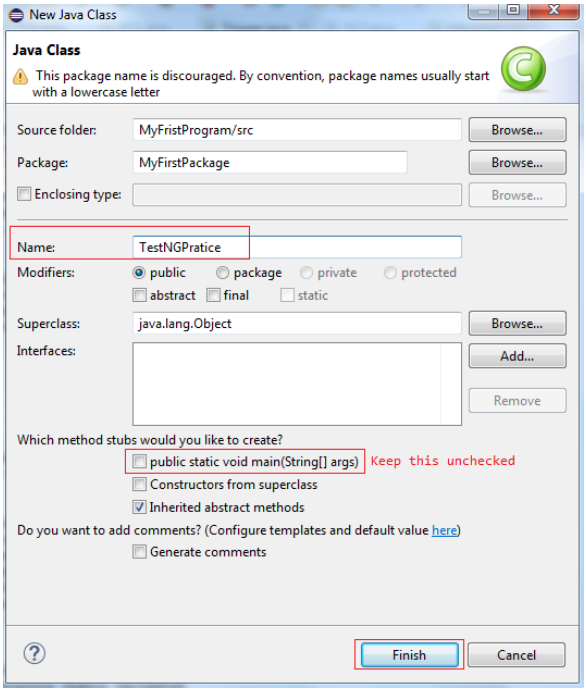
Using TestNG, reports can be generated conveniently, and it gives instance picture of the test execution with details of total Pass/Fail/Skipped test cases and hence it facilitates to execute the failed test case in Isolation. It is very easy to use Assertions and multiple cases together in using TestNG.

It has many annotations but the most important for us at the moment are

@Test: To run the test case
@BeforeTest: To run something before test
@BeforeMethod: To run something before every method (@Test annotation methods)
@AfterMethod: To run something after every method (@Test annotation methods)
@AfterTest: To run something after test

Java program uses public static void main(String[] args) to run the program but in TestNG we wont use this main method.
Instead of main method to execute the program we us @Test annotation

Create one class say TestNGPractice (although you can give any name to class)

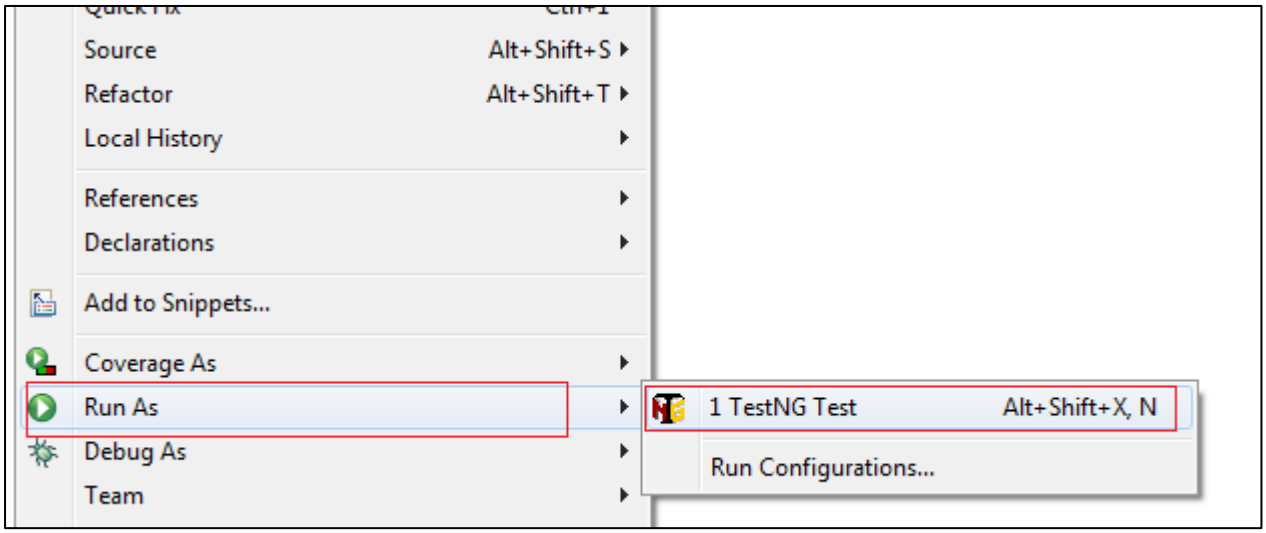


Just write the following simple code:

```
1  package MyFirstPackage;
2
3  import org.testng.annotations.*;
4
5  public class TestNGPratice
6  {
7
8      @Test
9      public void testcase1()
10 {
11     System.out.println("This is my first TestNG Program");
12 }
13 }
```

```
package MyFirstPackage;

import org.testng.annotations.*;

public class TestNGPratice
{

	@Test
	public void testcase1()
{
	System.out.println("This is my first TestNG Program");
}
}
```

Please do import the org.testng.annotations.* to use all the annotations as per requirement. Now right click on the project and select Run As TestNG Test



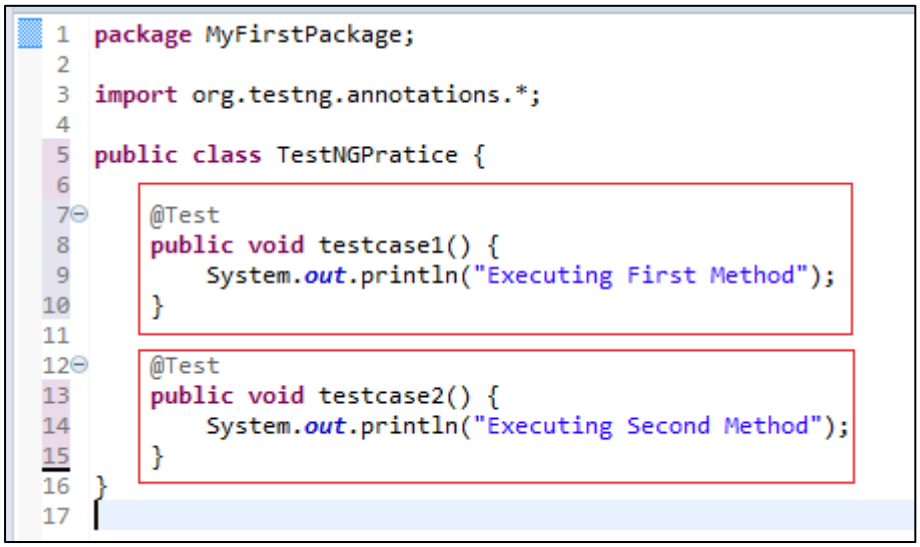
Output on console will displayed something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
This is my first TestNG Program
PASSED: testcase1

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

So we have executed out first simple test case using TestNG.

Executing Multiple Test Cases with TestNG

Now we will add multiple test cases and execute with the help of @Test annotation. Simply copy the code of Test case 1 and rename it to Test case 2. Also change the display method inside the test case as shown below:



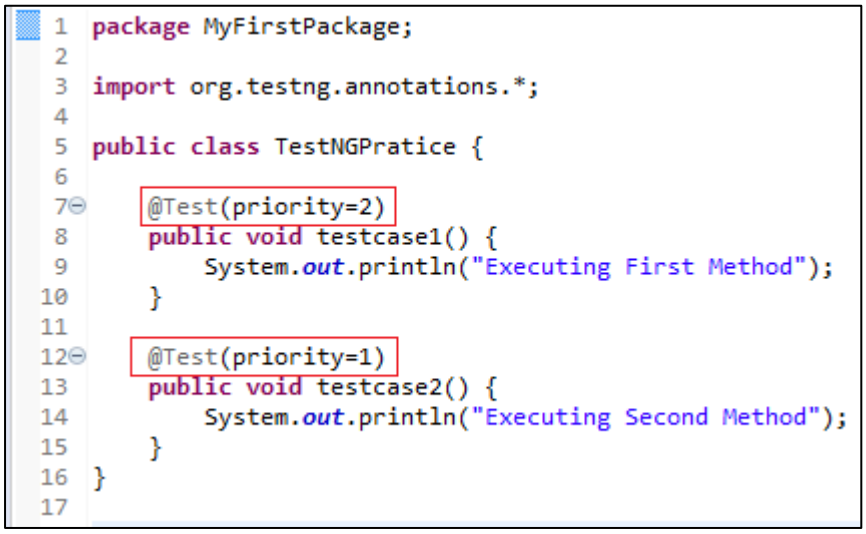

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test
	public void testcase2() {
		System.out.println("Executing Second Method");
	}
}
```

Again right click on the project and select Run As TestNG Test and output will be something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
Executing First Method
Executing Second Method
PASSED: testcase1
PASSED: testcase2

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

Setting The Priorities to the Test Cases in TestNG

By default the Test Cases are run in alphabetical order however we can also set the priority to our test case with the test annotation like as given below:



```
1  package MyFirstPackage;
2
3  import org.testng.annotations.*;
4
5  public class TestNGPratice {
6
7      @Test(priority=2)
8      public void testcase1() {
9          System.out.println("Executing First Method");
10     }
11
12     @Test(priority=1)
13     public void testcase2() {
14         System.out.println("Executing Second Method");
15     }
16 }
17
```

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test(priority=2)
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test(priority=1)
	public void testcase2() {
		System.out.println("Executing Second Method");
	}
}
```

Again right click on the project and select Run As TestNG Test and output will look something like this, please note the sequence of execution of Test Cases based on priority set:

```
[RemoteTestNG] detected TestNG version 7.0.0
Executing Second Method
Executing First Method
PASSED: testcase2
PASSED: testcase1

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

So the test case had been executed based upon their set priority.

Using different annotations: So far we have used @Test annotation to execute our test cases, let's try with the other annotations given below:

@BeforeTest: To run something before test
@BeforeMethod: To run something before every method (@Test annotation methods)
@AfterMethod: To run something after every method (@Test annotation methods)
@AfterTest: To run something after test

We will be using the same code to have the understanding of these annotations. Just update the code in the TestNGPratice class as given below;

The @BeforeTest Annotation: The @BeforeTest annotation is executed before the other methods and by methods here we mean by the test cases.

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test(priority=2)
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test(priority=1)
	public void testcase2() {
		System.out.println("Executing Second Method");
	}
	@BeforeTest
	public void ExecuteBeforeTest()
	{
		System.out.println("This is executed Before Test");
	}
}
```

The code something like this:

```
1  package MyFirstPackage;
2  import org.testng.annotations.*;
3  public class TestNGPratice {
4
5      @Test(priority=2)
6      public void testcase1() {
7          System.out.println("Executing First Method");
8      }
9
10     @Test(priority=1)
11     public void testcase2() {
12         System.out.println("Executing Second Method");
13     }
14     @BeforeTest
15     public void ExecuteBeforeTest()
16     {
17         System.out.println("This is executed Before Test");
18     }
19 }
```

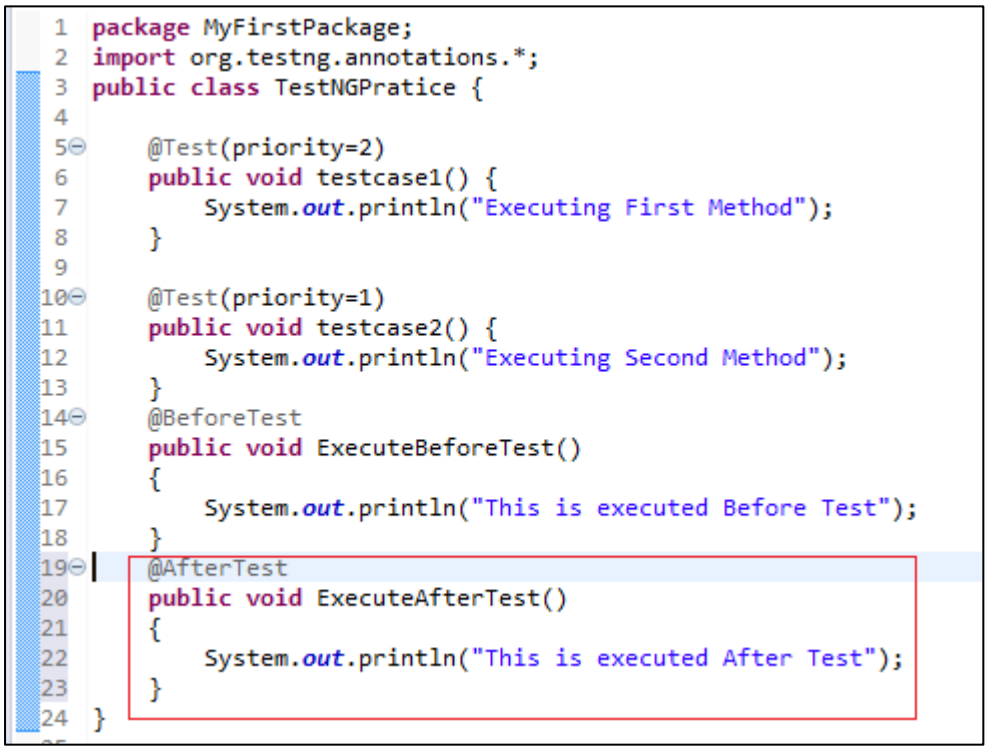
Right click on the project and select Run As TestNG Test and it is expected in the output that @BeforeTest method will be executed only once before an the other test case execution starts and the output will be look something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
This is executed Before Test
Executing Second Method
Executing First Method
PASSED: testcase2
PASSED: testcase1

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

The @AfterTest Annotation: The @AfterTest annotation will executed after all the test case execution gets completed. Update the code in the TestNGPractice class and add @AfterTest annotation as given below

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test(priority=2)
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test(priority=1)
	public void testcase2() {
```

```
		System.out.println("Executing Second Method");
	}
	@BeforeTest
	public void ExecuteBeforeTest()
	{
		System.out.println("This is executed Before Test");
	}
	@AfterTest
	public void ExecuteAfterTest()
	{
		System.out.println("This is executed After Test");
	}
}
```



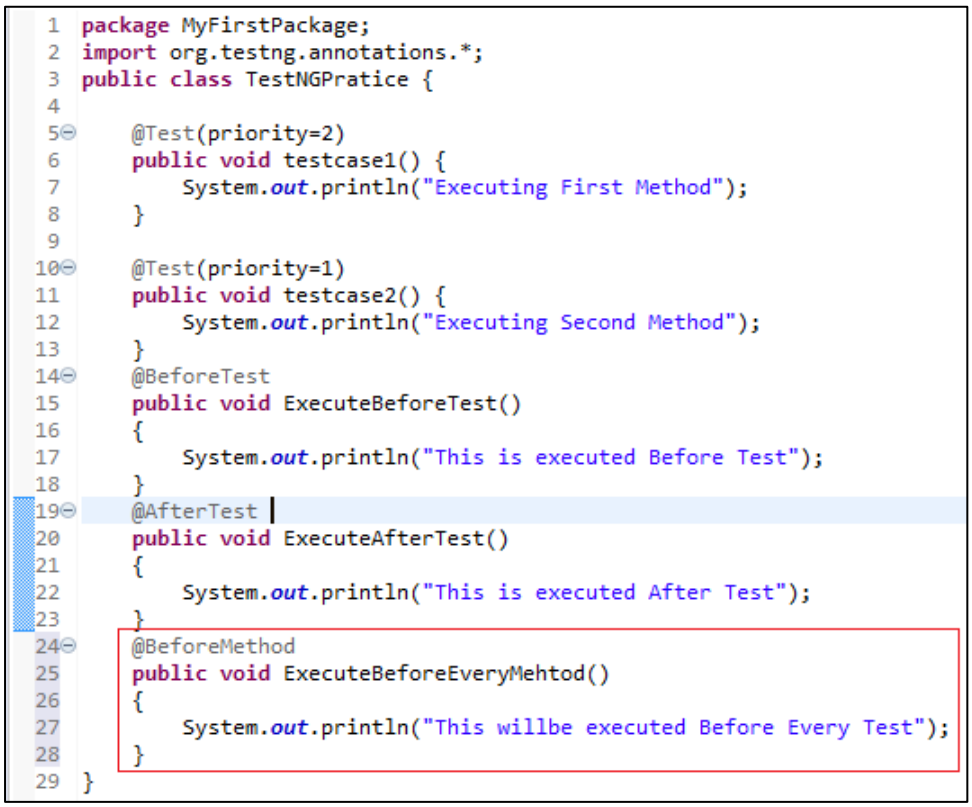
```
1  package MyFirstPackage;
2  import org.testng.annotations.*;
3  public class TestNGPratice {
4
5      @Test(priority=2)
6      public void testcase1() {
7          System.out.println("Executing First Method");
8      }
9
10     @Test(priority=1)
11     public void testcase2() {
12         System.out.println("Executing Second Method");
13     }
14     @BeforeTest
15     public void ExecuteBeforeTest()
16     {
17         System.out.println("This is executed Before Test");
18     }
19     @AfterTest
20     public void ExecuteAfterTest()
21     {
22         System.out.println("This is executed After Test");
23     }
24 }
```

Right click on the project and select Run As TestNG Test and it is expected in the output that @AfterTest method will be executed only once After all the other Test Cases execution gets completed and the output will be look something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
This is executed Before Test
Executing Second Method
Executing First Method
This is executed After Test
PASSED: testcase2
PASSED: testcase1

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================
```

The @BeforeMethod Annotation: Code under this method will be executed *once before every test case execution begins.* Let's update the code for understanding @BeforeMethod. Update the code in the TestNGPractice class and add @BeforeMethod annotation as given below:

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test(priority=2)
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test(priority=1)
	public void testcase2() {
		System.out.println("Executing Second Method");
	}
	@BeforeTest
	public void ExecuteBeforeTest()
	{
		System.out.println("This is executed Before Test");
	}
	@AfterTest
	public void ExecuteAfterTest()
	{
		System.out.println("This is executed After Test");
	}
	@BeforeMethod
	public void ExecuteBeforeEveryMehtod()
	{
		System.out.println("This willbe executed Before Every Test");
	}
}
```



Right click on the project and select Run As TestNG Test and it is expected in the output that @BeforeMethod method will be executed once before every Test Cases execution begins and the output will be look something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
This is executed Before Test
This willbe executed Before Every Test
Executing Second Method
This willbe executed Before Every Test
Executing First Method
This is executed After Test
PASSED: testcase2
PASSED: testcase1

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

The @AfterMethod Annotation: As @BeforeMethod annotation execute once before execution of every test case and is so the @AfterMethod annotation is which executed once after execution of each test case gets completed.

```
package MyFirstPackage;
import org.testng.annotations.*;
public class TestNGPratice {

	@Test(priority=2)
	public void testcase1() {
		System.out.println("Executing First Method");
	}

	@Test(priority=1)
	public void testcase2() {
		System.out.println("Executing Second Method");
	}
	@BeforeTest
	public void ExecuteBeforeTest()
	{
		System.out.println("This is executed Before Test");
	}
	@AfterTest
	public void ExecuteAfterTest()
	{
		System.out.println("This is executed After Test");
	}
	@BeforeMethod
	public void ExecuteBeforeEveryMehtod()
	{
		System.out.println("This willbe executed Before Every Test");
	}
	@AfterMethod
	public void ExecuteAfterEveryMehtod()
	{
```

```
            System.out.println("This willbe executed After Every Test");
        }
}
```

```
1  package MyFirstPackage;
2  import org.testng.annotations.*;
3  public class TestNGPratice {
4
5      @Test(priority=2)
6      public void testcase1() {
7          System.out.println("Executing First Method");
8      }
9
10     @Test(priority=1)
11     public void testcase2() {
12         System.out.println("Executing Second Method");
13     }
14     @BeforeTest
15     public void ExecuteBeforeTest()
16     {
17         System.out.println("This is executed Before Test");
18     }
19     @AfterTest
20     public void ExecuteAfterTest()
21     {
22         System.out.println("This is executed After Test");
23     }
24     @BeforeMethod
25     public void ExecuteBeforeEveryMehtod()
26     {
27         System.out.println("This willbe executed Before Every Test");
28     }
29     @AfterMethod
30     public void ExecuteAfterEveryMehtod()
31     {
32         System.out.println("This willbe executed After Every Test");
33     }
34 }
```

Right click on the project and select Run As TestNG Test and it is expected in the output that @AfterMethod method will be executed once After every Test Cases execution completes and the output will be look something like this:

```
[RemoteTestNG] detected TestNG version 7.0.0
This is executed Before Test
This willbe executed Before Every Test
Executing Second Method
This willbe executed After Every Test
This willbe executed Before Every Test
Executing First Method
This willbe executed After Every Test
This is executed After Test
PASSED: testcase2
PASSED: testcase1

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Passes: 2, Failures: 0, Skips: 0
===============================================
```

Automating Framework using TestNG

Open the Trigger.java file, remove the main method from the Trigger class and update the Test Cases with TestNG @Test annotation with respective priorities as shown below:

Explanation:

We have set the priority of TC2 as 1 with @Test annotation to this test case

```
@Test(priority=1)
public void second() throws IOException
	{
System.out.println("Called Test Case 2- post() Request");
TC2 TC2Obj=new TC2();
TC2Obj.testcase2();
System.out.println("--------------End of Test Case 2 the post() Request-------------");
System.out.println("\n");
	}
```

We have set the priority of TC1 as 2 with @Test annotation to this test case

```
@Test(priority=2)
public void first() throws IOException
	{
System.out.println("Called Test Case 1- get() Request");
TC1 TC1Obj=new TC1();
TC1Obj.testcase1();
System.out.println("--------------End of Test Case 1 the get() Request--------------");
System.out.println("\n");
	}
```

We have set the priority of TC4 as 3 with @Test annotation to this test case

```
@Test(priority=3)
public void fourth() throws IOException
	{
System.out.println("Called Test Case 4- put() Request");
TC4 TC4Obj=new TC4();
TC4Obj.testcase4();
System.out.println("--------------End of Test Case 4 the put() Request--------------");
System.out.println("\n");
	}
```

We have set the priority of TC3 as 4 with @Test annotation to this test case

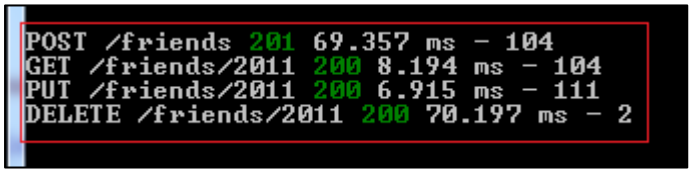
```
@Test(priority=4)
public void third() throws IOException
	{
System.out.println("Called Test Case 3- delete() Request");
TC3 TC3Obj=new TC3();
TC3Obj.testcase3();
System.out.println("-------------End of Test Case 3 the delete() Request------------");
	}
}
```

Please remember that as we have set the priority of each test case so the sequence in which the test case is written in the code is of importance and does not execute in the same sequence in which they are written. Now the execution sequence will be followed by the sequence of priority, lowest the number higher will be the priority.

And the output of the Test Case Execution will be as shown below:

```
[RemoteTestNG] detected TestNG version 7.0.1
Called Test Case 2- post() Request
Executing Test Case 2
Response Status Validated
The value of Json Key fecthed is :2011
Response Data Validated
--------------End of Test Case 2 the post() Request--------------

Called Test Case 1- get() Request
Executing Test Case 1
Response Status Validated
The value of Json Key fecthed is :Arshbir
{
  "id": 2011,
  "firstName": "Arshbir",
  "lastName": "Singh",
  "age": 9,
  "profession": "Student"
}
Response Data Validated
--------------End of Test Case 1 the get() Request--------------

Called Test Case 4- put() Request
Executing Test Case 4
Data Updated by put() http request
Response Status Validated
The value of Updated Json Key fecthed is :Singh Bhatti
Response Data Validated
--------------End of Test Case 4 the put() Request--------------

Called Test Case 3- delete() Request
Response Status Validated
Data Deleted Succesfully : 200
--------------End of Test Case 3 the delete() Request--------------
PASSED: second
PASSED: first
PASSED: fourth
PASSED: third
===============================================
    Default test
    Tests run: 4, Failures: 0, Skips: 0
===============================================

===============================================
Default suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
===============================================
```

Same can also be verified on the command prompt that all the test cases executed in the sequence defined by the priority under @Test annotation:



Appendix -1

Write below data to the db.json file in Notepad and save the file with .json extension

To ensure validation of the json file just copy the data on online tool: https://jsonlint.com

```
{
  "friends": [
    {
      "firstname": "Raghubir",
      "lastname": "Singh",
      "id": "manoj",
      "age": 38
    },
    {
      "firstname": "Tejinder",
      "lastname": "Bhatti",
      "id": "deepak",
      "age": 30
    }
  ],
  "posts": [
    {
      "id": 1,
      "title": "json-server",
      "author": "typicode"
    }
  ],
  "comments": [
    {
      "id": 1,
      "body": "some comment",
      "postId": 1
    }
  ],
  "profile": {
    "name": "typicode"
  }
}
```

Appendix -2

HTTP Status Codes
This page is created from HTTP status code information:

1xx Informational
100 Continue
101 Switching Protocols
102 Processing (WebDAV)
2xx Success
200 OK
201 Created
202 Accepted
203 Non-Authoritative Information
204 No Content
205 Reset Content
206 Partial Content
207 Multi-Status (WebDAV)
208 Already Reported (WebDAV)
226 IM Used

3xx Redirection
300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
304 Not Modified
305 Use Proxy
306 (Unused)
307 Temporary Redirect
308 Permanent Redirect (experimental)

4xx Client Error
400 Bad Request 401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed
418 I'm a teapot (RFC 2324)
420 Enhance Your Calm (Twitter)
422 Unprocessable Entity (WebDAV)
423 Locked (WebDAV)
424 Failed Dependency (WebDAV)
425 Reserved for WebDAV
426 Upgrade Required
428 Precondition Required
429 Too Many Requests
431 Request Header Fields Too Large
444 No Response (Nginx)
449 Retry With (Microsoft)
450 Blocked by Windows Parental Controls (Microsoft)
451 Unavailable For Legal Reasons
499 Client Closed Request (Nginx)

5xx Server Error
500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported
506 Variant Also Negotiates (Experimental)
507 Insufficient Storage (WebDAV)
508 Loop Detected (WebDAV)
509 Bandwidth Limit Exceeded (Apache)
510 Not Extended
511 Network Authentication Required
598 Network read timeout error
599 Network connect timeout error