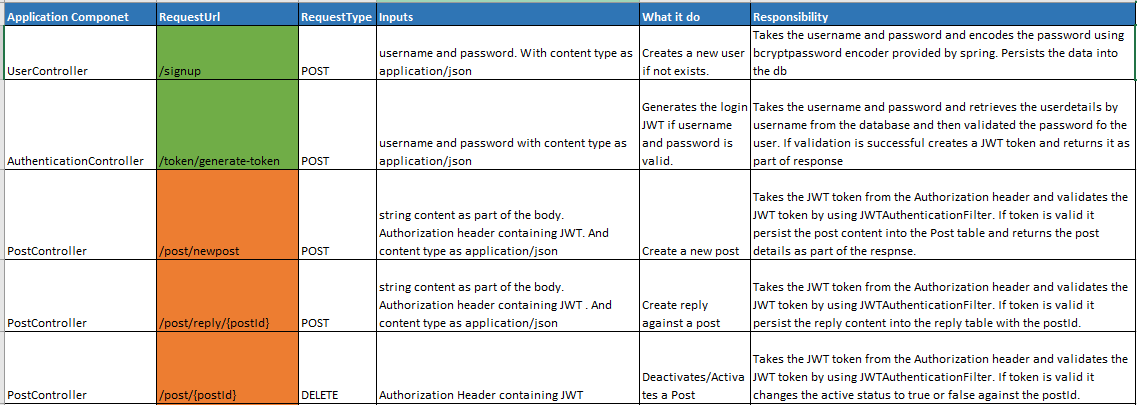
POST & REPLY MODULE:

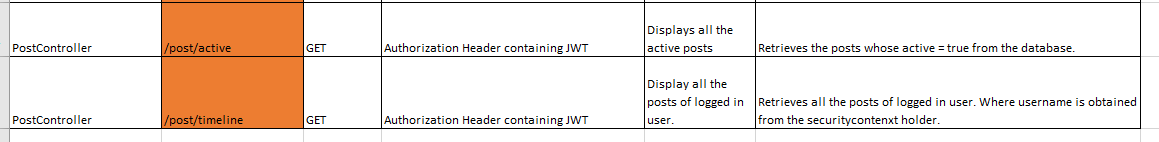
* Objectives: Post submitted by user. User can see his own post. Other logged in users can see posts and reply.
* Use cases:
  + User sign up
  + User login and generate token
  + User access the resource with JWT(JSON WEB TOKEN – BASE 64 Encoded)
  + User can add post / reply post / timeline(view own posts)/ activate or deactivate post / all posts (active posts)
* Technologies: JAVA 8, SPRING BOOT, MAVEN, SPRING DATA JPA, H2, JWT
* Design Overview: For all the use cases we have to expose the end points that exposes the data in JSON format.

Application Overview

1. Request Handling: Spring uses Dispatcher Servlet to accept the request from user and forwards those to dedicated Controller Methods. The controller methods forward the request to service layer. Service layer interacts with JPA. JPA has the responsibility to communicate with the database. The database configuration is kept inside src/main/resources/application.properties.
2. Authentication: To handle authentication, the app uses spring security. It uses JWT token to check the authenticity of a user. User has to get the JWT by providing the username and password. Now on the subsequent requests user has to sent the JWT as part of the Authorization request header.

Application Components:



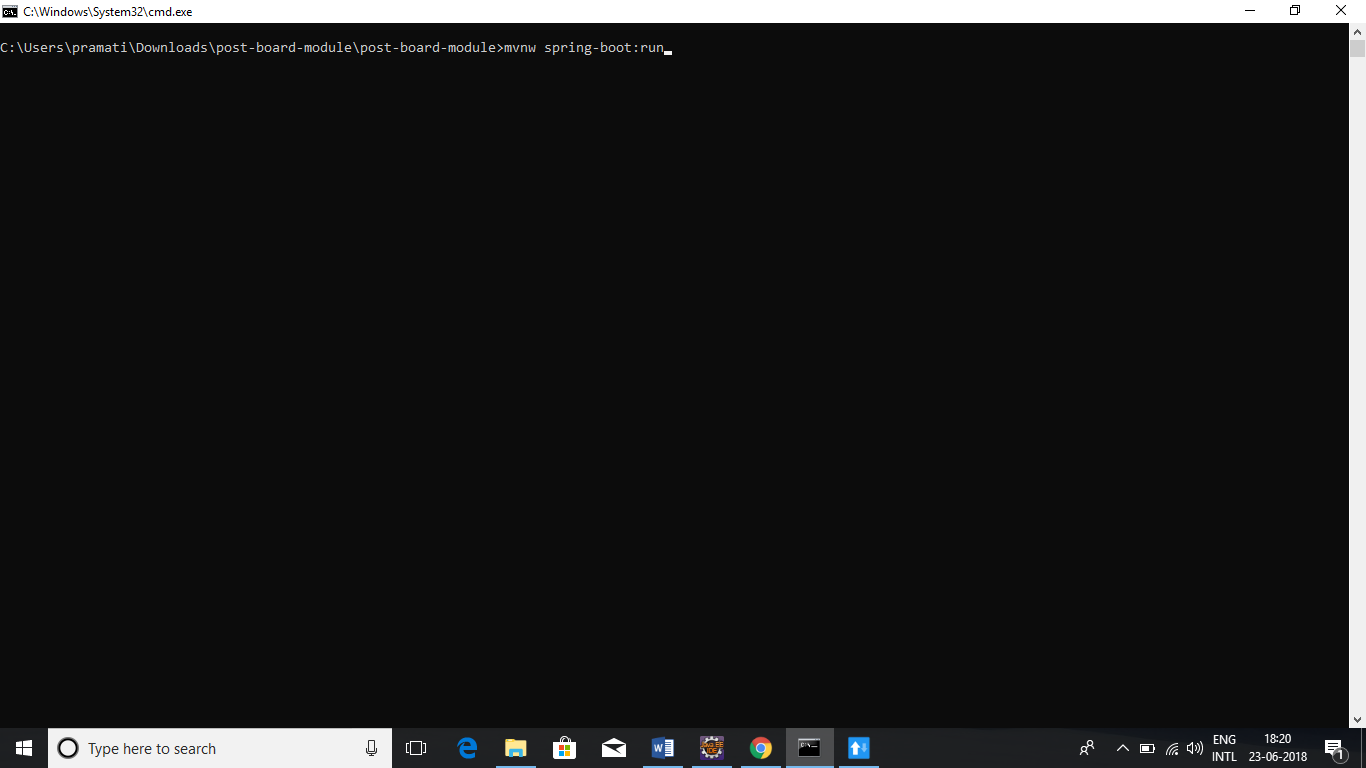


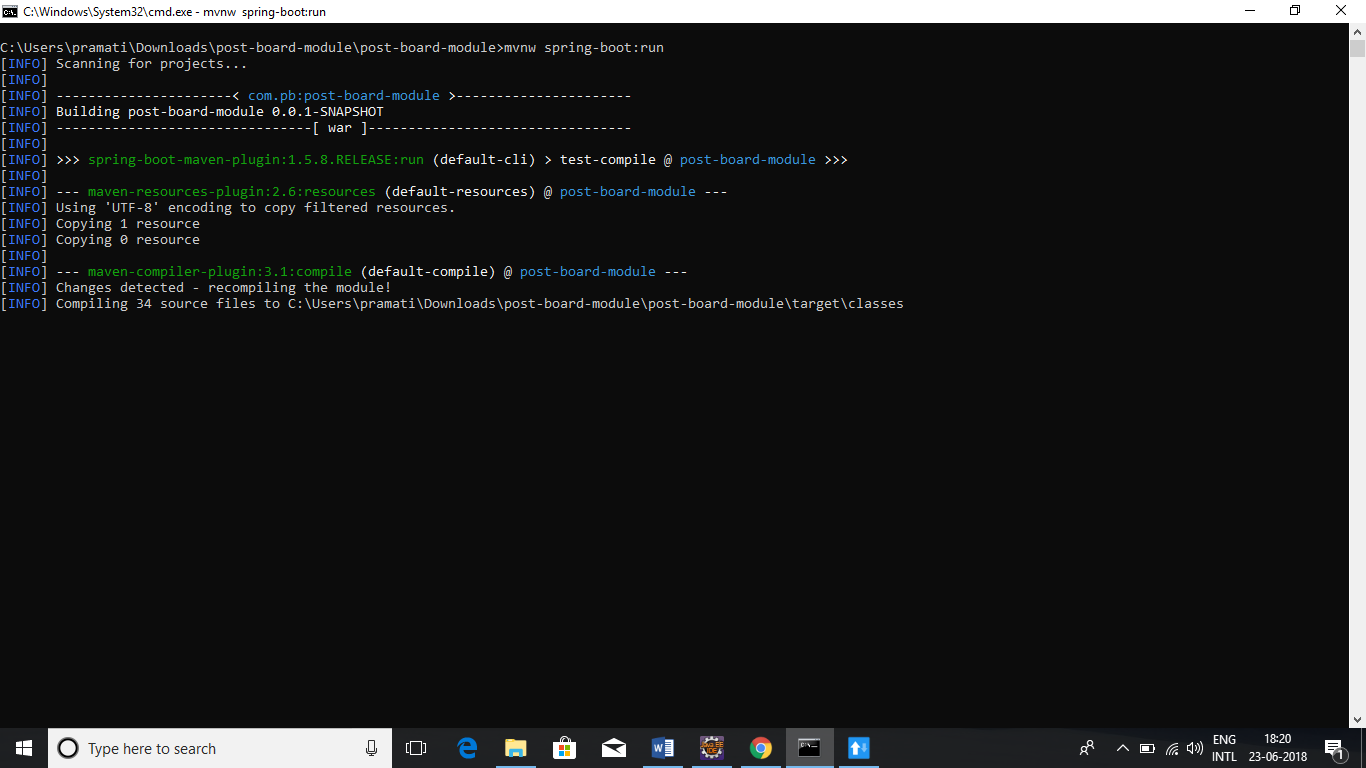
For details please refer the attached excel sheet.

Before starting the app make sure you advanced rest client extension installed as part of your chrome browser. If you want to install then visit here

<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo>

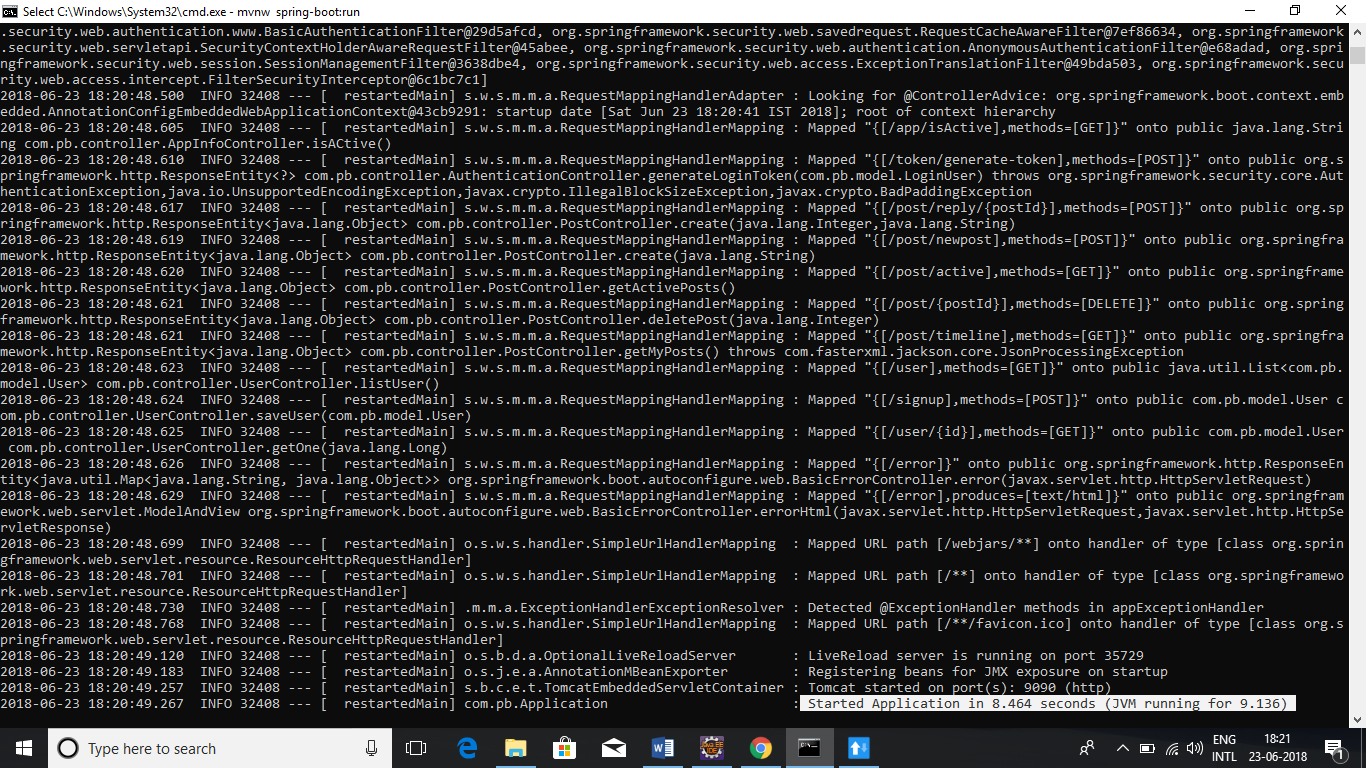
**Start the app**





After server is started you will see Server start up message at the end of above command line.

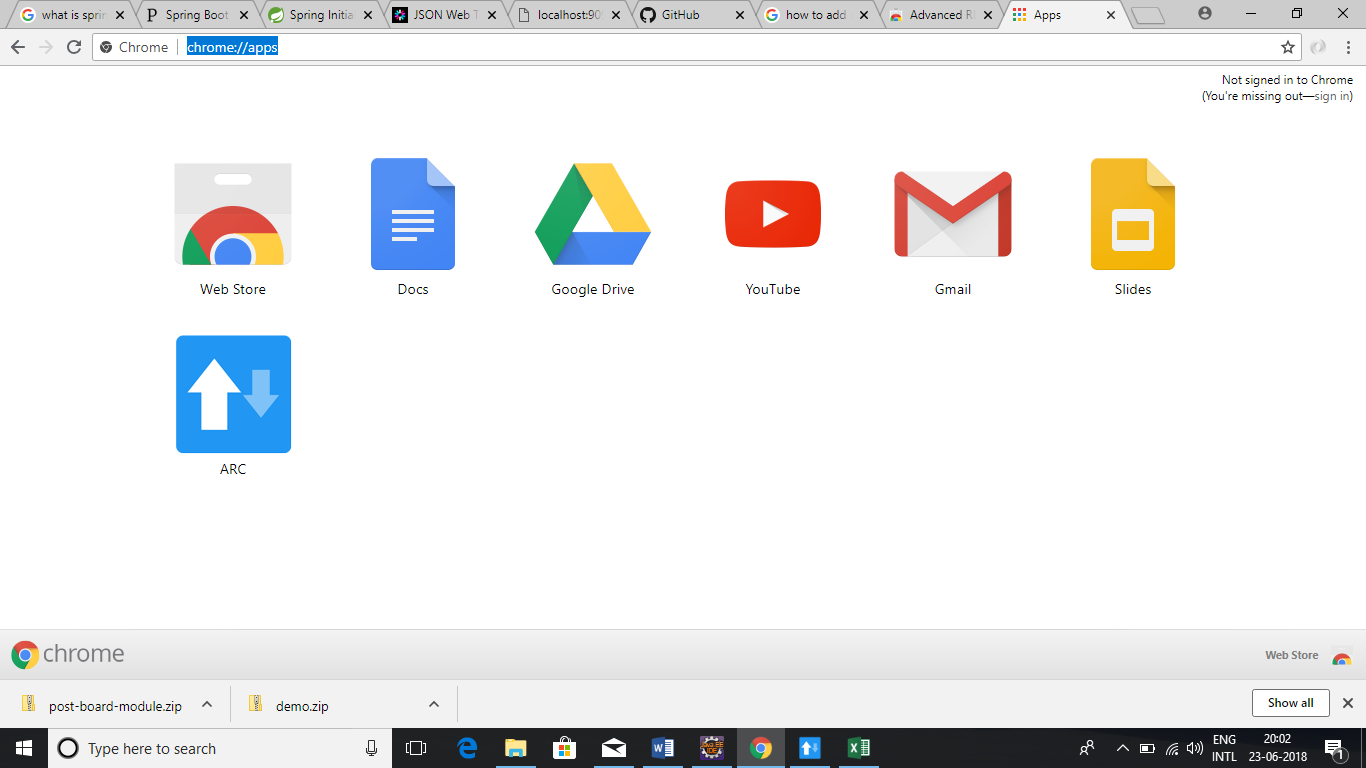
Refer below screenshot



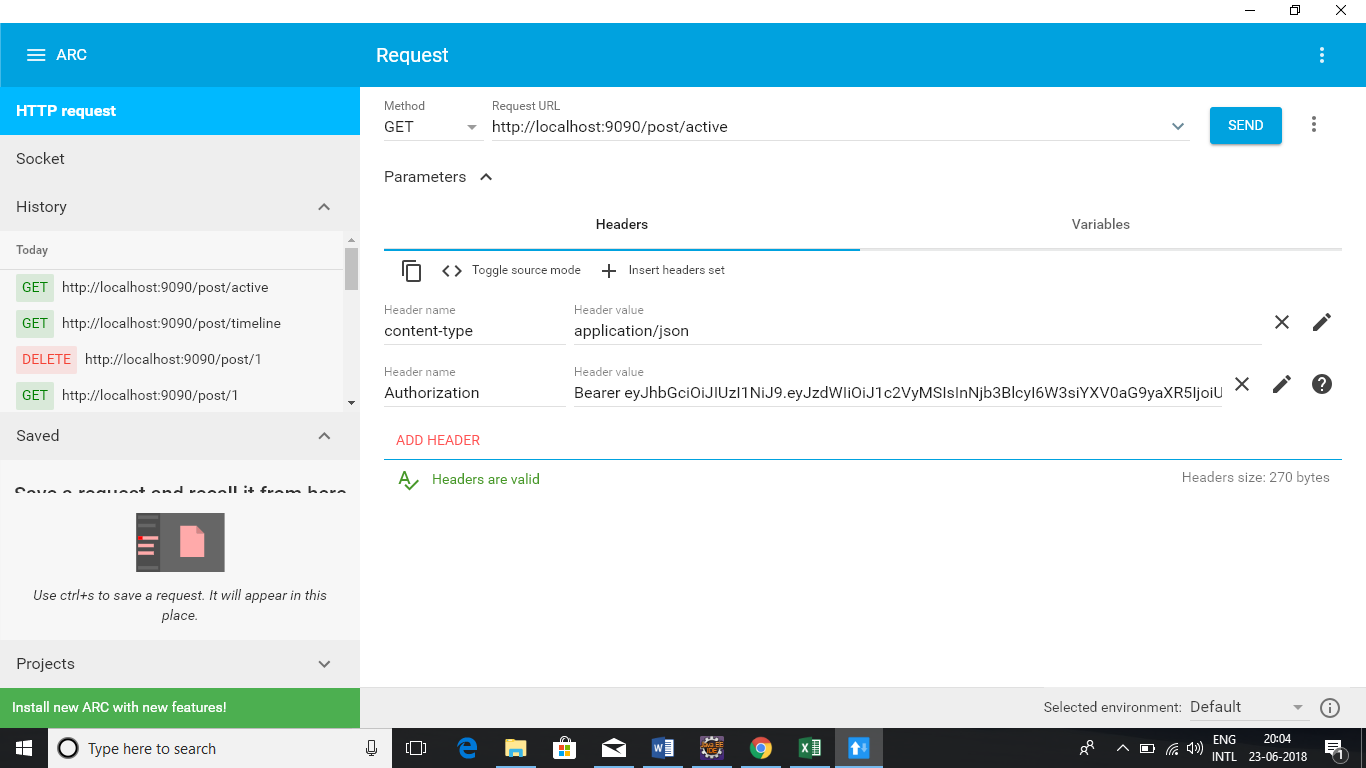
Above screenshot say that the app runs on 9090 port. The port has been configured at src/main/resources/application.properties

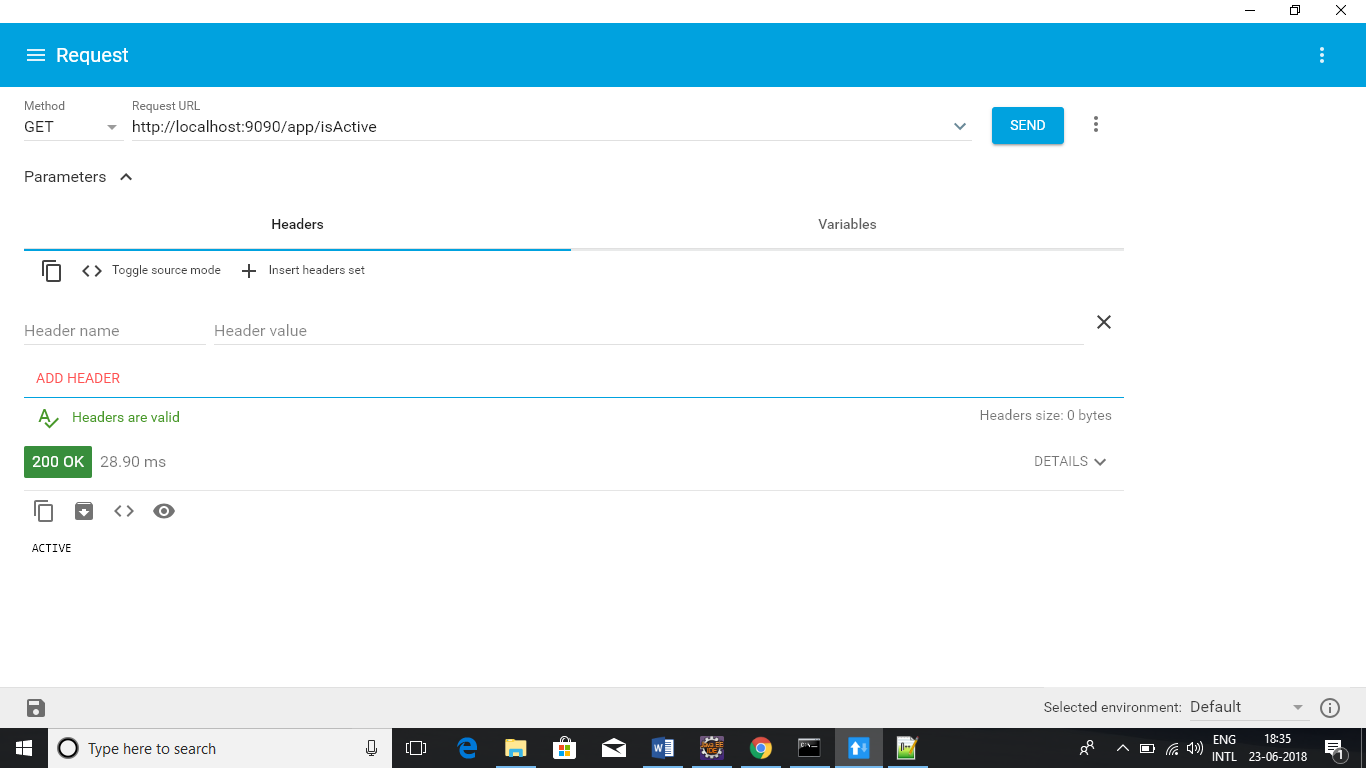
**Test the app is active or not:**

Open rest client type chrome://apps in the chrome tab as shown below.



Click on the ARC. You will find ARC in lunched state as shown below





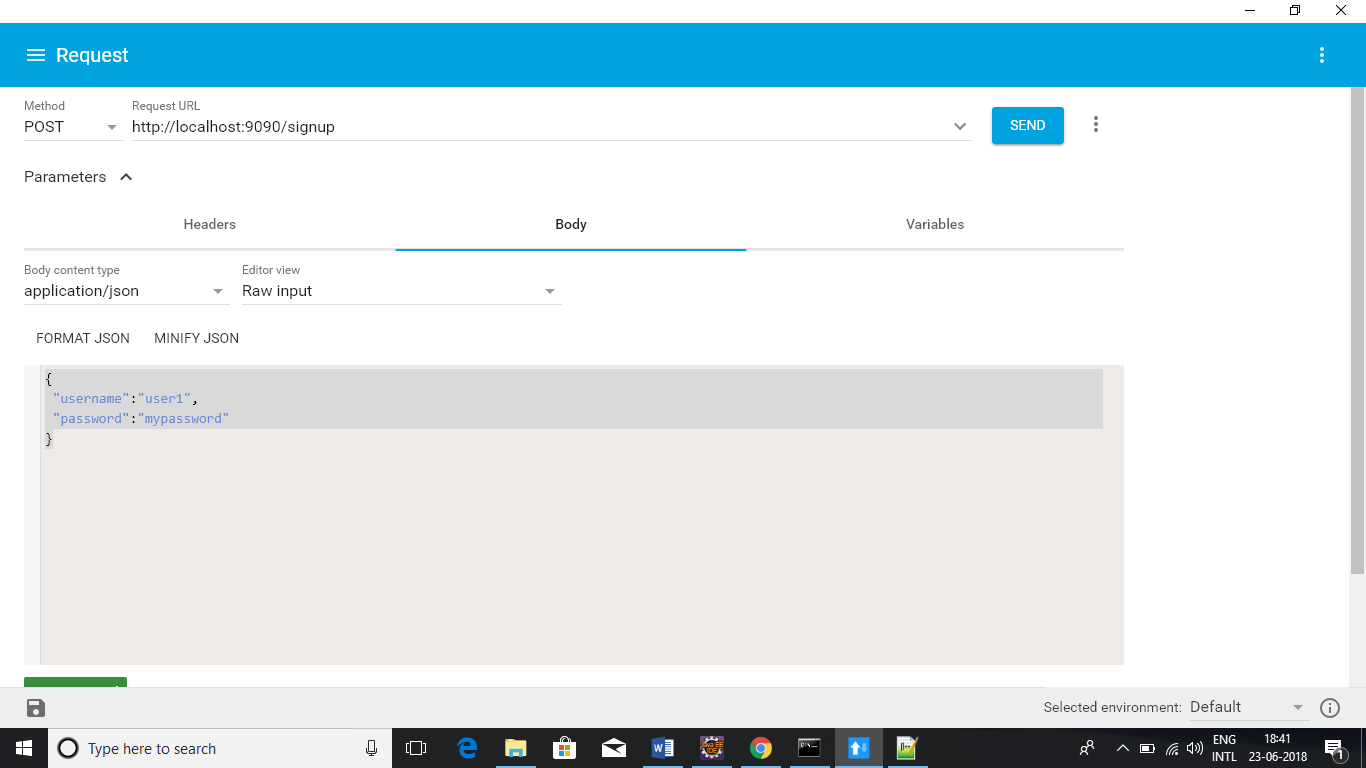
REQUEST

GET http://localhost:9090/app/isActive

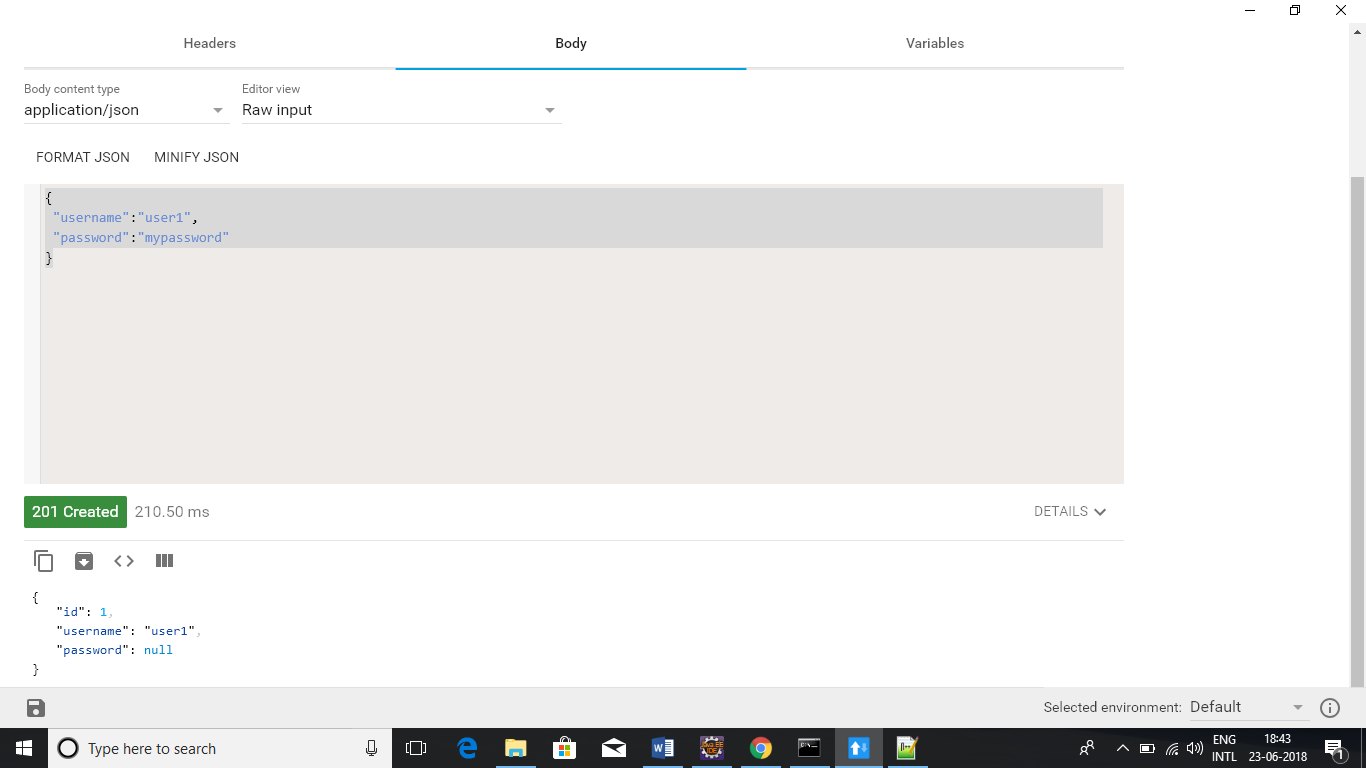
RESPONSE

ACTIVE

**User SignUp:**



User has been created. So we can see the id if user as part of below response.



POST http://localhost:9090/signup

Request Body:

{

"username":"user1",

"password":"mypassword"

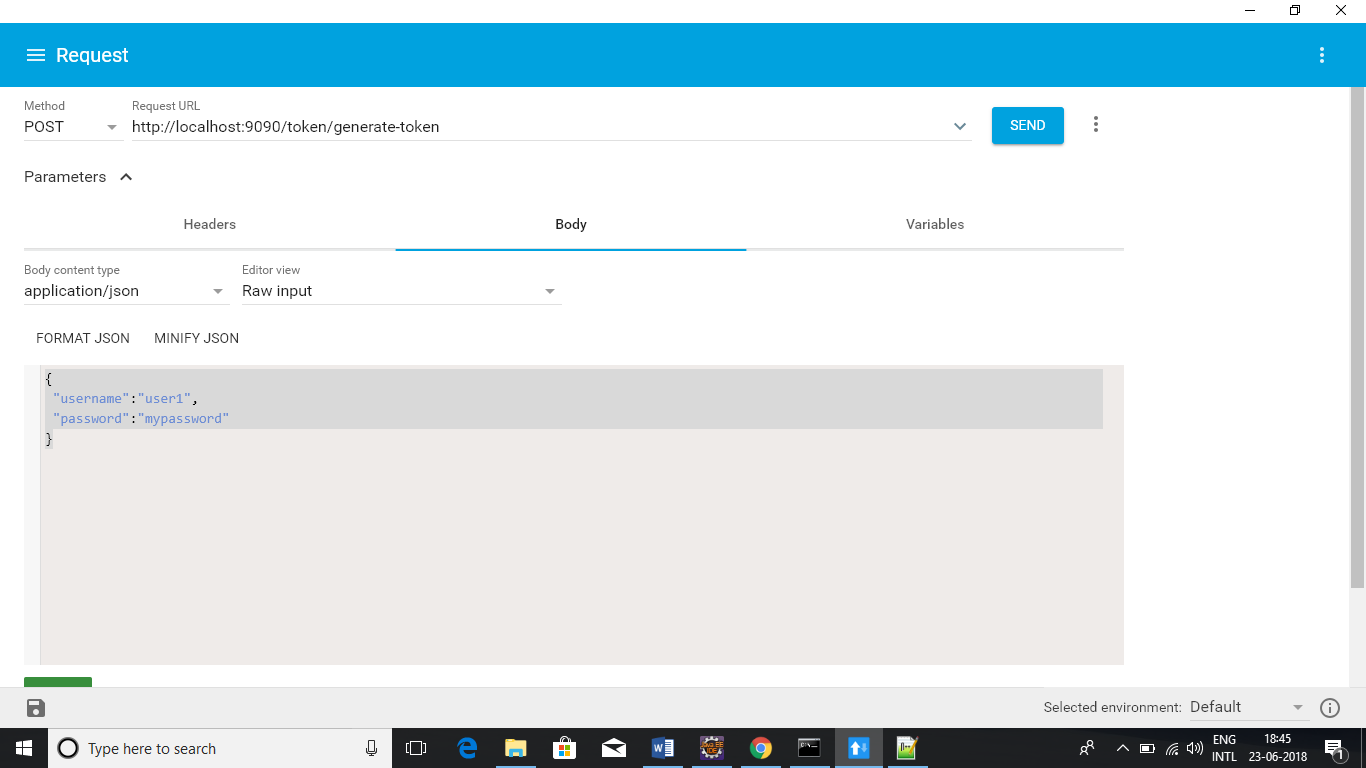
}

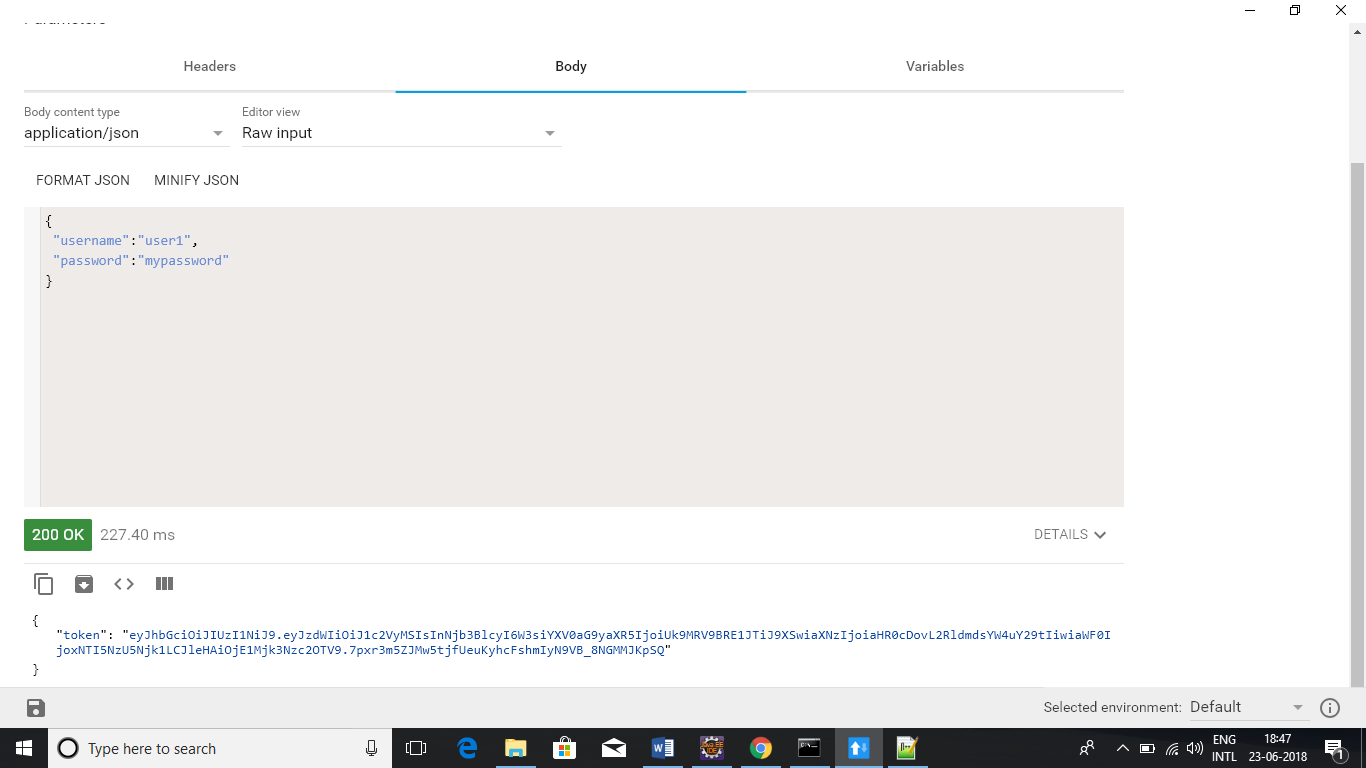
Set Body Content type as application/json

Response:

{"id":1,"username":"user1","password":null}

**Generate Login Token**





POST http://localhost:9090/token/generate-token

Request Body :

{

"username":"user1",

"password":"mypassword"

}

Set Body Content type as application/json

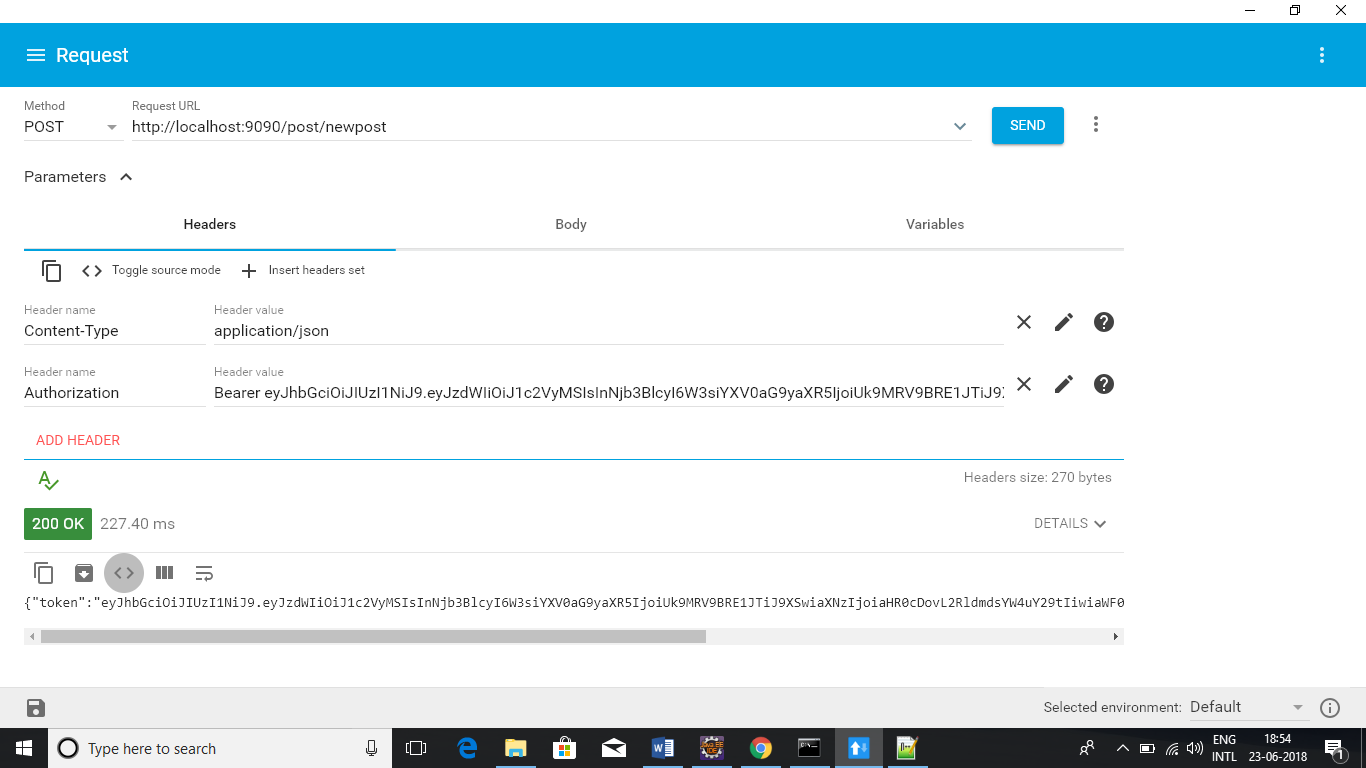
Response:

{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyMSIsInNjb3BlcyI6W3siYXV0aG9yaXR5IjoiUk9MRV9BRE1JTiJ9XSwiaXNzIjoiaHR0cDovL2RldmdsYW4uY29tIiwiaWF0IjoxNTI5NzU5Njk1LCJleHAiOjE1Mjk3Nzc2OTV9.7pxr3m5ZJMw5tjfUeuKyhcFshmIyN9VB\_8NGMMJKpSQ"}

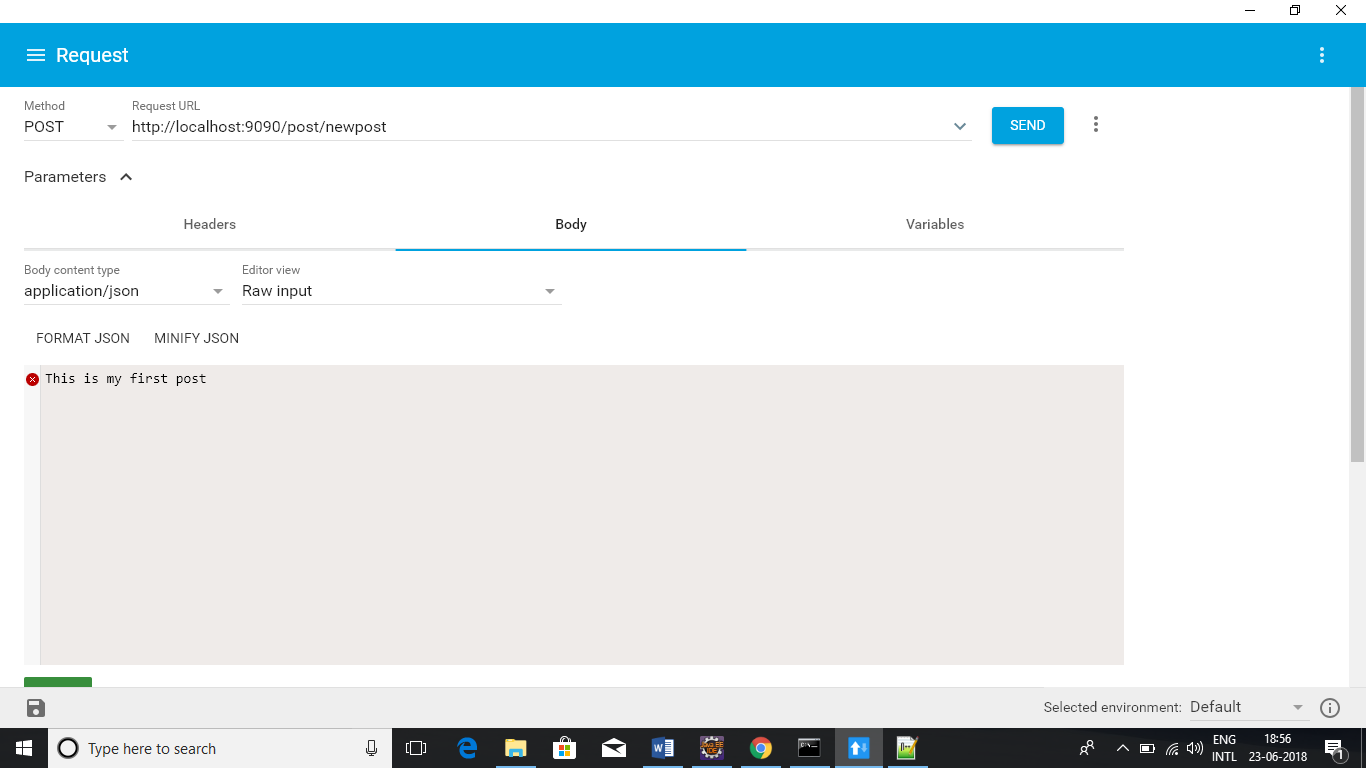
We obtained the token by providing the login credentials (username and password). Now we have to send the token as part of Authorization header while accessing the end points as explained below.

**Create a Post:**

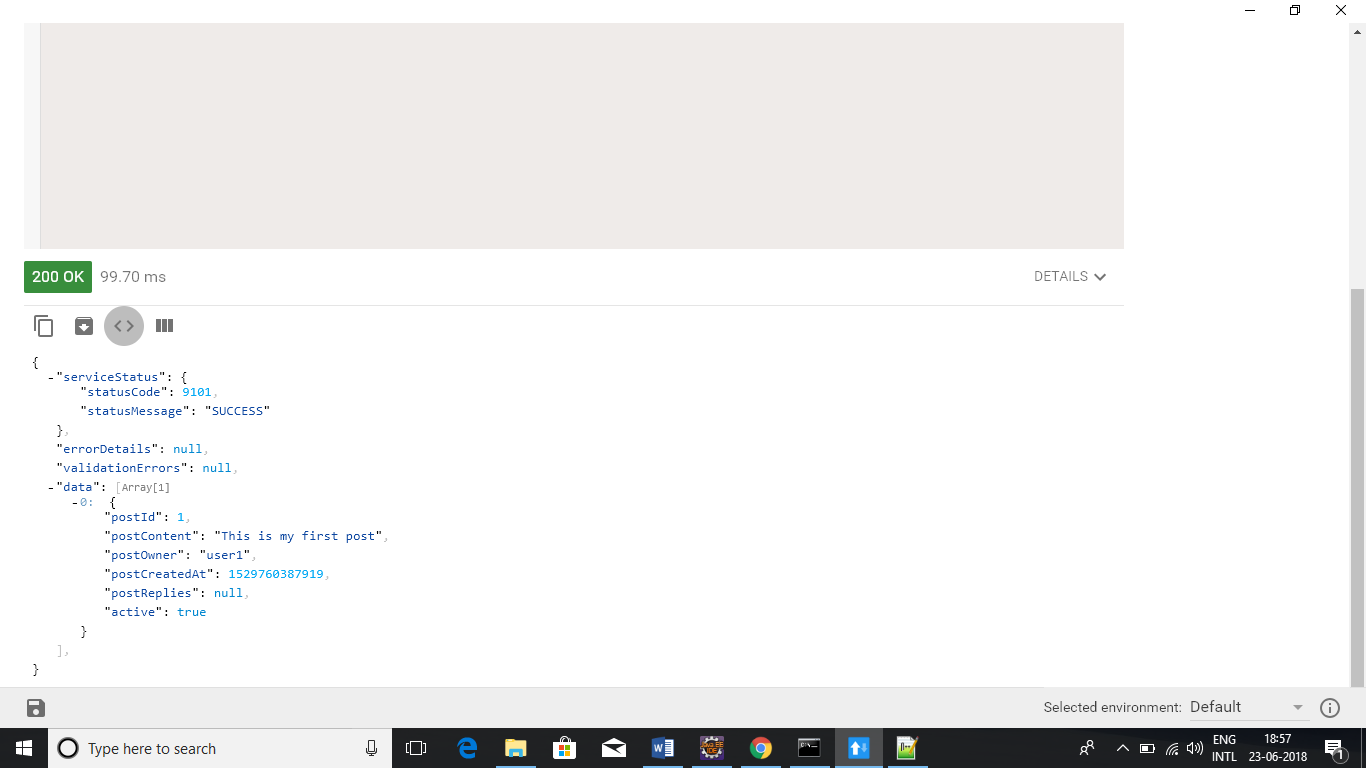
Lets know how to set Authorization header as part of every request. Setting of the Authorization header is same for every request in our application. Copy the above token and paste in the rest client as shown below with a prefix “Bearer ”.



Now go to body and write some content which we want to post. And click on send.



After post has been submitted we will get the below response.



POST http://localhost:9090/post/newpost

Set Authorization header = Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyMSIsInNjb3BlcyI6W3siYXV0aG9yaXR5IjoiUk9MRV9BRE1JTiJ9XSwiaXNzIjoiaHR0cDovL2RldmdsYW4uY29tIiwiaWF0IjoxNTI5NzU5Njk1LCJleHAiOjE1Mjk3Nzc2OTV9.7pxr3m5ZJMw5tjfUeuKyhcFshmIyN9VB\_8NGMMJKpSQ

Set Body Content type as application/json

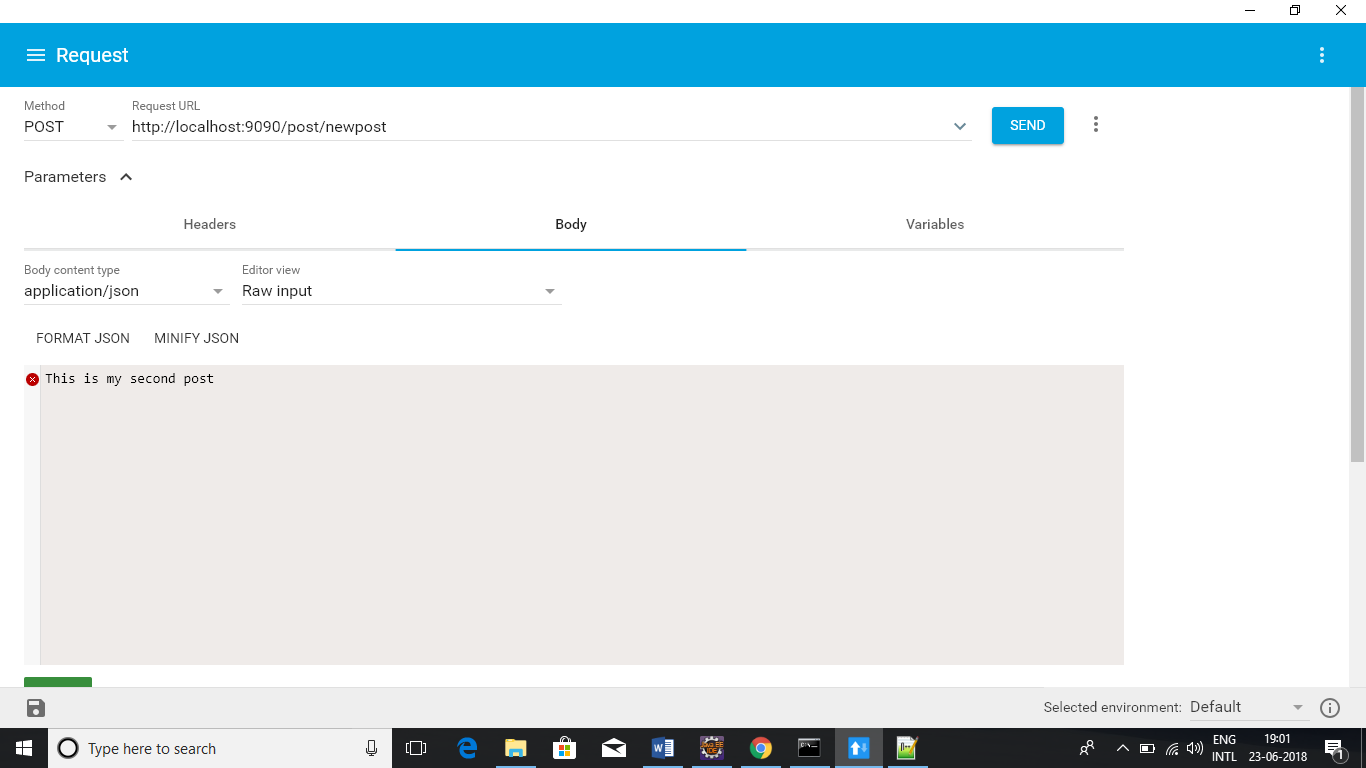
RequestBody:

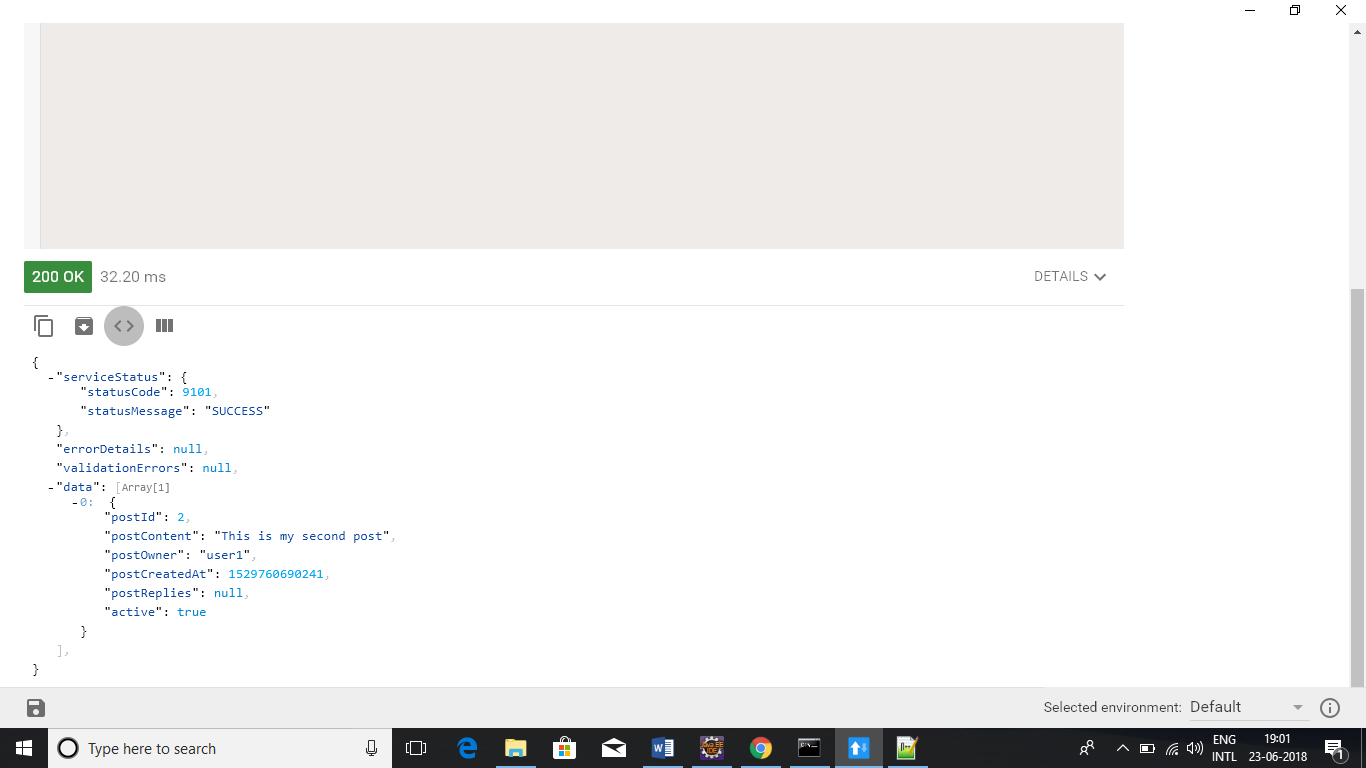
This is my first post

Response:

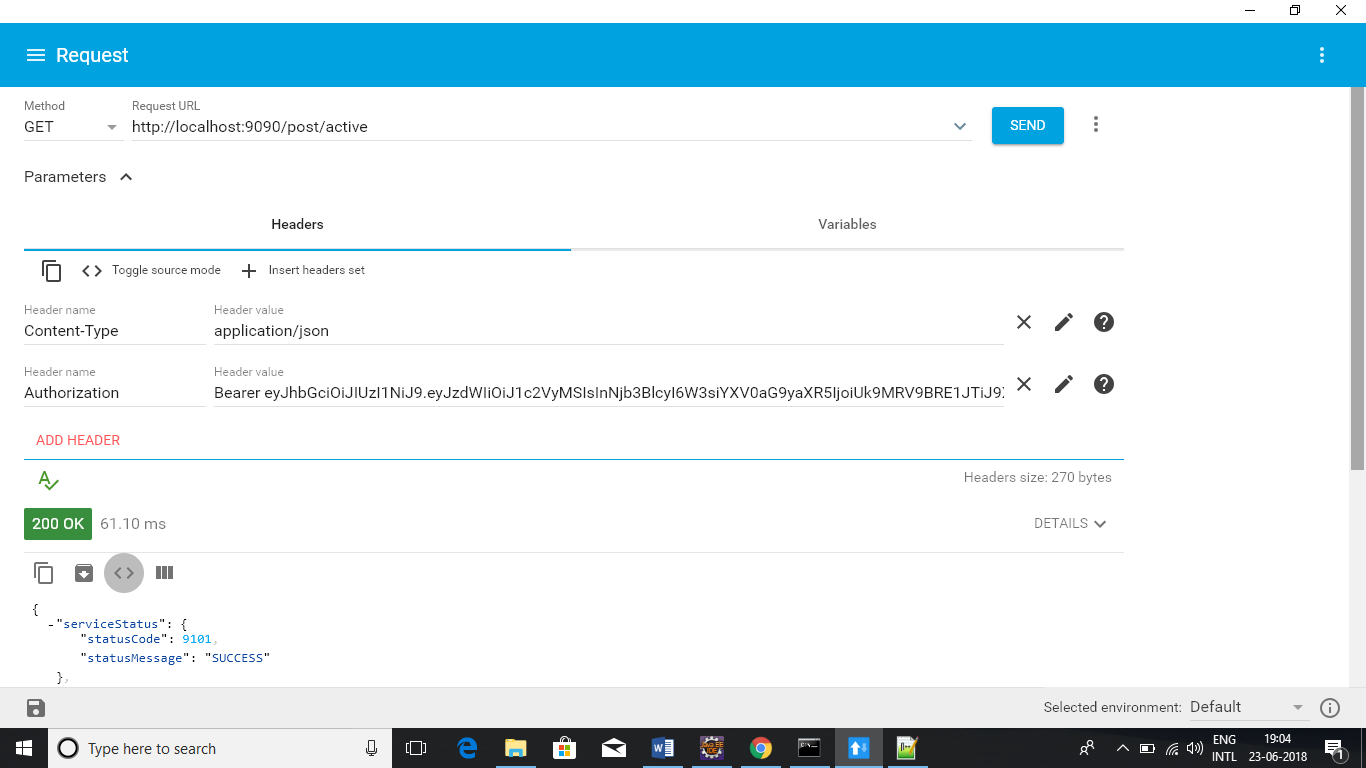
{"serviceStatus":{"statusCode":9101,"statusMessage":"SUCCESS"},"errorDetails":null,"validationErrors":null,"data":[{"postId":1,"postContent":"This is my first post","postOwner":"user1","postCreatedAt":1529760387919,"postReplies":null,"active":true}]}

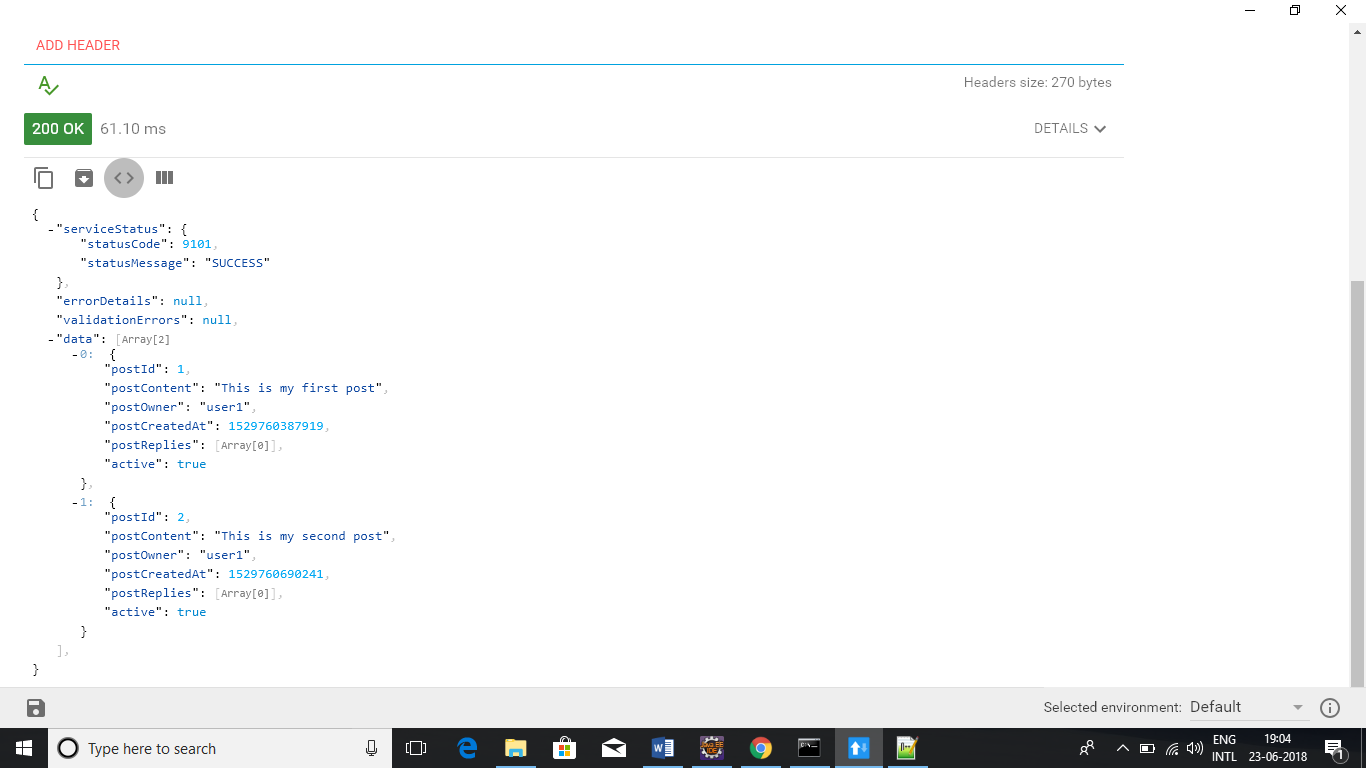
**Similarly create one more post as shown below**



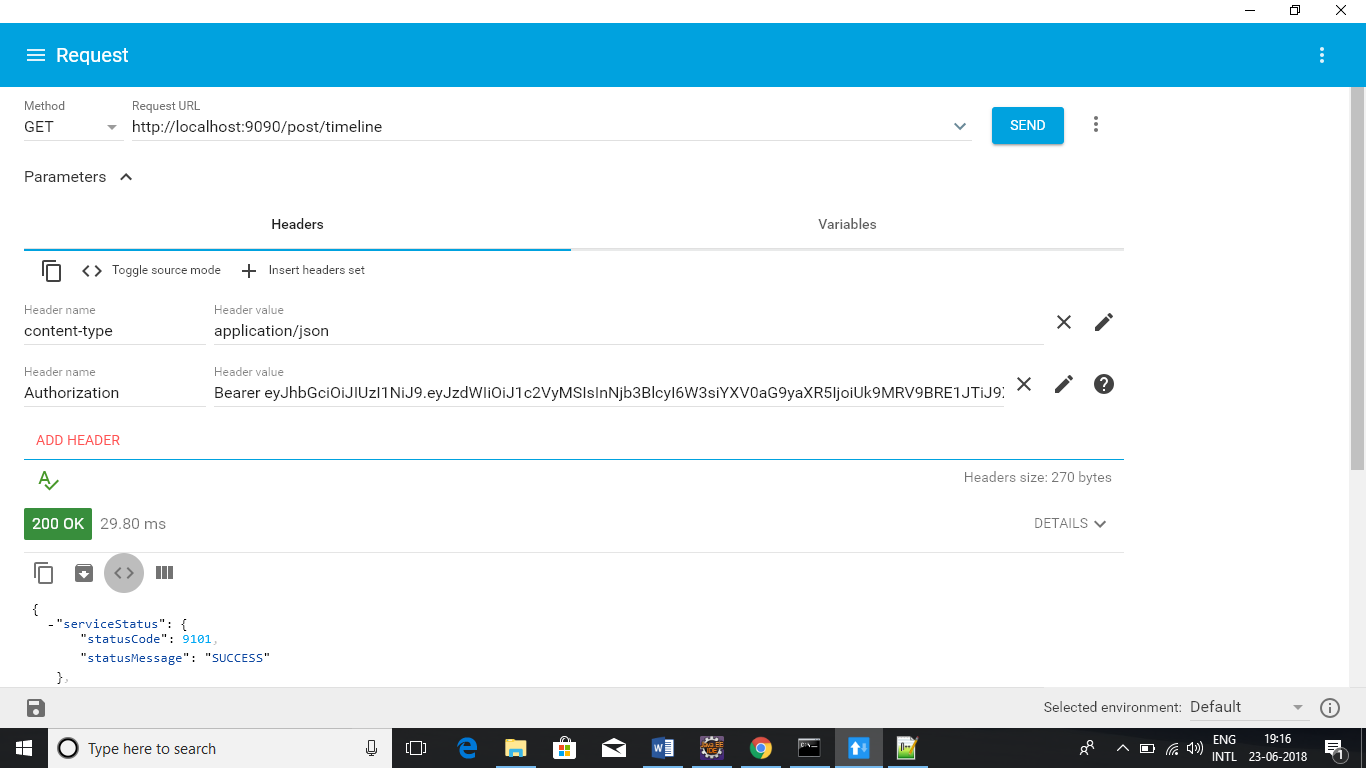


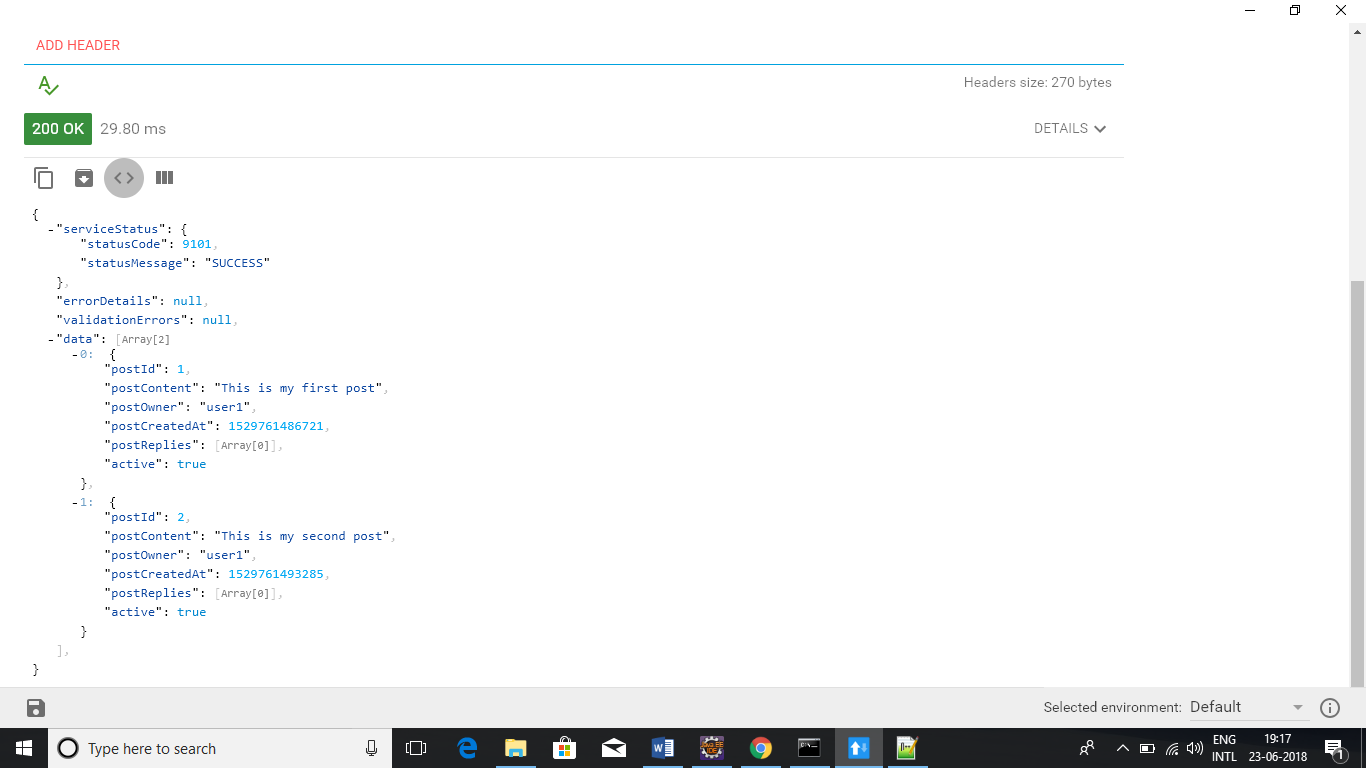
Now access active posts (or public posts)





**Access timeline (shows posts created by user)**



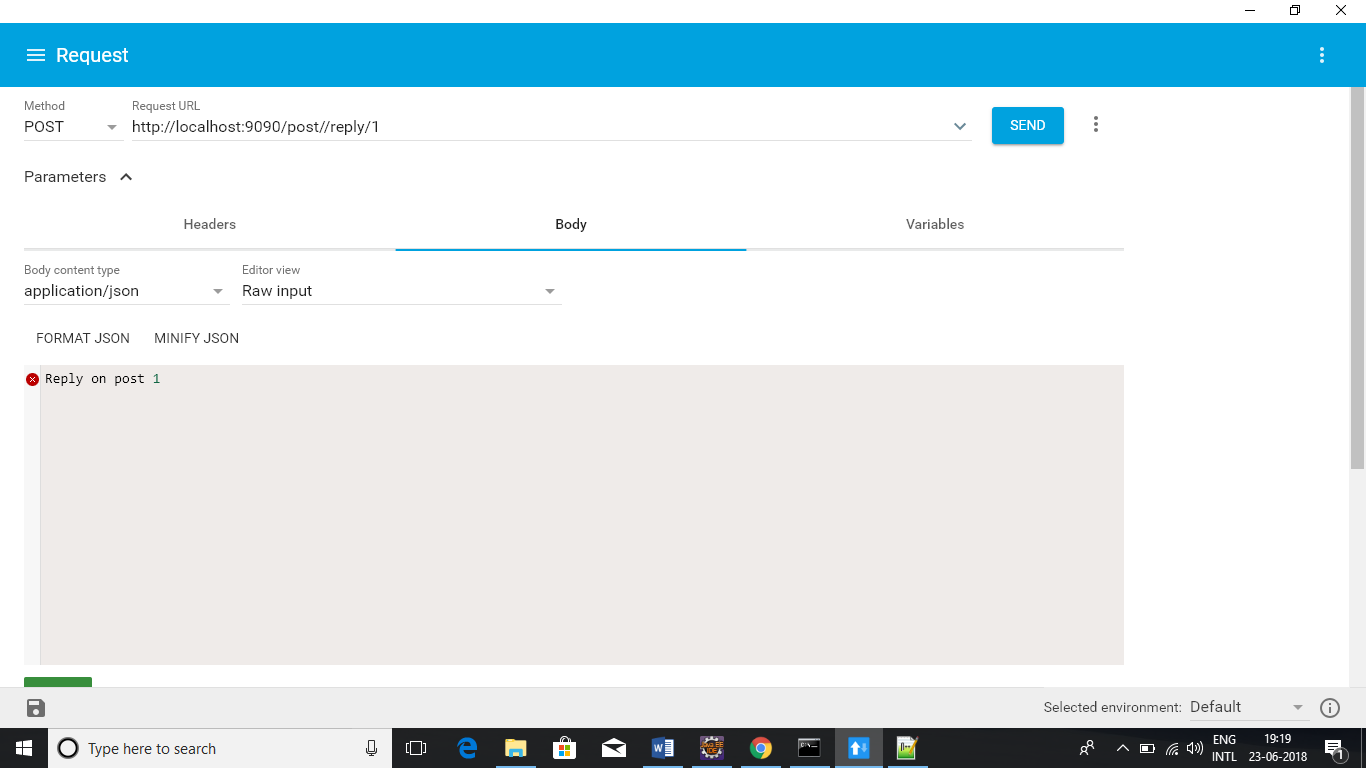


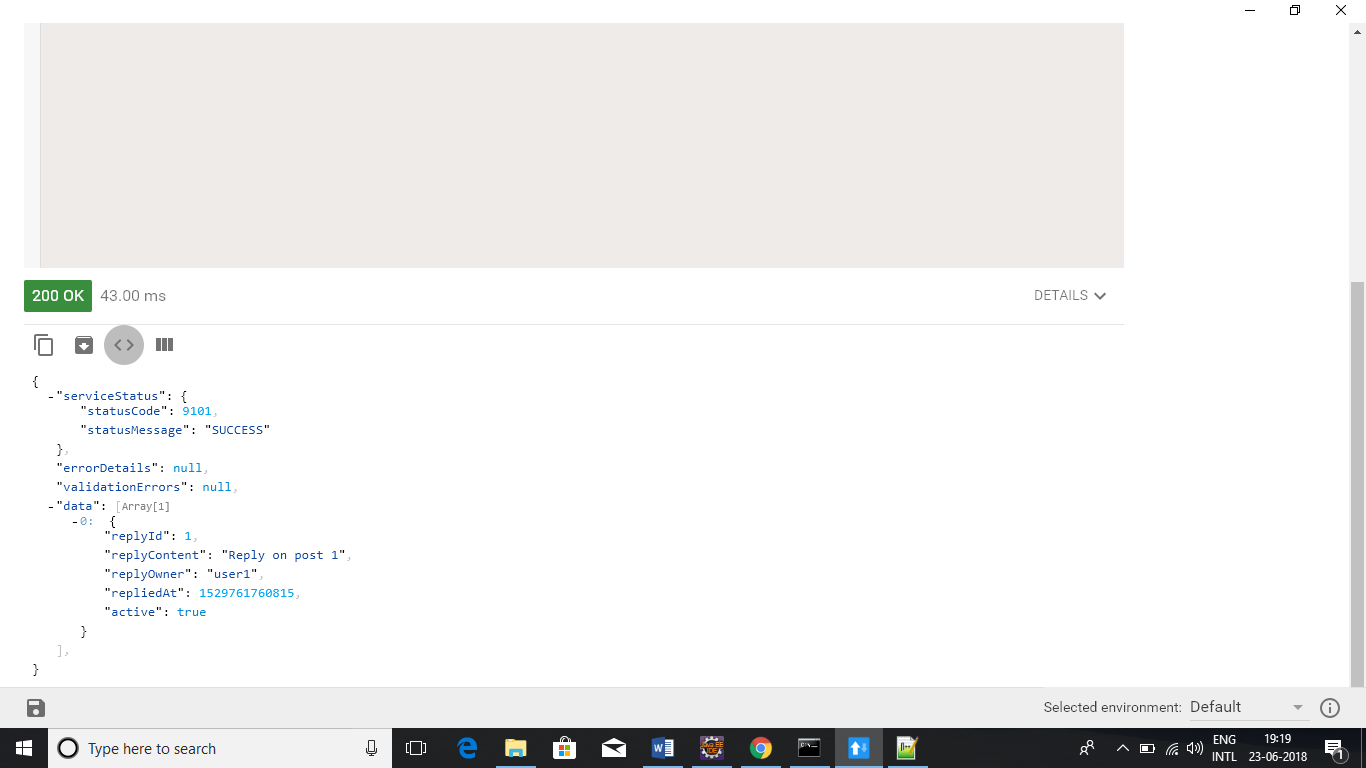
GET http://localhost:9090/post/timeline

{"serviceStatus":{"statusCode":9101,"statusMessage":"SUCCESS"},"errorDetails":null,"validationErrors":null,"data":[{"postId":1,"postContent":"This is my first post","postOwner":"user1","postCreatedAt":1529761486721,"postReplies":[],"active":true},{"postId":2,"postContent":"This is my second post","postOwner":"user1","postCreatedAt":1529761493285,"postReplies":[],"active":true}]}

**Reply to a post:**

So we have 2 posts with id as 1 and 2. Now we have to reply. So send a post request on reply endpoint with post content.





We can see that we have submitted a reply on a post.

Now we can see this reply as part of post with id=1.

Request

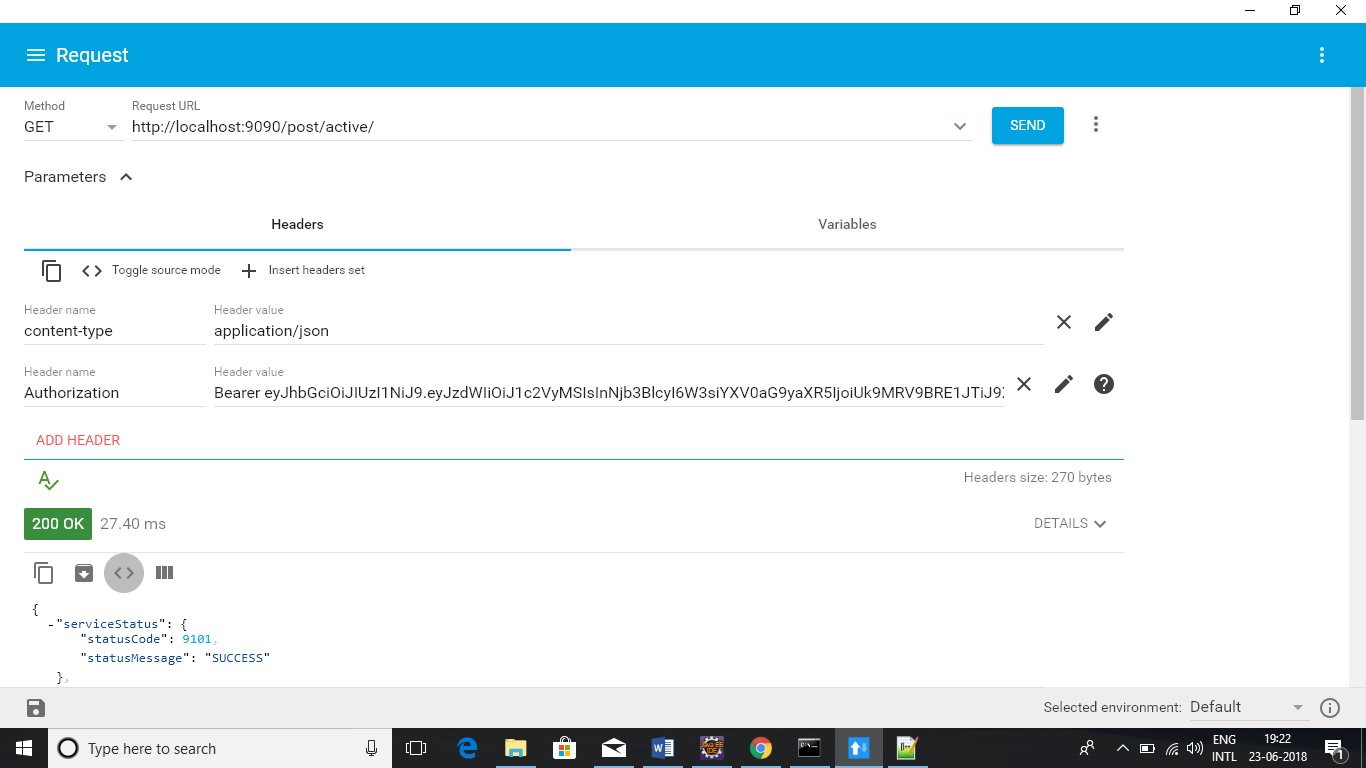
GET <http://localhost:9090/post//reply/1>

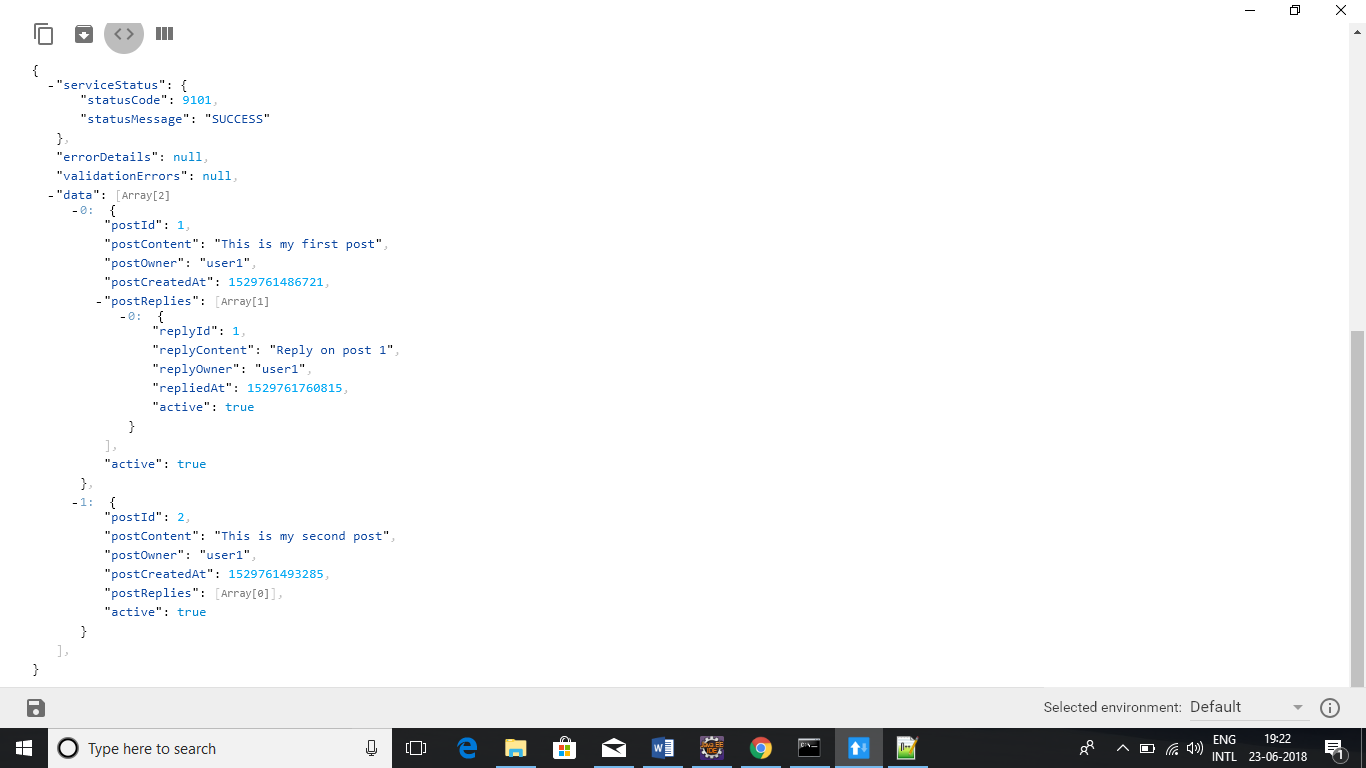
Your\_reply\_content

Response

{"serviceStatus":{"statusCode":9101,"statusMessage":"SUCCESS"},"errorDetails":null,"validationErrors":null,"data":[{"replyId":1,"replyContent":"Reply on post 1","replyOwner":"user1","repliedAt":1529761760815,"active":true}]}

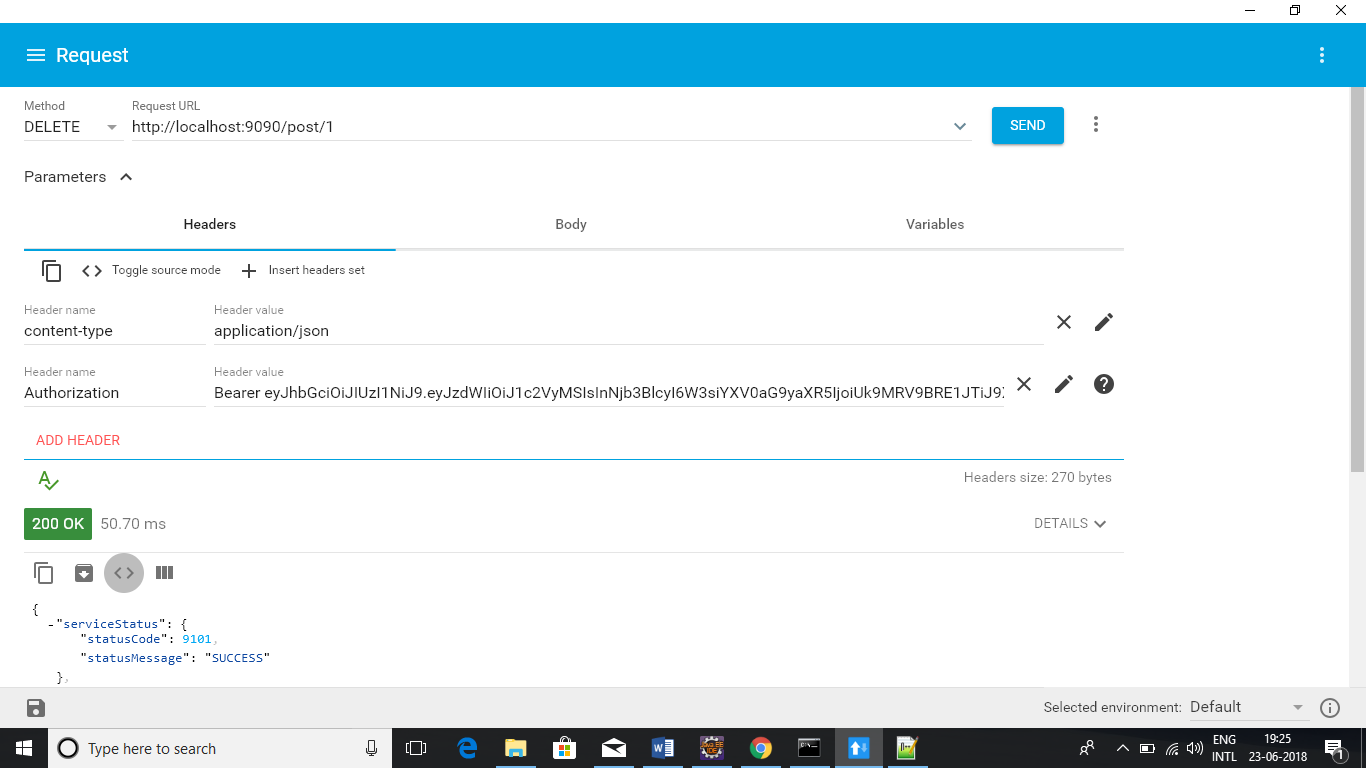
Accessing active post to see the reply:

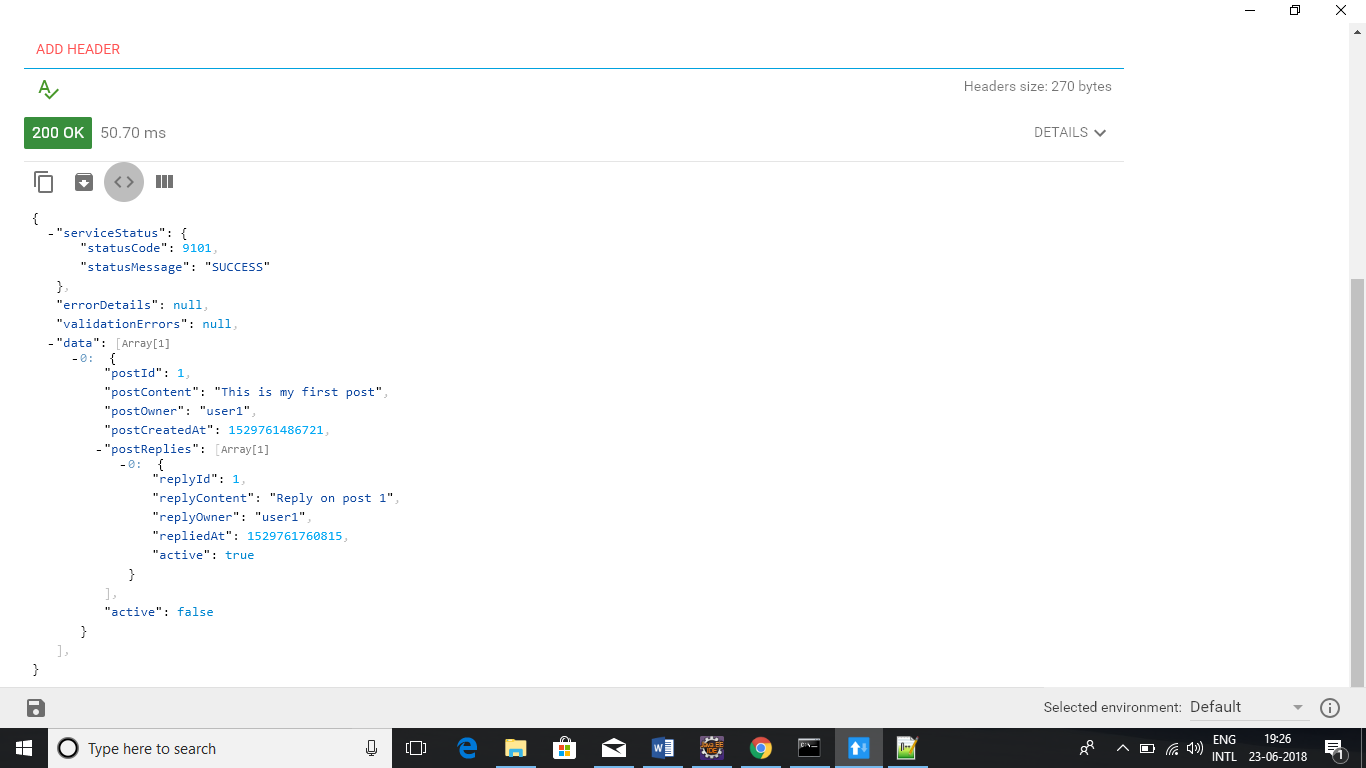




Above response shows the reply as part of post id =1.

**Deactivate or activate a post.**



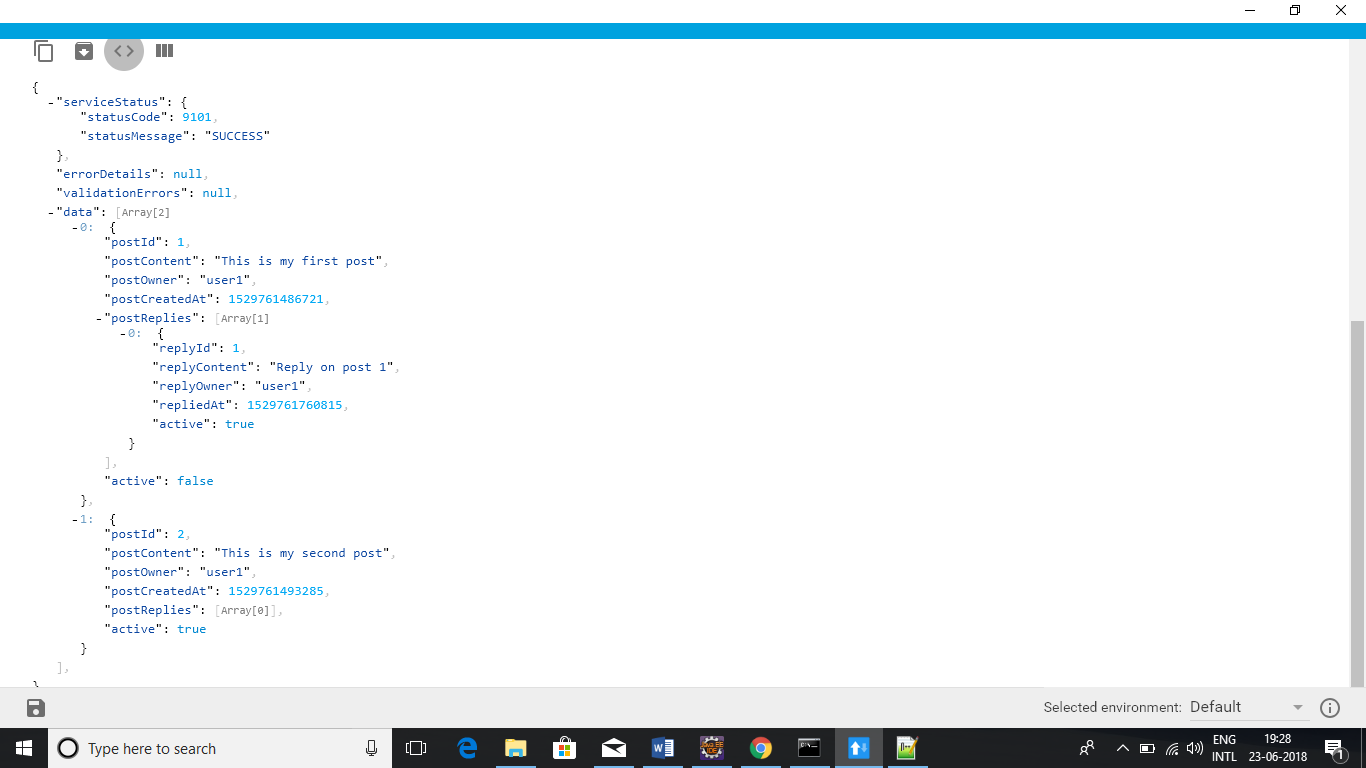


See the post is deactivated now. And it won’t be visible in /active but it will visible in /timeline.

DELETE <http://localhost:9090/post/1>

{"serviceStatus":{"statusCode":9101,"statusMessage":"SUCCESS"},"errorDetails":null,"validationErrors":null,"data":[{"postId":1,"postContent":"This is my first post","postOwner":"user1","postCreatedAt":1529761486721,"postReplies":[{"replyId":1,"replyContent":"Reply on post 1","replyOwner":"user1","repliedAt":1529761760815,"active":true}],"active":false}]}

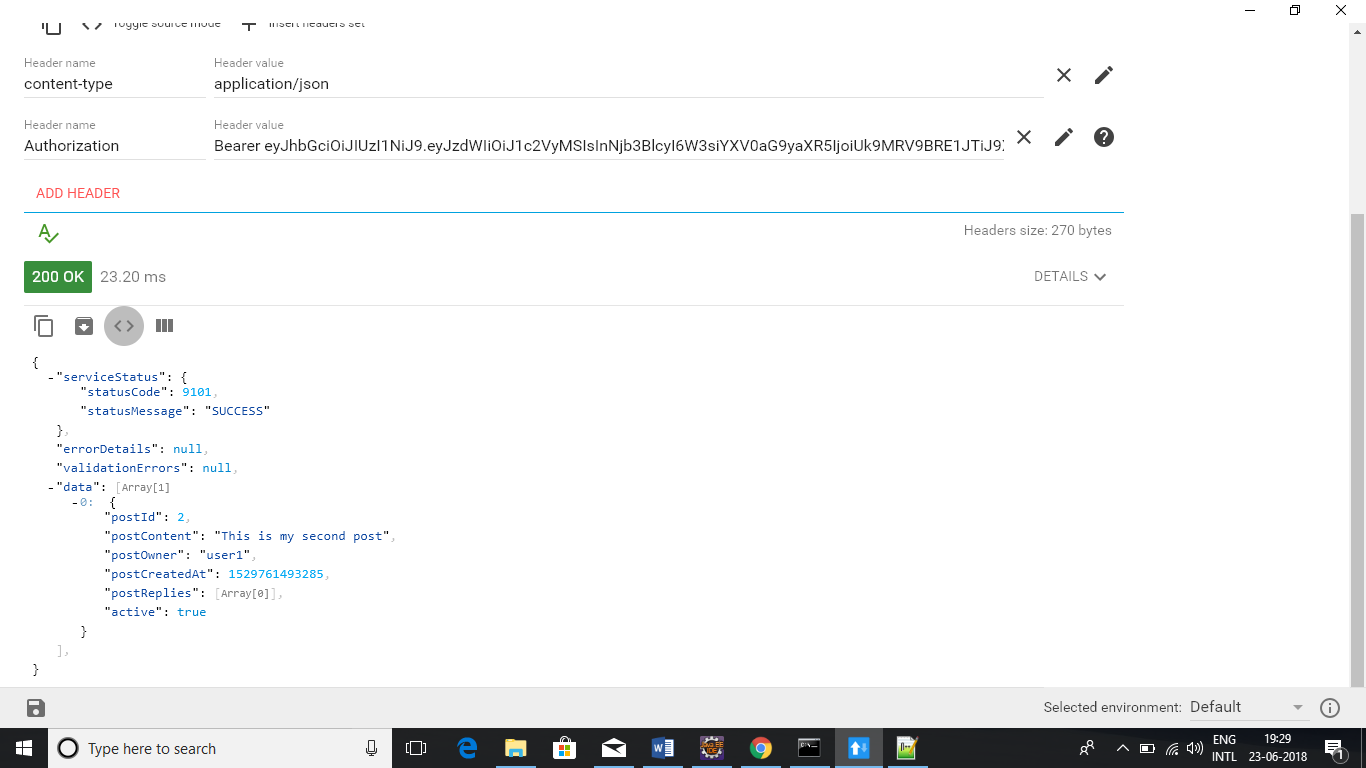
**Access timeline**



GET http://localhost:9090/post/timeline

Clearly deactivated post is visible in timeline with active=false.

**Access active posts**



GET <http://localhost:9090/post/timeline> Yes, the deactivated post is not visible as part of /active.

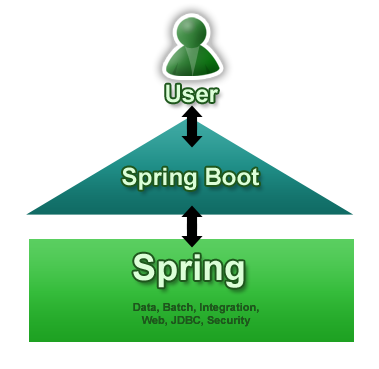
Similarly we can register multiple users. An each user can post or reply. Also they can see own posts as part of the timeline and active posts.

**TECHNOLOGY OVERVIEW**

**SPRING BOOT**

Spring Boot aims to make it easy to create Spring-powered, production-grade applications and services with minimum fuss. It takes an opinionated view of the Spring platform so that new and existing users can quickly get to the bits they need. You can use it to create stand-alone Java applications that can be started using ‘java -jar’ or more traditional WAR deployments. We also provide a command line tool that runs ‘spring scripts’.

The diagram below shows Spring Boot as a point of focus on the larger Spring ecosystem. It presents a small surface area for users to approach and extract value from the rest of Spring:



The primary goals of Spring Boot are:

* To provide a radically faster and widely accessible ‘getting started’ experience for all Spring development
* To be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults
* To provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration)

Spring Boot does *not* generate code and there is absolutely **no** requirement for XML configuration.

**Spring Scripts**

Spring Boot ships with a small command line application that can be used to run ‘spring scripts’. Spring scripts are written in [Groovy](http://groovy.codehaus.org/), which means that you have a familiar Java-like syntax, without so much boilerplate code. We are able to deduce a lot of information simply by looking at the way you have written your script. For example, here is a simple web application:

@Controller

class ThisWillActuallyRun {

@RequestMapping("/")

@ResponseBody

String home() {

return "Hello World!"

}

}

When you run this application using ‘spring run webapp.groovy’ a number things are happening:

* Your script is enhanced with common ‘import’ statements to save you typing them
* We recognize the @ResponseBody annotation and download appropriate Spring JARs
* We automatically create the Spring @Configuration that you would otherwise need to write
* We start up an embedded servlet container and handle incoming requests on port 8080

The command line tool recognizes a number of different types of Spring Applications, including Web, Batch and Integration. There are a number of [samples available in the GitHub repository](https://github.com/SpringSource/spring-boot/tree/master/spring-boot-cli/samples).

**Spring Boot with Java**

You don’t need use the command line tool or write Groovy code to get the benefits of Spring Boot. We also have first class Java support. For example, here is the same application written in Java:

import org.springframework.boot.\*;

import org.springframework.boot.autoconfigure.\*;

import org.springframework.stereotype.\*;

import org.springframework.web.bind.annotation.\*;

@Controller

@EnableAutoConfiguration

public class SampleController {

@RequestMapping("/")

@ResponseBody

String home() {

return "Hello World!";

}

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

SpringApplication.run(SampleController.class, args);

}

}

Other than import statements, the main difference between this example and the earlier Groovy script is the main() method that calls SpringApplication and the @EnableAutoConfiguration annotation.

Obviously with Java you also need a build system to compile and package your code. We provide a number of convenient ‘starter’ POMs that you can use with [Maven](https://maven.apache.org/), [Gradle](http://www.gradle.org/) or [Ant](https://ant.apache.org/)+[Ivy](https://ant.apache.org/ivy/)to quickly grab appropriate dependencies. For example, the application above would need just a single dependency to the spring-boot-starter-web module.

We also provide Maven and Gradle plugins that allow you to package a fully self contained 'fat jar' that can be started from the command line:

$ java -jar myproject.jar

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: v0.0.0.BUILD.SNAPSHOT

2013-07-31 00:08:16.117 INFO 56603 --- [ main] o.s.b.s.app.SampleApplication : Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)

2013-07-31 00:08:16.166 INFO 56603 --- [ main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@6e5a8246: startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy

**Production Ready**

Spring Boot also includes helpful features that you often need when you push an application into production. We can automatically provide web endpoints that you can use to monitor application health, provide basic metrics or use to analyze production issues (such as thread deadlocks). We also provide a new @ConfigurationProperties annotation that you can use to externalize your application configuration (complete with support for JSR-303 @Validannotations).

**SPRING DATA JPA:**

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Features

* Sophisticated support to build repositories based on Spring and JPA
* Support for [Querydsl](http://www.querydsl.com/) predicates and thus type-safe JPA queries
* Transparent auditing of domain class
* Pagination support, dynamic query execution, ability to integrate custom data access code
* Validation of @Query annotated queries at bootstrap time
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing @EnableJpaRepositories.

**Memory Database (H2):**

Typical databases involve a lot of setup.

For example, with Oracle or mySQL databases, you would need to

* Install the Database
* Setup a Schema
* Setup the tables
* Populate the data
* Connect the application to the database by setting up a data source and a lot of other code

Scenario 1 - Let’s consider a situation where you would want to do a quick POC. Using a traditional database involves a lot of overhead.

Scenario 2 - Consider your unit tests

* You don’t want them to fail when some data/schema in the database changes
* You would want to be able to run them in parallel - multiple developers might be running the tests in parallel.

In these kind of scenarios, an in memory database provides an ideal solution.

An in memory database is created when an application starts up and destroyed when the application is stopped.

Advantages

* Zero project setup or infrastructure
* Zero Configuration
* Zero Maintainance
* Easy to use for Learning, POCs and Unit Tests
* Spring Boot provides Simple Configuration to switch between a real database and an in memory database like H2

## H2

H2 is one of the popular in memory databases. Spring Boot has very good integration for H2.

From https://en.wikipedia.org/wiki/H2\_(DBMS)

*H2 is a relational database management system written in Java. It can be embedded in Java applications or run in the client-server mode.*

H2 supports a sub set of the SQL standard.

H2 also provides a web console to maintain the database.

## Spring Boot and H2

You need very little configuration to connect Spring Boot application with H2.

In most situations, just adding the H2 runtime jar into dependencies should be sufficient.

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

Create a simple Student Entity with a primary key id.

@Entity

public class Student {

@Id

@GeneratedValue

private Long id;

private String name;

private String passportNumber;

## H2 - A Few Tips

* An in-memory database is live only during the time of execution of the application. It is an efficient way to learn a framework.
* This is not how you want your real world applications to behave.
* We explain how to connect to a database of your choice in the answer to the question “How do we connect to a external database?”.

## Frequently asked questions about Spring Boot, JPA, Hibernate and H2

### Q : How does H2 and Spring Boot combination work?

First and most important thing - Spring Boot is intelligent.

If you are talking to an in memory db, by default, it looks at the entities and creates the database and the tables.

However, if you connect to a mysql database, Spring Boot knows that its a permanent database. By default, it expects you to set up the database, set up the tables and it uses the connection that you established.

### Q : How did the Spring Boot Application connect to the database H2?

Its down to Spring Boot Auto Configuration!

First thing you would need to understand is Spring Boot Auto Configuration.

Here’s a good read

* http://www.springboottutorial.com/spring-boot-auto-configuration

As far as H2 is concerned, as soon as Spring Boot sees H2 in the class path, it auto configures a data source similar to what you see below:

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

It knows that you are using an inmemory database H2 and it uses the default url if you don’t provide one.

### Q : Where is the database connection info specified? How does it know to automatically connect to H2?

Thats Spring Boot Autoconfiguration magic.

From https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html

*Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring Boot will auto-configure an in-memory database.*

Recommended Reading

* http://www.springboottutorial.com/spring-boot-auto-configuration

### Q : What happens if H2 is not in the classpath?

You get this error

Cannot determine embedded database driver class for database type NONE

Add H2 to the pom.xml and Restart your server

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

### Q : Why is the data lost between restart?

H2 is an in memory database. Its not a persisted database.

H2 is a great tool for learning because you need zero setup.

## Running H2 as a persisted database with Spring Boot

While we dont recommend this , it interesting to note that H2 has a persisted database mode

* With this configuration, the data is not lost even after spring boot restart and computer restart.

You would find H2 being very rarely used in this way. If you are really interested in a persistent database, we recommend exploring MySQL, Oracle or some other relational database.

application.properties

spring.datasource.name=yourdbname

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.initialize=false

spring.datasource.url=jdbc:h2:file:~/yourdbname;DB\_CLOSE\_ON\_EXIT=FALSE;IFEXISTS=TRUE;DB\_CLOSE\_DELAY=-1;

spring.jpa.hibernate.ddl-auto = update

## Using H2 for unit tests

The standard properties file that Spring Boot picks up automatically when running an application is called application.properties and resides in the src/main/resources folder.

If we want to use different properties for tests, then we can override the properties file in the main folder by placing another file with the same name in src/test/resources.

The application.properties file in src/test/resources folder should contain the standard key-value pairs necessary for configuring a in memory connection.

First add the dependencies for your database driver (mysql in the example below) and make the dependency for h2 test scoped.

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>test</scope>

</dependency>

Use the mysql database for your real code

src\main\resources\application.properties

spring.jpa.hibernate.ddl-auto=none

spring.datasource.url=jdbc:mysql://localhost:3306/person\_example

spring.datasource.username=personuser

spring.datasource.password=YOUR\_PASSWORD

Use in memory database for your unit tests

src\test\resources\application.properties

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:testdb;DB\_CLOSE\_DELAY=-1

spring.datasource.username=sa

spring.datasource.password=sa

**JWT**

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

## When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

* **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
* **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

## What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz