**Understand Spring Security Architecture and implement Spring Boot Security**

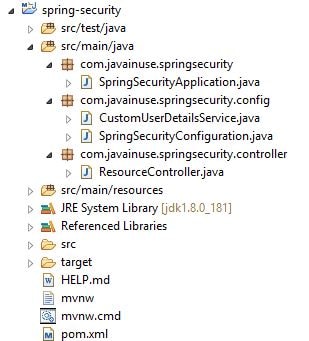
[Close X](https://www.javainuse.com/)

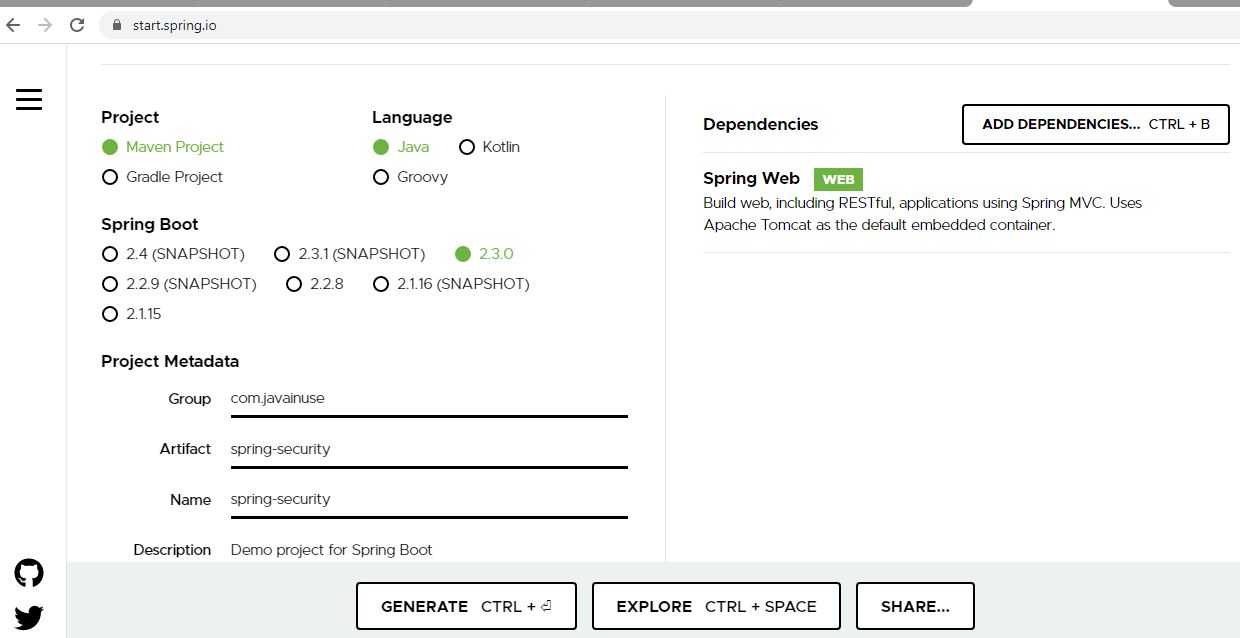
In this tutorial we will be looking at how Spring Security works and its architecture. We will be creating a Spring Boot Project to expose two REST API's

* /helloadmin
* /hellouser

We will then be implementing Spring Security such that a client having **Admin role will be able to access both /helloadmin and /hellouser API**. While a **client having User role will be able to access only /hellouser API**

Spring Boot Project to expose REST API's

Our Maven Project at the end of this tutorial will be as follows-  


Go to [Spring Initializr website and create a new Spring Boot Project.](https://start.spring.io/) We will only include the **Web** dependency now.  
  
The pom.xml will be as follows-

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.2.0.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>com.javainuse</groupId>

<artifactId>spring-security</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-security</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Next create the Controller class to expose the REST API's -

package com.javainuse.springsecurity.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ResourceController {

@RequestMapping({"/hellouser"})

public String helloUser(){

return "Hello User";

}

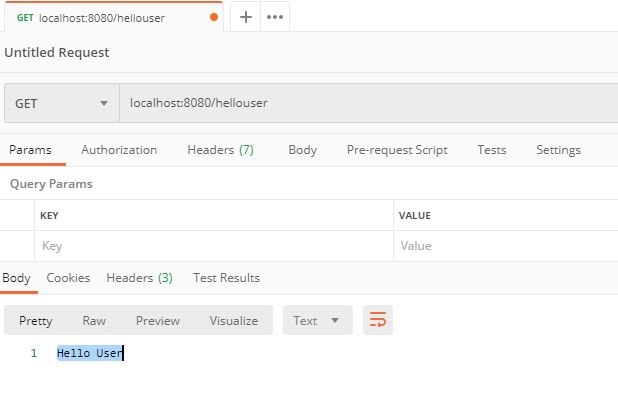
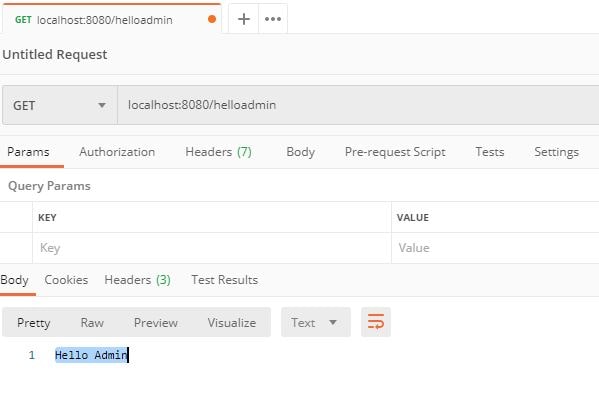
@RequestMapping({"/helloadmin"})

public String helloAdmin(){

return "Hello Admin";

}

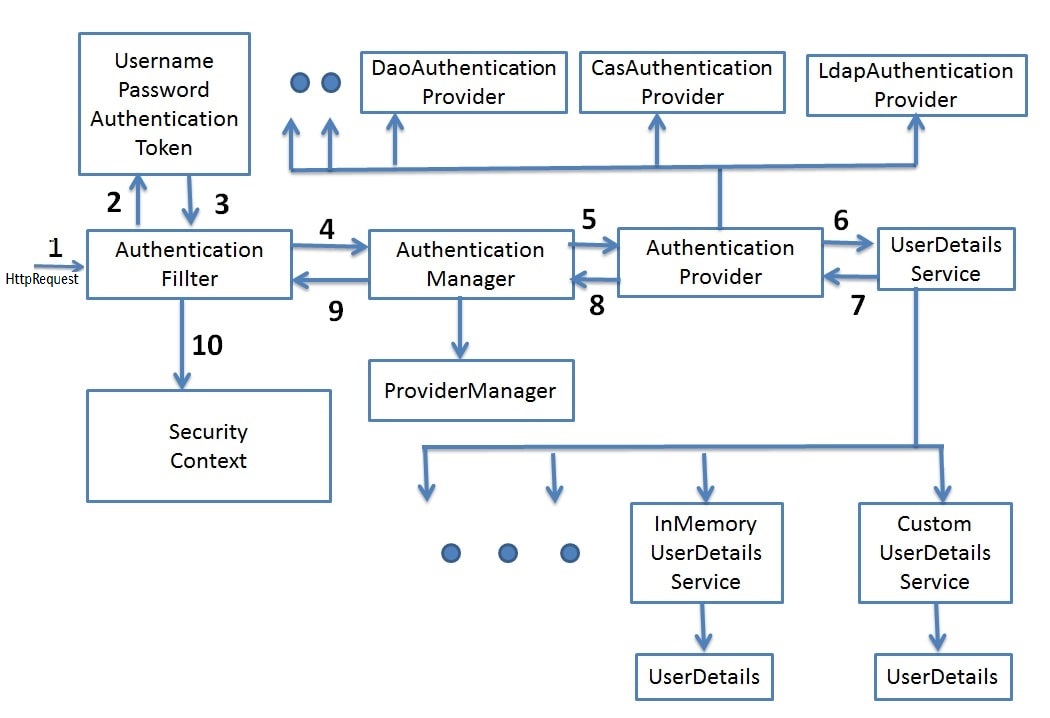
}

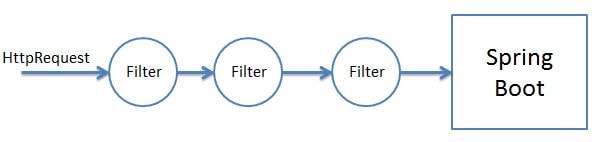
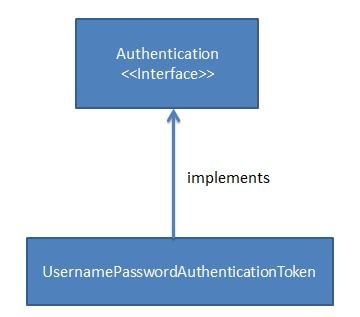
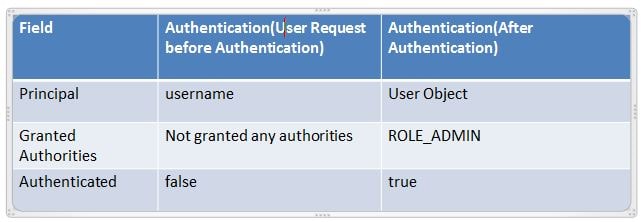
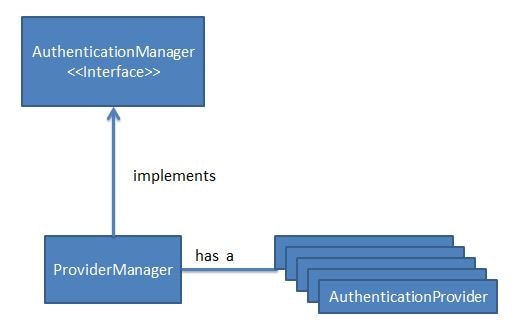
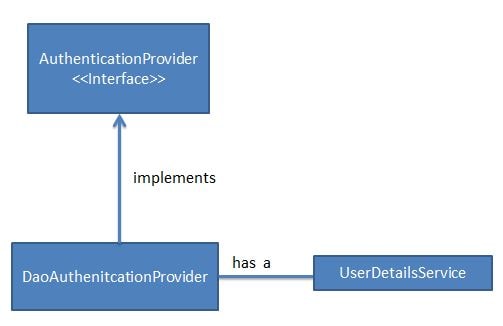
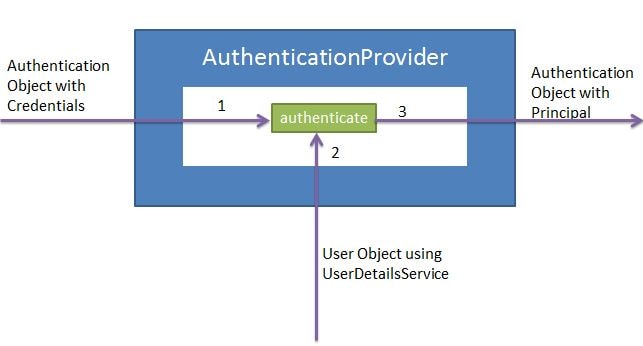
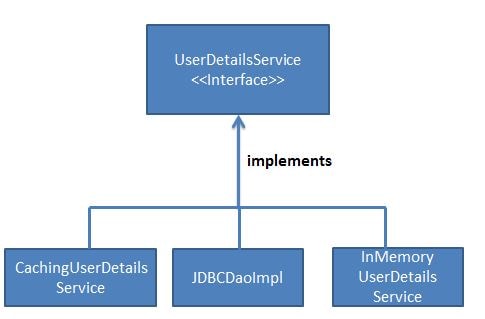
If we now start our Spring Boot project, we will be able to hit the webservice using Postman -  
Test /hellouser API  
  
Test /helloadmin API  


Configure Spring Security for Spring Boot Project

We will be configuring Spring Security for the Spring Boot project we just created. Let us first understand the Spring Security Architecture.

Understanding Spring Security Architecture

Let us understand how Spring Security Works.  


* **Filters -** Before the request reaches the Dispatcher Servlet, it is first intercepted by a chain of filters.  
    
  These filters are responsible for Spring Security. So any incoming request will go through these filters and it is here that authentication and authorization takes place. Based on the type of requests there are different Authentication Filters like the BasicAuthenticationFilter,UsernamePasswordAuthenticationFilter etc
* **Authentication Object Creation -** When the request is intercepted by the appropriate AuthenticationFilter it retrieves the username and password from the request and creates the Authentication Object. If the extracted credentials are username and password, then UsernamePasswordAuthenticationToken is created.  
  
* **AuthenicationManager -** Using the Authentication Object created the filter will then call the authenticate method of the Authentication Manager. The Authentication Manager is only a interface and actual implementation of the authenticate method is provided by the ProviderManager.  
    
  Important point to note here is that the Authentication Manager takes an Authentication object as input and after successful authentication again returns an object of type Authentication.  
    
  The ProviderManager has a list of AuthenticationProviders. From it's authenticate method it calls the authenticate method of the appropriate AuthenticateProvider. In response it gets the Principal Authentication Object if the authentication is successful.  
  
* **AuthenticationProvider -** The AuthenicationProvider is an interface with a single authenticate method.  
    
  It has various implementations like CasAuthenticationProvider,DaoAuthenticationProvider. Depending on the implementation an appropriate AuthenicationProvider implementation is used. It is in the AuthenticationProvider Implementation authenticate method where all the actual authentication takes place.  
    
  Using the UserDetails service the AuthenticationProvider fetches the User Object corresponding to the username. It fetches this User Object from either a database, internal memory or other sources. This User object credentials are then compared with the incoming Authentication Object credentials. If Authentication is successful then the Principal Authentication Object is returned in response.  
  
* **UserDetailsService -** The UserDetailsService is an interface having a single method named loadUserByUsername.  
    
  It has various implementations CachingUserDetailsService, JDBCDaoImpl etc. Based on the implementation an appropriate UserDetailsService is called.  
    
  It is responsible for fetching the User Object with username and password against which the incoming User Object will be compared.

Add Spring Security to Spring Boot

We will need to add the Spring Security Starter dependency in the pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.2.0.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>com.javainuse</groupId>

<artifactId>spring-security</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-security</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-security</artifactId>**

**</dependency>**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

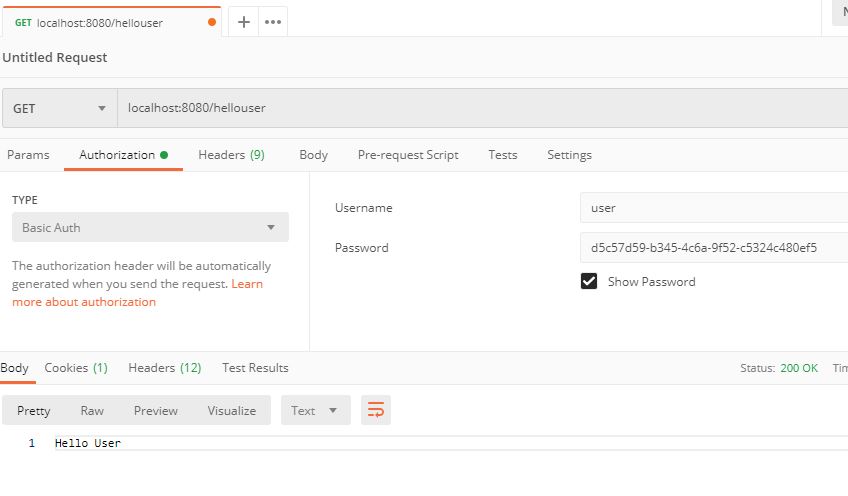
<artifactId>spring-boot-maven-plugin</artifactId>

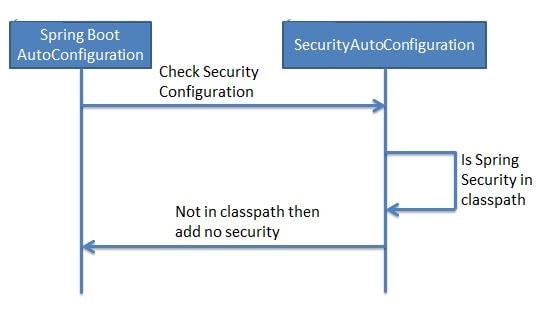
</plugin>

</plugins>

</build>

</project>

As soon as we add the spring security dependency to the project the basic authentication gets activated by default. If we now start the application, **Basic Security is enabled by default by Spring security due to the spring auto configurations.**  
In the console we get the password while the username is user-  
  
Let us have a look Spring Security Autoconfigurations.

* When no Spring Security dependency is added -  
  
* When Spring Security is added -

We will now be creating our own custom Spring Security Configuration by extending the WebSecurityConfigurerAdapter In this class we will be making use of the PasswordEncoder. In [previous tutorial we have already seen the need for password encoder.](https://www.javainuse.com/spring/boot_security_jdbc_authentication_bcrypt)

package com.javainuse.springsecurity.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration

@EnableWebSecurity

public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter{

@Autowired

CustomUserDetailsService userDetailsService;

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Override

public void configure(AuthenticationManagerBuilder auth) throws Exception

{

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

}

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

// We don't need CSRF for this example

httpSecurity.csrf().disable()

.authorizeRequests().antMatchers("/helloadmin")

.hasRole("ADMIN")

.antMatchers("/hellouser")

.hasAnyRole("ADMIN","USER")

.and().httpBasic();

}

}

Create a custom UserDetails Service class-

package com.javainuse.springsecurity.config;

import java.util.Arrays;

import java.util.List;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

@Service

public class CustomUserDetailsService implements UserDetailsService {

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

List<SimpleGrantedAuthority> roles=null;

if(username.equals("admin"))

{

roles = Arrays.asList(new SimpleGrantedAuthority("ROLE\_ADMIN"));

return new User("admin", "$2y$12$I0Di/vfUL6nqwVbrvItFVOXA1L9OW9kLwe.1qDPhFzIJBpWl76PAe",

roles);

}

else if(username.equals("user"))

{

roles = Arrays.asList(new SimpleGrantedAuthority("ROLE\_USER"));

return new User("user", "$2y$12$VfZTUu/Yl5v7dAmfuxWU8uRfBKExHBWT1Iqi.s33727NoxHrbZ/h2",

roles);

}

throw new UsernameNotFoundException("User not found with username: " + username);

}

}

Start the Spring Boot Application  
