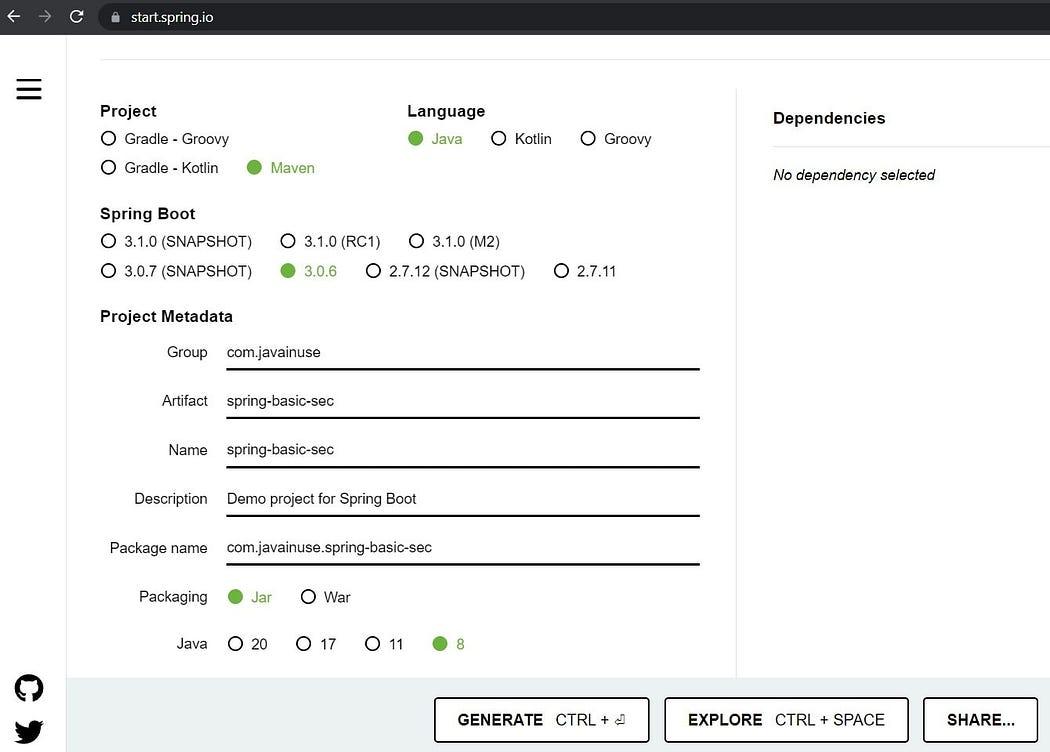
Spring Boot 3 Simple example

Go to [spring initializr website](https://start.spring.io/) and create a simple spring boot project as follows-



The maven project we will be creating is as follows -

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>3.0.6</version>

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

</parent>

<groupId>com.pack</groupId>

<artifactId>spring-basic-sec</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-basic-sec</name>

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

<properties>

<java.version>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>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

Next we will be creating a controller class to expose a **GET API** which returns a string.

package com.pack.springbasicsec.controller;

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

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

@RestController

public class TestController {

@GetMapping("/hello")

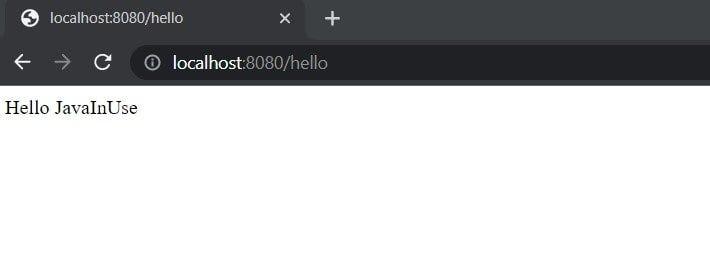
public String hello() {

return "Hello pack";

}

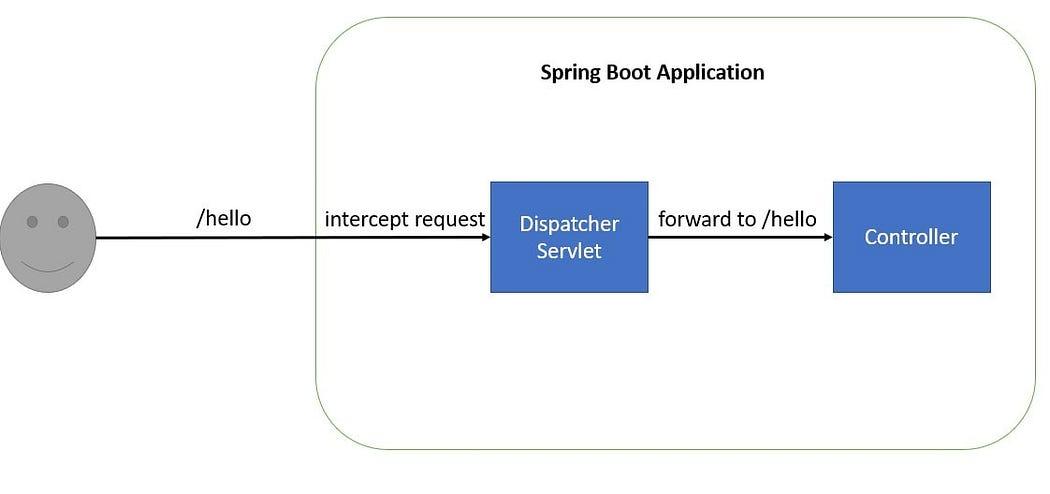
}

If we now go to localhost:8080/hello we get the following -



Internal Working-

Whenever we expose REST endpoints in Spring Boot, all the incoming requests are always first intercepted by the DispatcherServlet. The DispatcherServlet is the front controller in Spring web applications. The core responsibility of a DispatcherServlet is to dispatch incoming HttpRequests to the correct handlers. Next we will start the spring boot project in debug mode. Add a debug point in DispatcherServlet.



Spring Security

Add the spring security dependency to 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>3.0.6</version>

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

</parent>

<groupId>com.pack</groupId>

<artifactId>spring-basic-sec</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-basic-sec</name>

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

<properties>

<java.version>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>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

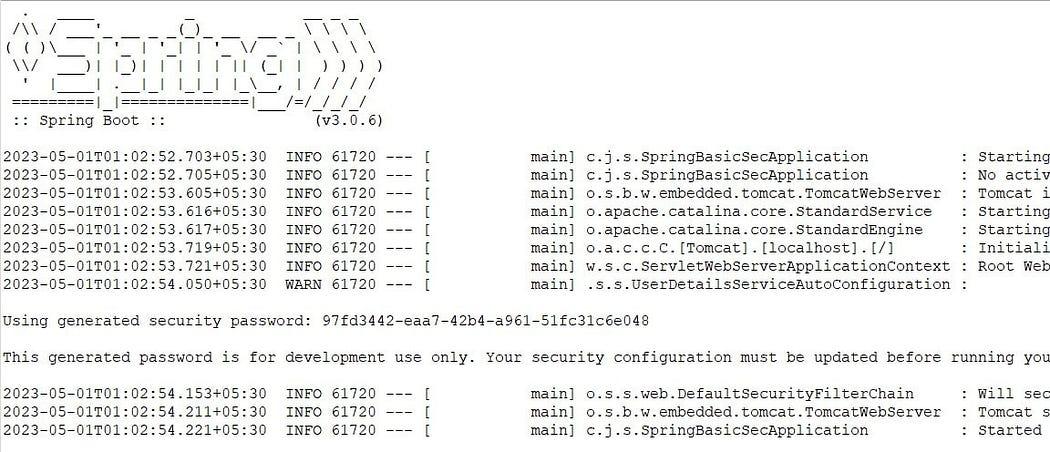
</plugin>

</plugins>

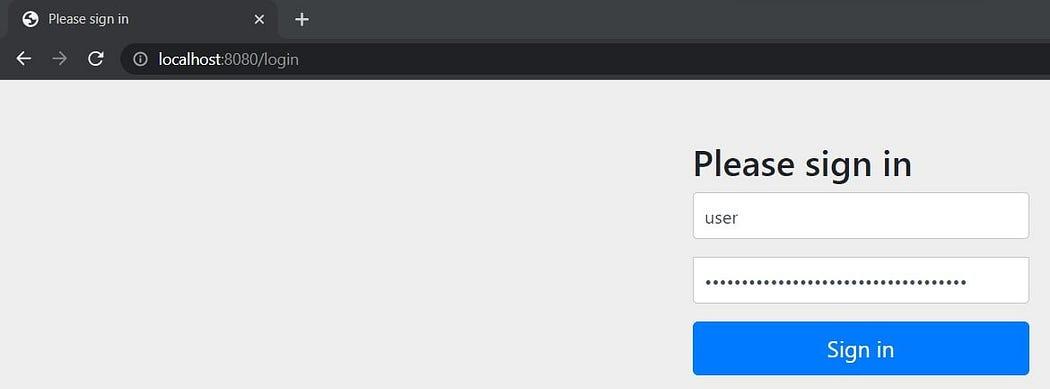
</build>

</project>

Once we add the spring security dependency in the application, spring security gets enabled by default. Start the Spring Boot application.

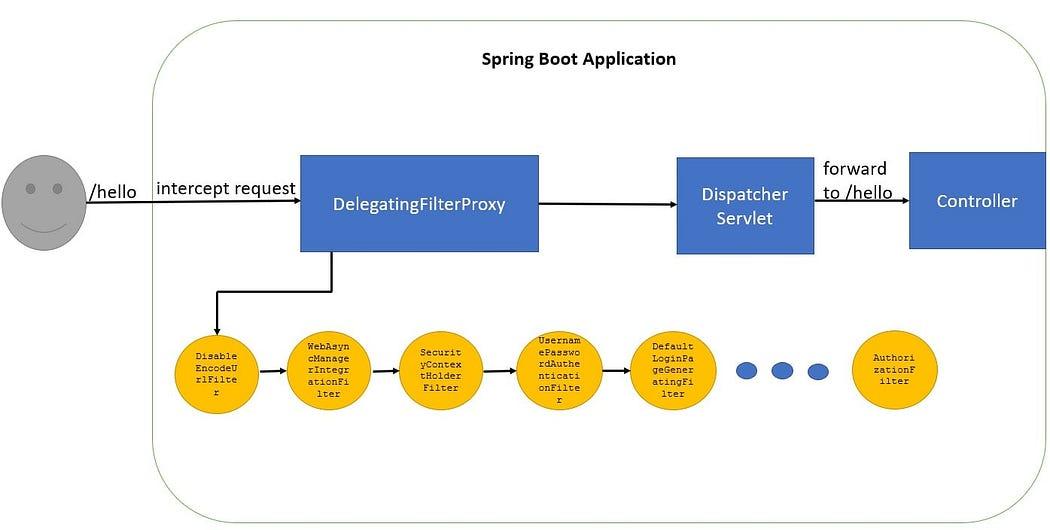


If we now go to localhost:8080/hello we are redirected to the default spring security login page. The default username for the form login is **user** and the password is printed by the spring boot application on startup console If we login, we can now access the /hello url.



Internal Working -

When we add the spring security dependency, spring security enables the security filter chain. 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.



Spring Security maintains a filter chain internally where each of the filters has a particular responsibility and filters are added or removed from the configuration depending on which services are required. Based on the type of requests there are different Authentication Filters like the BasicAuthenticationFilter,UsernamePasswordAuthenticationFilter. The ordering of the filters is important as there are dependencies between them.  
There will be three scenarios here -

Redirect to the default login page i.e redirect to /login -

Let us have a look at the filter chain in detail.  
We can see on startup the following filters are configured in the filter chain.

FilterChainProxy[Filter Chains: [DefaultSecurityFilterChain [RequestMatcher=any request, Filters=

[org.springframework.security.web.session.DisableEncodeUrlFilter@1b13467c,

org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@7bd96822,

org.springframework.security.web.context.SecurityContextHolderFilter@6824b913,

org.springframework.security.web.header.HeaderWriterFilter@f425231,

org.springframework.security.web.csrf.CsrfFilter@49a6f486,

org.springframework.security.web.authentication.logout.LogoutFilter@48a663e9,

org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@2dbfcf7,

org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@73c3cd09,

org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter@64dae3b7,

org.springframework.security.web.authentication.www.BasicAuthenticationFilter@7ed3df3b,

org.springframework.security.web.savedrequest.RequestCacheAwareFilter@705a8dbc,

org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@9e54c59,

org.springframework.security.web.authentication.AnonymousAuthenticationFilter@1fedf0a4,

org.springframework.security.web.access.ExceptionTranslationFilter@5eea5627,

org.springframework.security.web.access.intercept.AuthorizationFilter@45b15381]]]]

All the above Filters get called from the FilterChainProxy class. For this example we will mostly look at UsernamePasswordAuthenticationFilter, DefaultLoginPageGeneratingFilter and ExceptionTranslationFilter.

* **UsernamePasswordAuthenticationFilter —** This filter checks if the login url contains ‘/login’ and is a POST request or not. As this is a GET request with url ‘/hello’ so this filter does nothing.
* **DefaultLoginPageGeneratingFilter —** This filter checks if for the incoming request any authentication is done or not. Also it checks if the url is /login, /logout or /login?error. If it is niether of these it does nothing. In our case the url is /hello so it does nothing.
* **AuthorizationFilter —** This is the last filter in the filter chain. As no other filter has completed the authentication process, so this Filter throws an AccessDeniedException
* **ExceptionTranslationFilter —** This filter catches the AccessDeniedException. It then forces the application to redirect the browser to call the /login request.

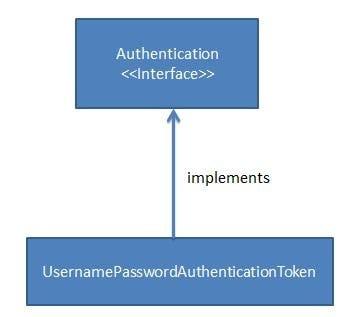
Display the default login page -

* **UsernamePasswordAuthenticationFilter —** This filter checks if the login url contains ‘/login’ and is a POST request or not. As this is a GET request with url ‘/login’ so this filter does nothing.
* **DefaultLoginPageGeneratingFilter —** This filter checks if for the incoming request any authentication is done or not. Also it checks if the url is /login, /logout or /login?error. If it is niether of these it does nothing. In our case the url is /login, so it generates a login page and returns to the user.

User enters credentials -

So now the browser is redirected to the /login page. The user enters the credentials and presses the submit button. This makes a POST call to /login url. Again the security filter chain intercepts the request. This time the following filters and classes get called.

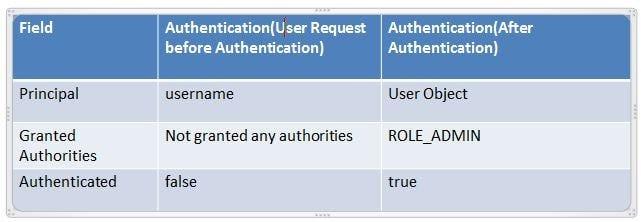
* **UsernamePasswordAuthenticationFilter —** This filter checks if the login url contains ‘/login’ and is a POST request or not. As this is a POST request with url ‘/login’ so this filter extracts the username and password from the requests and creates the Authentication Object i.e UsernamePasswordAuthenticationToken .



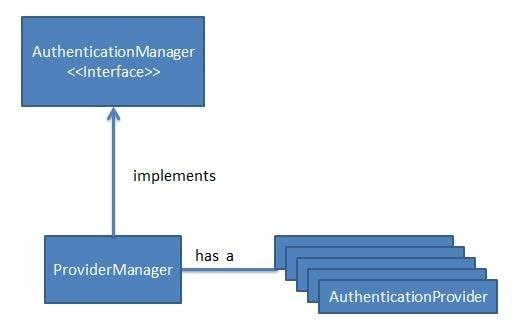
* **AuthenicationManager** — Using the Authentication Object created the filter will then call the authenticate method of the Authentication Manager. The Authentication Manager is only a interface and actual implementation of the authenticate method is provided by the ProviderManager.



* 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.



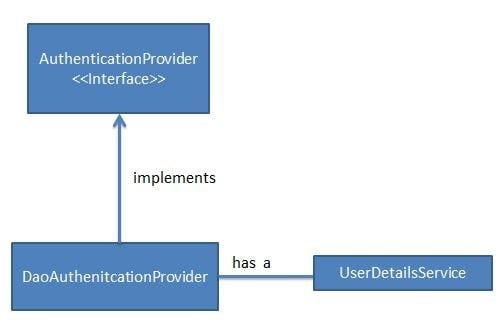
* The ProviderManager has a list of AuthenticationProviders. From it’s authenticate method it calls the authenticate method of the appropriate AuthenticateProvider. In response it gets the Principal Authentication Object if the authentication is successful.



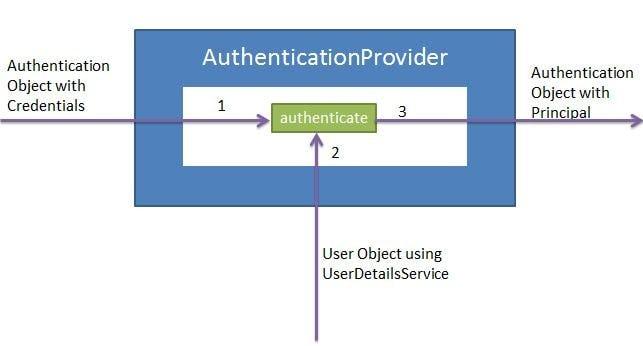
* **AuthenticationProvider** — The AuthenicationProvider is an interface with a single authenticate method.



* 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.



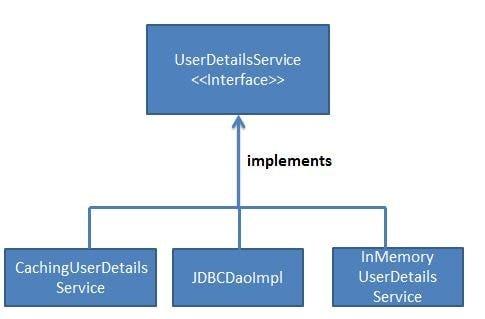
* Using the UserDetails service the AuthenticationProvider fetches the User Object corresponding to the username. It fetches this User Object from either a database, internal memory or other sources. This User object credentials are then compared with the incoming Authentication Object credentials. If 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