HowToDoInJava

Adding Role Based Security with Spring Boot REST APIs

Last Updated: January 28, 2022 By: Lokesh Gupta Spring Boot REST APIs

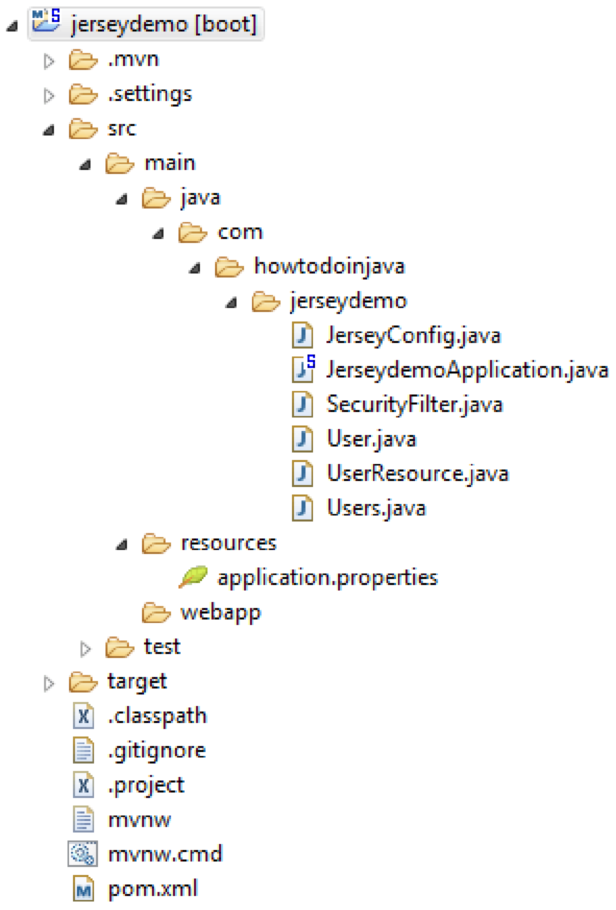
Learn to create JAX-RS 2.0 REST APIs using Spring Boot and Jersey framework, and add **role based security** using JAX-RS annotations e.g. `@PermitAll`, `@RolesAllowed` or `@DenyAll`.

Table of Contents

Project Structure
Create REST APIs
Secure REST APIs with JAX-RS Annotations
Write security filter using JAX-RS ContainerRequestFilter
Demo

Project Structure

The project structure of application created in this tutorial is as below:



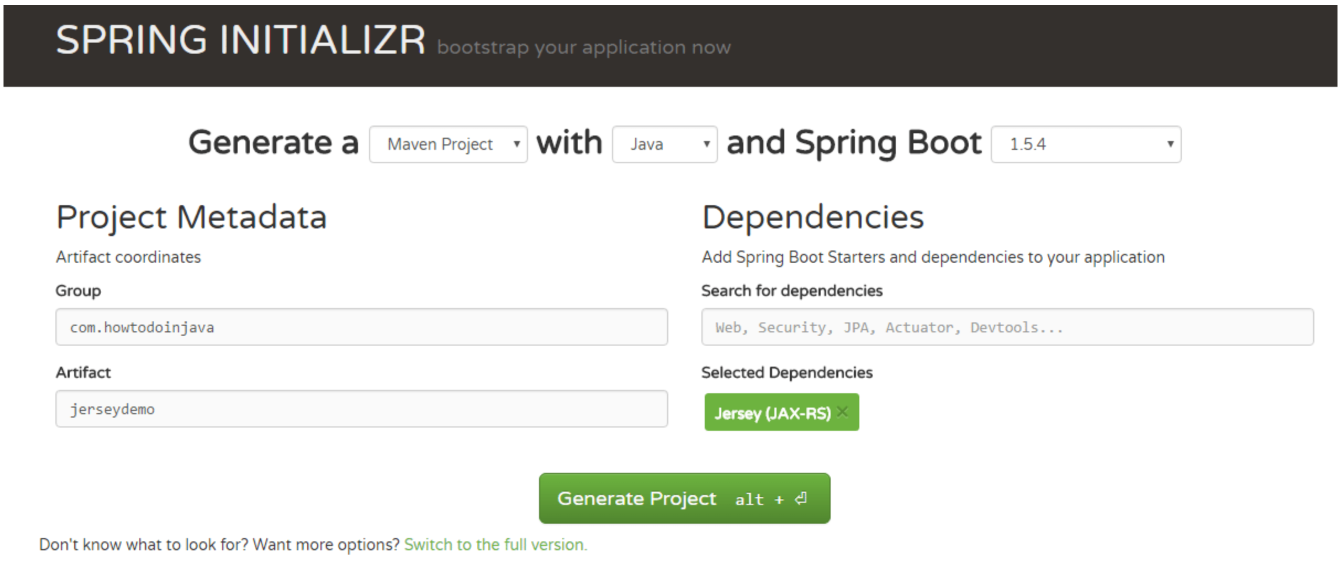
```
jerseydemo [boot]
  .mvn
  .settings
  src
    main
      java
        com
          howtodoinjava
            jerseydemo
              JerseyConfig.java
              JerseydemoApplication.java
              SecurityFilter.java
              User.java
              UserResource.java
              Users.java
      resources
        application.properties
      webapp
    test
  target
  .classpath
  .gitignore
  .project
  mvnw
  mvnw.cmd
  pom.xml
```

Spring Boot JAX-RS Security Demo – Project Structure

Create REST APIs

1. Create Spring Boot Project

Go to Spring Initializr portal and create spring boot application with **Jersey (JAX-RS)** dependency.



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.4

Project Metadata

Artifact coordinates

Group: com.howtodoinjava

Artifact: jerseydemo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Jersey (JAX-RS)

Don't know what to look for? Want more options? Switch to the full version.

Select Jersey in Spring Boot Initializr

2. **Import in Eclipse**

   Generate the project as zip file. Extract it in some place in your computer. Import the project as 'Existing maven application' into eclipse.

3. **Check maven dependencies**

   Check the maven file should have **spring-boot-starter-jersey** dependency in it.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

4. **Create REST APIs**

   Now create some JAX-RS resources which we will access into testing phase. I have created `UserResource` class.

UserResource.java

```
package com.howtodoinjava.jerseydemo;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.NONE)
@XmlRootElement(name = "users")
@Path("/users")
public class UserResource
{
  private static Map<Integer, User> DB = new HashMap<>();

  @GET
  @Produces("application/json")
  public Users getAllUsers() {
    Users users = new Users();
    users.setUsers(new ArrayList<>(DB.values()));
    return users;
  }

  @POST
  @Consumes("application/json")
  public Response createUser(User user) throws URISyntaxException
  {
    if(user.getFirstName() == null || user.getLastName() == null) {
      return Response.status(400).entity("Please provide all mandatory input
    }
    user.setId(DB.values().size()+1);
    user.setUri("/user-management/"+user.getId());
    DB.put(user.getId(), user);
    return Response.status(201).contentLocation(new URI(user.getUri())).buil
  }

  @GET
  @Path("/{id}")
  @Produces("application/json")
  public Response getUserById(@PathParam("id") int id) throws URISyntaxExcep
  {
    User user = DB.get(id);
    if(user == null) {
      return Response.status(404).build();
    }
    return Response
        .status(200)
        .entity(user)
        .contentLocation(new URI("/user-management/"+id)).build();
  }

  @PUT
  @Path("/{id}")
  @Consumes("application/json")
  @Produces("application/json")
  public Response updateUser(@PathParam("id") int id, User user) throws URIS
  {
    User temp = DB.get(id);
    if(user == null) {
      return Response.status(404).build();
    }
    temp.setFirstName(user.getFirstName());
    temp.setLastName(user.getLastName());
    DB.put(temp.getId(), temp);
    return Response.status(200).entity(temp).build();
  }

  @DELETE
  @Path("/{id}")
  public Response deleteUser(@PathParam("id") int id) throws URISyntaxExcept
    User user = DB.get(id);
    if(user != null) {
      DB.remove(user.getId());
      return Response.status(200).build();
    }
    return Response.status(404).build();
  }

  static
  {
    User user1 = new User();
    user1.setId(1);
    user1.setFirstName("John");
    user1.setLastName("Wick");
    user1.setUri("/user-management/1");

    User user2 = new User();
    user2.setId(2);
    user2.setFirstName("Harry");
    user2.setLastName("Potter");
    user2.setUri("/user-management/2");

    DB.put(user1.getId(), user1);
    DB.put(user2.getId(), user2);
  }
}
```

Users.java

```java
package com.howtodoinjava.jerseydemo;

import java.util.ArrayList;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.NONE)
@XmlRootElement(name = "users")
public class Users {

    @XmlElement(name="user")
    private ArrayList<User> users;

    public ArrayList<User> getUsers() {
        return users;
    }

    public void setUsers(ArrayList<User> users) {
        this.users = users;
    }
}
```

User.java

```java
package com.howtodoinjava.jerseydemo;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.NONE)
@XmlRootElement(name = "user")
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

    @XmlAttribute(name = "id")
    private int id;

    @XmlAttribute(name="uri")
    private String uri;

    @XmlElement(name = "firstName")
    private String firstName;

    @XmlElement(name = "lastName")
    private String lastName;

    // Getters and Setters
}
```

5. Configure Jersey

Now we have a JAX-RS resource and we want to access it from spring boot application which include Jersey dependency. Let's register this resource as Jersey resource.

```
package com.howtodoinjava.jerseydemo;

import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.stereotype.Component;

@Component
public class JerseyConfig extends ResourceConfig
{
  public JerseyConfig()
  {
    register(SecurityFilter.class);
    register(UserResource.class);
  }
}
```

- Look at the `@Component` annotation. It enables this class to be registered while spring boot auto scans the java classes in source folder.
- **`ResourceConfig`** provides advanced capabilities to simplify registration of JAX-RS components.
- `SecurityFilter` class is the actual auth details processor which we will see later in this tutorial.

Extend spring boot application with `SpringBootServletInitializer`.

```
package com.howtodoinjava.jerseydemo;

import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

@SpringBootApplication
public class JerseydemoApplication extends SpringBootServletInitializer
{
  public static void main(String[] args)
  {
    new JerseydemoApplication().configure(new SpringApplicationBuilder(Jerse
  }
}
```

Secure REST APIs with JAX-RS Annotations

Now when our APIs are ready, we will start securing them. Let's annotate the APIs with **JAX-RS annotations** based on their desired access level and user roles allowed to access them.

```
package com.howtodoinjava.jerseydemo;

@XmlAccessorType(XmlAccessType.NONE)
@XmlRootElement(name = "users")
@Path("/users")
public class UserResource
{
  private static Map<Integer, User> DB = new HashMap<>();

  @GET
  @PermitAll
  @Produces("application/json")
  public Users getAllUsers() {
    Users users = new Users();
    users.setUsers(new ArrayList<>(DB.values()));
    return users;
  }

  @POST
  @Consumes("application/json")
  @RolesAllowed("ADMIN")
  public Response createUser(User user) throws URISyntaxException
  {
    if(user.getFirstName() == null || user.getLastName() == null) {
      return Response.status(400).entity("Please provide all mandatory inputs").bu
    }
    user.setId(DB.values().size()+1);
    user.setUri("/user-management/"+user.getId());
    DB.put(user.getId(), user);
    return Response.status(201).contentLocation(new URI(user.getUri())).build();
  }

  @GET
  @Path("/{id}")
  @Produces("application/json")
  @PermitAll
  public Response getUserById(@PathParam("id") int id) throws URISyntaxException
  {
    User user = DB.get(id);
    if(user == null) {
      return Response.status(404).build();
    }
    return Response
        .status(200)
        .entity(user)
        .contentLocation(new URI("/user-management/"+id)).build();
  }

  @PUT
  @Path("/{id}")
  @Consumes("application/json")
  @Produces("application/json")
  @RolesAllowed("ADMIN")
  public Response updateUser(@PathParam("id") int id, User user) throws URISyntaxE
  {
    User temp = DB.get(id);
    if(user == null) {
      return Response.status(404).build();
    }
    temp.setFirstName(user.getFirstName());
    temp.setLastName(user.getLastName());
    DB.put(temp.getId(), temp);
    return Response.status(200).entity(temp).build();
  }

  @DELETE
  @Path("/{id}")
  @RolesAllowed("ADMIN")
  public Response deleteUser(@PathParam("id") int id) throws URISyntaxException {
    User user = DB.get(id);
    if(user != null) {
      DB.remove(user.getId());
      return Response.status(200).build();
    }
```

```
        return Response.status(404).build();
    }

    static
    {
        User user1 = new User();
        user1.setId(1);
        user1.setFirstName("John");
        user1.setLastName("Wick");
        user1.setUri("/user-management/1");

        User user2 = new User();
        user2.setId(2);
        user2.setFirstName("Harry");
        user2.setLastName("Potter");
        user2.setUri("/user-management/2");

        DB.put(user1.getId(), user1);
        DB.put(user2.getId(), user2);
    }
}
```

You can see the security related JAX-RS annotations in above highlighted lines.

Write security filter using JAX-RS ContainerRequestFilter

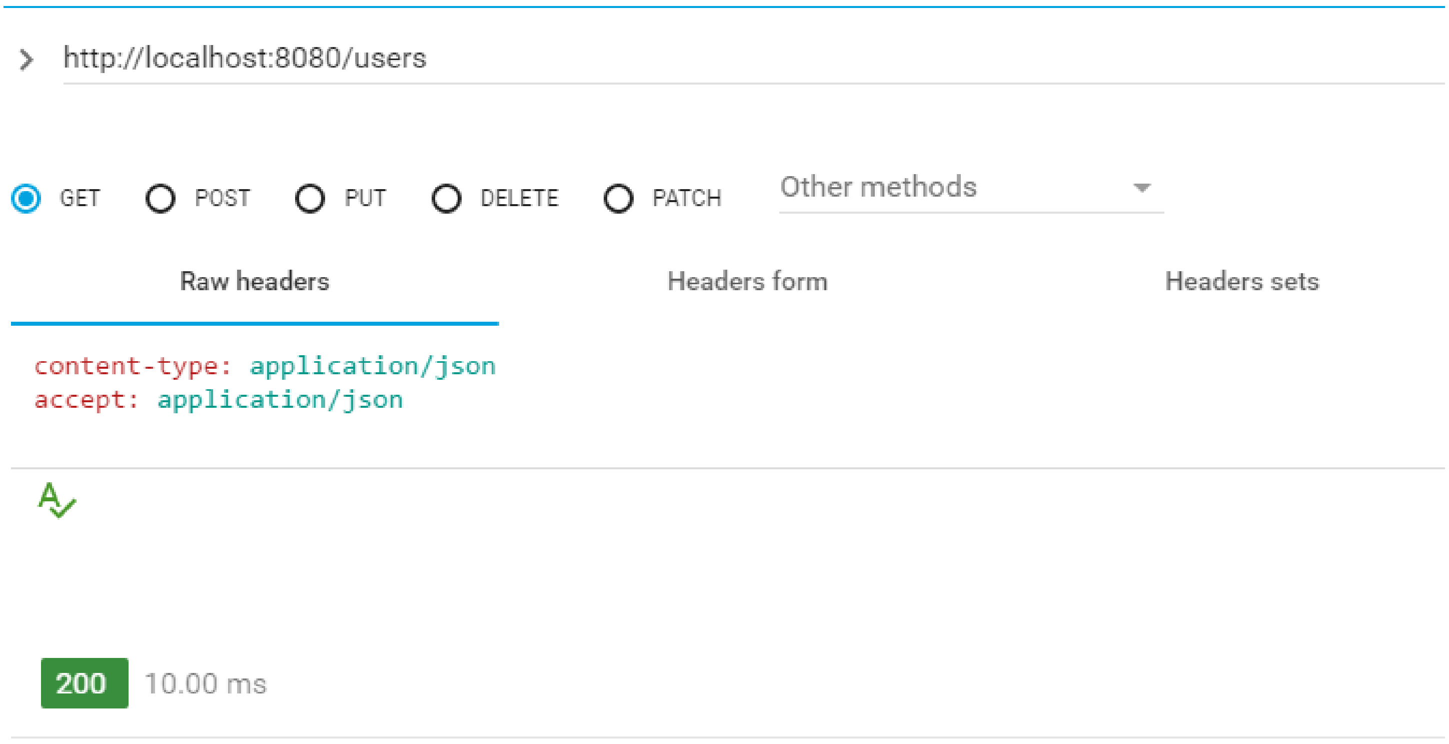
Now it's time to write our security filter which will examine the incoming requests, fetch the authorization information (basic auth in this example), and then will match user name and password, and finally it will verify the user's access level by it's role. If everything matches, API will be accessed else user will get access denied response.

```
package com.howtodoinjava.jerseydemo;

import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.Base64;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

import javax.annotation.security.DenyAll;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;

/**
 * This filter verify the access permissions for a user based on
 * user name and password provided in request
 * */
@Provider
public class SecurityFilter implements javax.ws.rs.container.ContainerRequestFilte
{
    private static final String AUTHORIZATION_PROPERTY = "Authorization";
    private static final String AUTHENTICATION_SCHEME = "Basic";
    private static final Response ACCESS_DENIED = Response.status(Response.Status.
    private static final Response ACCESS_FORBIDDEN = Response.status(Response.Stat
    private static final Response SERVER_ERROR = Response.status(Response.Status.I

    @Context
    private ResourceInfo resourceInfo;

    @Override
    public void filter(ContainerRequestContext requestContext)
    {
        Method method = resourceInfo.getResourceMethod();
        //Access allowed for all
        if( ! method.isAnnotationPresent(PermitAll.class))
        {
            //Access denied for all
            if(method.isAnnotationPresent(DenyAll.class))
            {
                requestContext.abortWith(ACCESS_FORBIDDEN);
                return;
            }

            //Get request headers
            final MultivaluedMap<String, String> headers = requestContext.getHeade

            //Fetch authorization header
            final List<String> authorization = headers.get(AUTHORIZATION_PROPERTY)

            //If no authorization information present; block access
            if(authorization == null || authorization.isEmpty())
            {
                requestContext.abortWith(ACCESS_DENIED);
                return;
            }
            //Get encoded username and password
            final String encodedUserPassword = authorization.get(0).replaceFirst(A

            //Decode username and password
            String usernameAndPassword = null;
            try {
                usernameAndPassword = new String(Base64.getDecoder().decode(encode
            } catch (Exception e) {
                requestContext.abortWith(SERVER_ERROR);
                return;
            }

            //Split username and password tokens
            final StringTokenizer tokenizer = new StringTokenizer(usernameAndPassw
            final String username = tokenizer.nextToken();
            final String password = tokenizer.nextToken();

            //Verifying Username and password
            if(!(username.equalsIgnoreCase("admin") && password.equalsIgnoreCase("
              requestContext.abortWith(ACCESS_DENIED);
                return;
            }

            //Verify user access
            if(method.isAnnotationPresent(RolesAllowed.class))
            {
                RolesAllowed rolesAnnotation = method.getAnnotation(RolesAllowed.c
                Set<String> rolesSet = new HashSet<String>(Arrays.asList(rolesAnno

                //Is user valid?
                if( ! isUserAllowed(username, password, rolesSet))
                {
                    requestContext.abortWith(ACCESS_DENIED);
                    return;
                }
            }
        }
    }
    private boolean isUserAllowed(final String username, final String password, fi
    {
        boolean isAllowed = false;

        //Step 1. Fetch password from database and match with password in argument
        //If both match then get the defined role for user from database and conti
        //Access the database and do this part yourself
        //String userRole = userMgr.getUserRole(username);
        String userRole = "ADMIN";

        //Step 2. Verify user role
        if(rolesSet.contains(userRole))
        {
            isAllowed = true;
        }
        return isAllowed;
    }
}
```

Demo

Run the project as Spring boot application. Now test rest resources.

Access GET /users resource

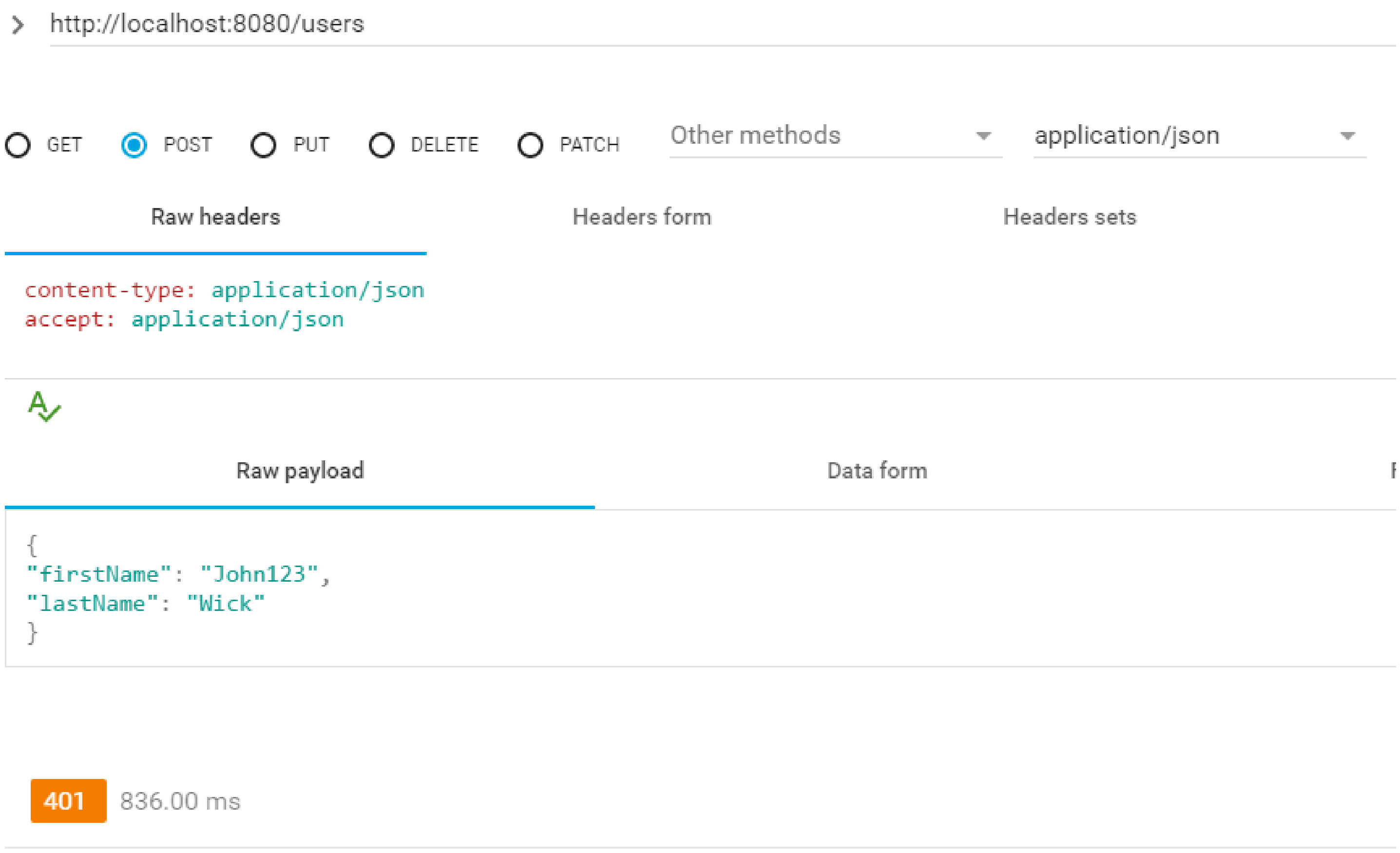

http://localhost:8080/users

```
content-type: application/json
accept: application/json
```

GET Users collection

Access POST /users resource with no authtentication details

Look at the returned status code 401.



```
content-type: application/json
accept: application/json
```

```
{
"firstName": "John123",
"lastName": "Wick"
}
```

Auth Required for POST APIs

Access POST /users resource with authentication details added

Use this link to generate base64 encoded user name and password combination to pass into `Authorization` header.

Request with Auth Details Success

Drop me your questions in comments section.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Adding Custom Headers to Spring Boot REST APIs
2. Adding Multiple Items to ArrayList

3. Create Jersey REST APIs with Spring Boot
4. JUnit 4 – Assumption Based Testcases
5. Location-Based Currency Formatting in Java
6. Java Version – Time-Based Release Versioning
7. Location based Date Time Formatting
8. Building REST APIs with Spring Boot
9. REST API Security Guide
0. Jersey REST API Security Example

3 thoughts on “Adding Role Based Security with Spring Boot REST APIs”

Minh

December 16, 2021 at 2:08 pm

I get the error like that
Error starting ApplicationContext. To display the conditions report re-run your application with ‘debug’ enabled.
2021-12-16 09:34:08 ERROR [main] org.springframework.boot.SpringApplication: Application run failed
org.springframework.context.ApplicationContextException: Unable to start web server;
nested exception is org.springframework.context.ApplicationContextException: Unable to start ServletWebServerApplicationContext due to missing ServletWebServerFactory bean.

I have added @SpringBootApplication and extends *SpringBootServletInitializer* in the main class.
in gradle.build I have added ‘spring-boot-starter-jersey’, spring-boot-starter-web already

Reply

jeeva

August 3, 2019 at 9:04 pm

super example easy to understand every one …
I need your help how decrypt password from properties file using spring boot application .
note: already i have declared encrypted values in properties file

Reply

Jenny

August 30, 2018 at 8:35 pm

Excellent Example Lokesh Gupta.

Reply

Leave a Comment

Name *

Email *

Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

Discover how

S

Platform = Operating System + Processor
ubuntu
CPU