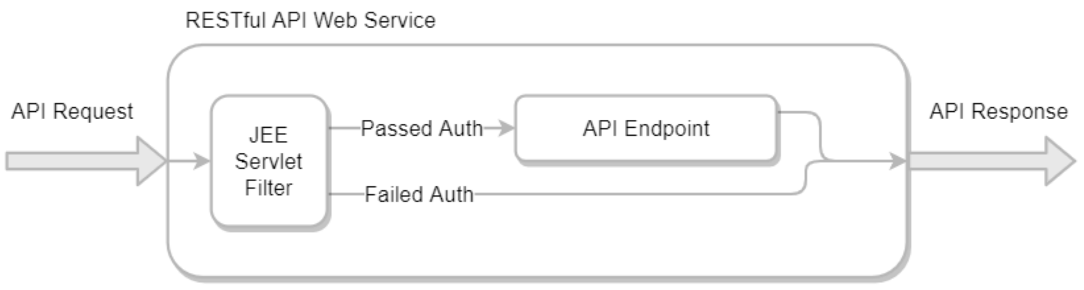
Lab 1 - Authentication with JWT

In this lab we'll create a web service (api) and secure it using JSON web tokens. Our goal is twofold:

1. Understand how authentication of API services works.
2. Learn how to use a JWT code library to create, read and validate tokens.

The lab takes a step by step approach. We'll start with an unsecured JEE (Java Enterprise Edition) RESTful API Web service and add a JEE Servlet Filter that intercepts calls to the service, checks for the presence of a JWT token on the request and determines if the request should go through or not.

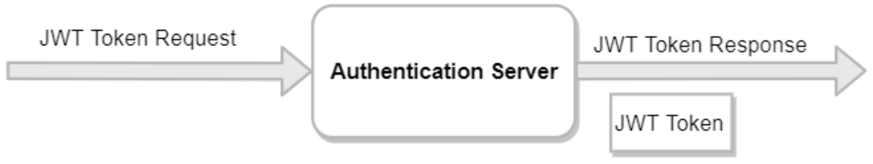


The filter will pass requests on to the API Endpoint as long as:

1. The 'Authorization' (token) header exists
2. The token itself (the header's value string) is valid

The JEE Servlet filter will use a JWT utility class to read and validate the token. The initial version of the JWT Utility will simply check the length of the token string. Later on in the lab we will update the utility to work with real JWT tokens.

Next we create an Authentication server that we can call to get tokens. This server will use the JWT Utility class to create new tokens:



Once the Authentication server is up and running we'll test the system by:

1. Calling the Authentication service to get a token
2. Passing the token when making a request of the data API endpoint

When we've determined that the system is working we will update the JWT utility code to create and verify real JWT tokens. The system will then be re-tested using the actual JWT tokens.

In the real world the Authentication server and the API data service are run as separate applications on separate servers. For this lab though, to simplify development and testing, we will be running these services as separate endpoints on the same server.

Our authentication system will be developed over the course of the following lab sections:

1. Installing and Running the Starter Application
2. Add a Servlet Filter that Checks the Authorization Header
3. Create an API Service for Retrieving JWT Tokens
4. Update the JWT Utility to Manage Real Tokens
5. Test the JWT Protected API

Part 1 - Installing and Running the Starter Application

The starter application is a simple "Customer" api using:

- Spring Boot
- Maven
- Gradle

To reduce setup issues data is hard coded in the service. This is fine for our purposes as the focus of this lab is integration of JWT tokens and not the service itself.

__1. If you don't already have one, create a LabWork directory somewhere on the development machine (e.g. "c:\LabWork" on windows, "{user-home}\LabWork" on linux)

__2. Unzip \LabFiles\jwt-lab\jwt-lab-starter.zip and copy the resulting "jwt-lab" directory into your LabWork folder.

__3. Open Eclipse

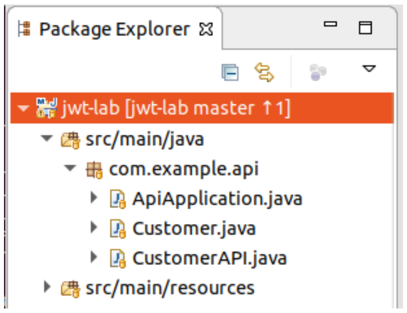
__4. Import the project into eclipse using the following steps:

1. Choose Import from the File menu
2. Choose Maven\"Existing Maven Projects"
3. The dialog box that appears should show "pom.xml" with a check mark next to it.
4. Click "Finish" and wait for the project to finish importing

__5. Right click on the project in the 'Package Explorer" and choose **Run-As/Maven Install** to retrieve dependencies and compile the project.

__6. Navigate under 'src/main/java' in the 'Package Explorer' until you see

'ApiApplication.java'

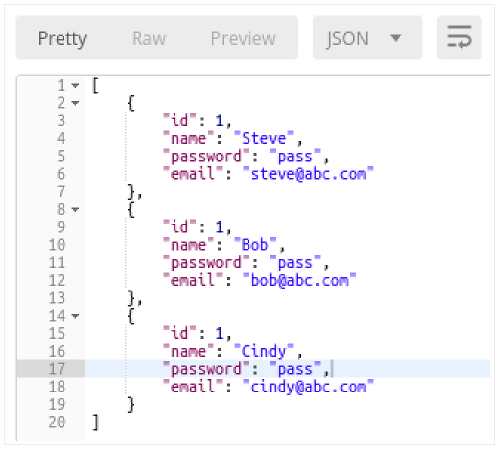


__7. Right-Click on 'ApiApplication.java' and choose Run-As/Run As Java Application. This is how we will be running the server throughout the lab.

__8. Open the Postman REST client and execute the following request:

```
Type: GET
URL: localhost:8080/customers
```

__9. You should get the following results:



```
[
    {
        "id": 1,
        "name": "Steve",
        "password": "pass",
        "email": "steve@abc.com"
    },
    {
        "id": 1,
        "name": "Bob",
        "password": "pass",
        "email": "bob@abc.com"
    },
    {
        "id": 1,
        "name": "Cindy",
        "password": "pass",
        "email": "cindy@abc.com"
    }
]
```

At this point the Customer service allows anyone to retrieve data without authentication or authorization. We will update the service during the course of this lab. In the next part we will alter the service so that it checks for an authorization header.

Part 2 - Add a Servlet Filter that Checks the Authorization Header

The Java Enterprise edition allows developers to run code after a request comes in and before it is sent to the API using Servlet filters. In this part we will add a Servlet filter and some supporting classes that will require users to add an Authorization header to their requests in order to retrieve Customer data.

__1. Copy the following files from LabFiles into the project:

```
Copy Files:
    AuthFilter.java
    Token.java
    JWTUtil.java
    JWTMockUtil.java

From Dir: LabFiles/jwt-lab/
To Dir: LabWork/jwt-lab/src/main/java/com/example/api/
```

__2. We will take a look at these files one at a time:

__3. Open **Token.java** in your editor.

```
package com.example.api;

public class Token {
          String token;

          public Token(String token) {
                    super();
                    this.token = token;
          }

          public String getToken() {
                    return token;
          }

          public void setToken(String token) {
                    this.token = token;
          }

}
```

Token is a basic data object. It holds a string representing the token and has methods for setting and getting the string.

__4. Open **JWTUtil.java** in your editor:

```
package com.example.api;

public interface JWTUtil {
    public boolean verifyToken(String jwt_token);
    public String getScopes(String jwt_token) ;
    public Token createToken(String scopes) ;
}
```

JWTUtil.java is an interface. It specifies three methods for working with tokens including:

- `createToken`
- `verifyToken`
- `getScopes`

We will start with a simple implementation of of JWTUtil that creates and manages a mock token. Later we will use an implementation based on a 3rd party code library that allows us to work with real JWT tokens. Doing this in stages will make it easier to test as we go along.

__5. Open **JWTMockUtil.java** in your editor:

```
package com.example.api;

public class JWTMockUtil implements JWTUtil {

  public boolean verifyToken(String jwt_token) {
    // mock tokens greater than 10 characters are always valid
    if(jwt_token != null && jwt_token.length()> 10) {
          return true;
    }
    return false;
  }

  public String getScopes(String jwt_token) {
    // mock method always returns the correct customer scope
    return "com.api.customer.r";
  }

  public Token createToken(String scopes) {
    // create and return a mock token
    Token token = new Token("lasjflasjdfasljk " + scopes + " sdjfkljd");
    return token;
  }

}
```

The createToken method creates a mock token consisting of some random text and the passed in 'scopes' string.

The verifyToken method is used to check tokens that come in on requests. Any token greater than 10 characters in length passes.

The getScopes method in this simple mock implementation is hard coded to return a scope string that matches the scope string of the customer api.

The mock versions of these methods are sufficient for testing the AuthFilter we are about to use. Later, once we have verified that the filter is working, we will substitute implementations that work with real JWT tokens.

__6. Open the **AuthFilter.java** file in your editor:

```
package com.example.api;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;

@Component
public class AuthFilter implements Filter {

  JWTUtil jwtUtil = new JWTMockUtil();
  // JWTUtil jwtUtil = new JWTHelper();

  private String api_scope = "com.api.customer.r";

  @Override
  public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
      throws IOException, ServletException {

    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse res = (HttpServletResponse) response;
    String uri = req.getRequestURI();
    if (uri.startsWith("/token")) {
      // continue on to get-token endpoint
      chain.doFilter(request, response);
      return;
    } else {
      // check JWT token
      String authheader = req.getHeader("authorization");
      if (authheader != null && authheader.length() > 7
          && authheader.startsWith("Bearer")) {
        String jwt_token = authheader.substring(7, authheader.length());
        if (jwtUtil.verifyToken(jwt_token)) {
          String request_scopes = jwtUtil.getScopes(jwt_token);
          if (request_scopes.contains(api_scope)) {
            // continue on to api
            chain.doFilter(request, response);
            return;
          }
        }
      }
    }
    // reject request and return error instead of data
    res.sendError(HttpServletResponse.SC_FORBIDDEN, "failed
authentication");
  }

}
```

Take a moment to read through the code. It includes a single class named AuthFilter that implements a single method doFilter.

The AuthFilter class implements the javax.servlet.Filter interface.

The doFilter method holds code that gets the value of the authorization header, makes sure it is not null or too short, extracts the token from the value portion of the header and calls the jwtUtil implementation to verify it.

```
if (jwtUtil.verifyToken(jwt_token)) { … }
```

After verifying the token it checks to make sure the scopes passed in the token match the scope required for the api which appears as a property of the filter:

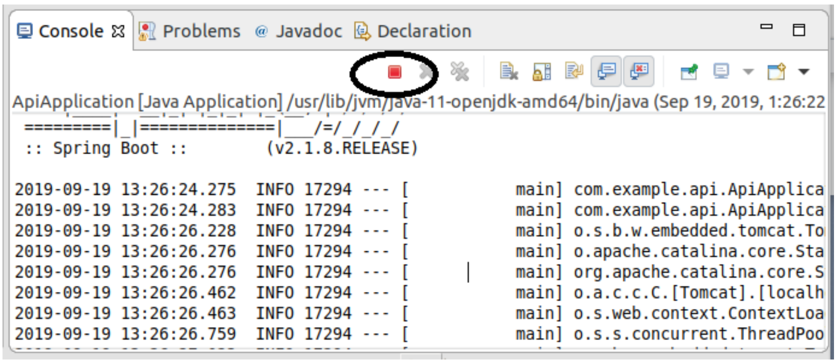
```
private String api_scope = "com.api.customer.r";
```

For now the code is using the mock token utility implementation.

```
JWTUtil jwtUtil = new JWTMockUtil();
```

__7. Close the editor windows.

__8. Shut down the existing instance of the api server by clicking on the red 'Terminate' box on the 'Console' tab.



```
ApiApplication [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Sep 19, 2019, 1:26:22
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.8.RELEASE)

2019-09-19 13:26:24.275  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:24.283  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:26.228  INFO 17294 --- [           main] o.s.b.w.embedded.tomcat.To
2019-09-19 13:26:26.276  INFO 17294 --- [           main] o.apache.catalina.core.Sta
2019-09-19 13:26:26.276  INFO 17294 --- [           main] org.apache.catalina.core.S
2019-09-19 13:26:26.462  INFO 17294 --- [           main] o.a.c.c.C.[Tomcat].[localh
2019-09-19 13:26:26.463  INFO 17294 --- [           main] o.s.web.context.ContextLoa
2019-09-19 13:26:26.759  INFO 17294 --- [           main] o.s.s.concurrent.ThreadPoo
```

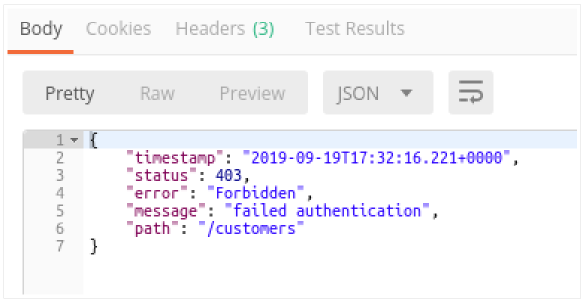
__9. Right click on the project in the 'Package Explorer" and choose **Run-As/Maven Install** to retrieve dependencies and compile the project.

__10. Right-Click on '**ApiApplication.java**' and choose **Run-As/Run As Java Application**.

__11. Open the Postman REST client and execute the same request we tried earlier:

```
Type: GET
URL: localhost:8080/customers
```

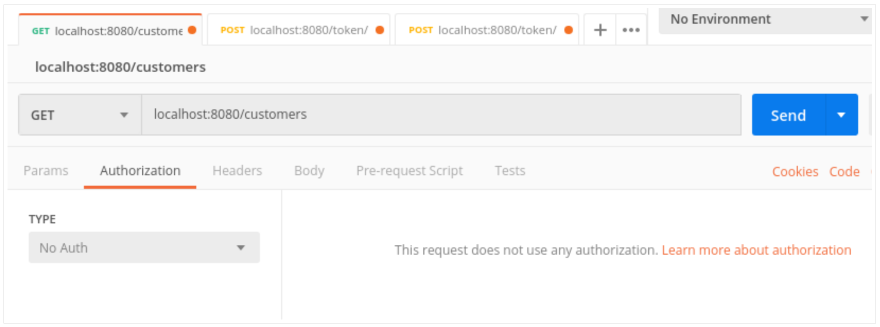
__12. This time you should get an error like this:



```
{
    "timestamp": "2019-09-19T17:32:16.221+0000",
    "status": 403,
    "error": "Forbidden",
    "message": "failed authentication",
    "path": "/customers"
}
```

We did not add an “authorization” header to the request so the servlet filter denies us access to the data.

__13. Go back to Postman and click on the Authorization tab.


GET localhost:8080/custome
POST localhost:8080/token/
POST localhost:8080/token/
No Environment
Send
Params
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Learn more about authorization

localhost:8080/customers

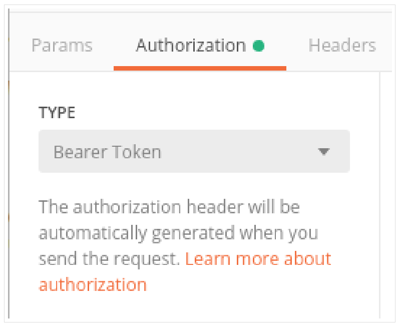
GET localhost:8080/customers

TYPE

No Auth

This request does not use any authorization.

__14. Next click on the TYPE dropdown and choose “Bearer Token”.



TYPE

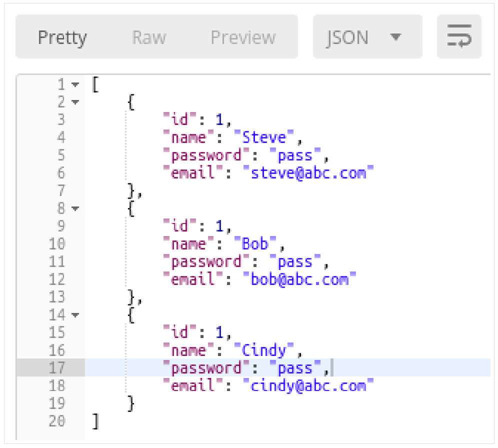
Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

__15. To the right of the drop down is a field labeled 'Token'. Enter the following string into the field.

```
abcdefghigklmnop
```

__16. Click the Send button to send the request, it should return the customer data:



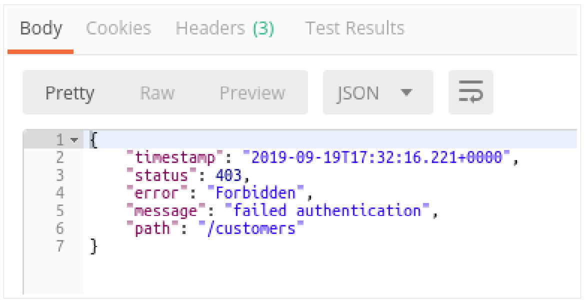
```
[
    {
        "id": 1,
        "name": "Steve",
        "password": "pass",
        "email": "steve@abc.com"
    },
    {
        "id": 1,
        "name": "Bob",
        "password": "pass",
        "email": "bob@abc.com"
    },
    {
        "id": 1,
        "name": "Cindy",
        "password": "pass",
        "email": "cindy@abc.com"
    }
]
```

It works because the mock JWTUtil class we are currently using only checks for strings over 10 characters long.

__17. Try the same request but with a shorter string for the token:

```
abcdefg
```

__18. Click on Send again after updating the token with the shorter string. This time the request should again fail.



```
{
    "timestamp": "2019-09-19T17:32:16.221+0000",
    "status": 403,
    "error": "Forbidden",
    "message": "failed authentication",
    "path": "/customers"
}
```

We have verified that our filter works. In the next part of the lab we will test out a JWTUtil implementation that uses a 3rd party library to create and verify real JWT tokens.

Part 3 - Create an API Service for Retrieving JWT Tokens

In our tests we passed a random string as a token when calling the Customer API. In the real world JWT tokens needed to make API calls are obtained from an Authorization service. In this part of the lab we will create an Authorization service that can be used to obtain token strings.

__1. Copy the following files from LabFiles into the project:

```
Copy Files:
    TokenRequestData.java
    Authenticator.java
    TokenAPI.java

From Dir: LabFiles/jwt-lab/
To Dir: LabWork/jwt-lab/src/main/java/com/example/api/
```

__2. We will take a look at these files one at a time:

__3. Open **TokenRequestData.java** in a text editor.

```
package com.example.api;

public class TokenRequestData {
  String username;
  String password;
  String scopes;

  public TokenRequestData(String username, String password, String
scopes) {
    super();
    this.username = username;
    this.password = password;
    this.scopes = scopes;
  }

  public String getUsername() {
    return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(String password) {
    this.password = password;
  }
  public String getScopes() {
    return scopes;
  }

  public void setScopes(String scopes) {
    this.scopes = scopes;
  }

}
```

TokenRequestData.java is a standard data object having properties for, username, password and scopes. These are the values we are going to send to the Token service we are creating. The Token service will check the username and password and then create a JWT token including the scopes. Getters and setters exist for each of the three properties. Classes structured like this one are typically refered to as Java Beans.

__4. Open **Authenticator.java** in a text editor.

```
package com.example.api;

public class Authenticator {

  public static boolean checkUser(String username) {
    if( (username != null && username.length() > 0) &&
      ( username.equalsIgnoreCase("john")
        || username.equalsIgnoreCase("susan"))) {
      return true;
    }else {
      return false;
    }
  }

  public static boolean checkPassword(String username, String password)
{
    if(checkUser(username)) {
      if(username.equalsIgnoreCase("john") && password.equals("pass")) {
        return true;
      }
      if(username.equalsIgnoreCase("susan") && password.equals("pass"))
{
        return true;
      }
    }else {
      return false;
    }
    return false;
  }

}
```

The Authenticator class has two methods; checkUser and checkPassword. The checkPassword method will be called with the username and password credentials passed into the Token service. This implementation allows the following two possible users:

Username: john, Password: pass

Username: susan, Password: pass

This is good enough for our purposes. In a real world implementation a class such as this would be expected to call another service such as an LDAP server to verify the username and password credentials.

__5. Open **TokenAPI.java** in a text editor.

```
package com.example.api;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/token")
public class TokenAPI {
  JWTUtil jwtUtil = new JWTMockUtil();
  // JWTUtil jwtUtil = new JWTHelper();

  @PostMapping(consumes = "application/json")
  public ResponseEntity<?> getToken(@RequestBody
    TokenRequestData tokenRequestData) {

    String username = tokenRequestData.getUsername();
    String password = tokenRequestData.getPassword();
    String scopes = tokenRequestData.getScopes();

    if (username != null && username.length() > 0
        && password != null && password.length() > 0
        && Authenticator.checkPassword(username, password)) {
      Token token = jwtUtil.createToken(scopes);
      ResponseEntity<?> response = ResponseEntity.ok(token);
      return response;
    }
    // bad request
    return (ResponseEntity<?>) ResponseEntity
    .status(HttpStatus.UNAUTHORIZED).build();

  }

}
```

The TokenAPI class implements an API service that accepts username, password and scopes and returns a JWT token. Take a minute to read through the code. The endpoint is clear from the @RequestMapping annotation that appears before the class signature:

```
@RestController
@RequestMapping("/token")
public class TokenAPI { … }
```

The class contains one method – getToken. The getToken method accepts JSON data which is converted to a TokenRequestData object and then passed into the method:

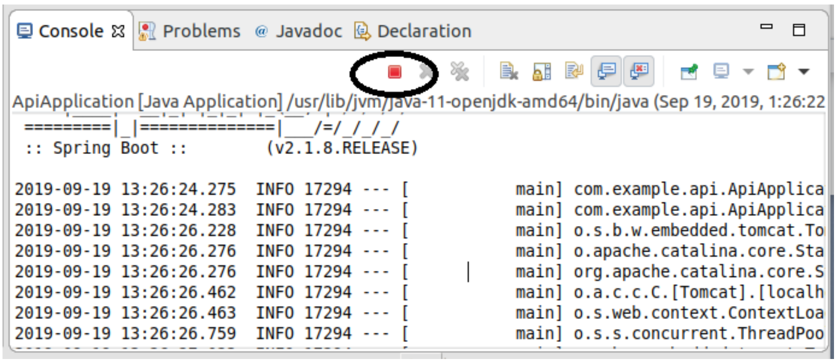
```
@PostMapping(consumes = "application/json")
public ResponseEntity<?> getToken(@RequestBody
           TokenRequestData tokenRequestData) { … }
```

The method retrieves credentials from the TokenRequestData object, and passes them to Authenticator.checkPassword which returns true for valid users and passwords. For requests with valid credentials the method calls the jwtUtil.createToken method which takes scopes as a parameter and returns a token. The token is then returned to the user who made the request. If the credentials fail validation an error message is returned instead.

Next we will test the Token service.

__6. Close the editor windows.

__7. Shut down the existing instance of the api server by clicking on the red 'Terminate' box on the 'Console' tab.



```
ApiApplication [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Sep 19, 2019, 1:26:22
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.8.RELEASE)

2019-09-19 13:26:24.275  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:24.283  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:26.228  INFO 17294 --- [           main] o.s.b.w.embedded.tomcat.To
2019-09-19 13:26:26.276  INFO 17294 --- [           main] o.apache.catalina.core.Sta
2019-09-19 13:26:26.276  INFO 17294 --- [           main] org.apache.catalina.core.S
2019-09-19 13:26:26.462  INFO 17294 --- [           main] o.a.c.c.C.[Tomcat].[localh
2019-09-19 13:26:26.463  INFO 17294 --- [           main] o.s.web.context.ContextLoa
2019-09-19 13:26:26.759  INFO 17294 --- [           main] o.s.s.concurrent.ThreadPoo
```

__8. Right click on the project in the 'Package Explorer” and choose **Run-As/Maven Install** to retrieve dependencies and compile the project.

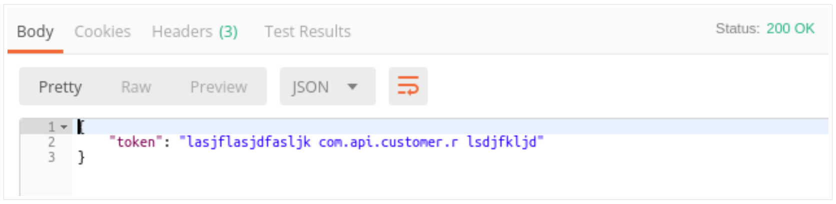
__9. Right-Click on '**ApiApplication.java**' and choose **Run-As/Run As Java Application**.

__10. Open the Postman REST client and execute a POST request against the Token endpoint:

```
Type: POST
URL: localhost:8080/token
BODY:
{"username": "john","password":"pass", "scopes":"com.api.customer.r"}
```

Make sure the body is in JSON format as shown above.

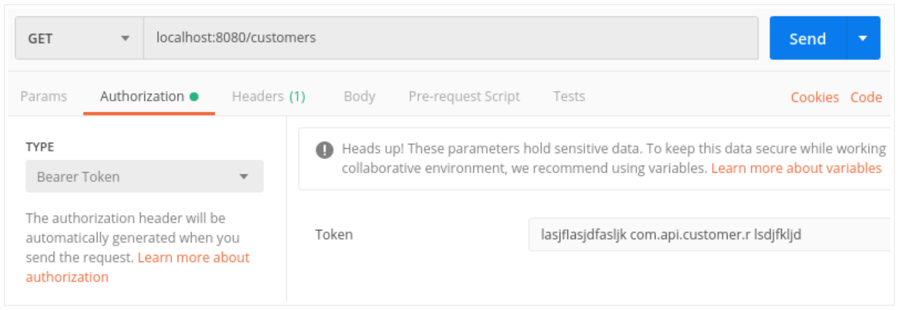
__11. You should get the following response:



```
{
    "token": "lasjflasjdfasljk com.api.customer.r lsdjfkljd"
}
```

The returned token string was generated by our mock JWTUtil class. It is longer than 10 characters so it should allows us to retrieve customer data. (At least for now)

__12. Update the authentication token on the Customer request with the one that was just returned:



The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working collaborative environment, we recommend using variables. Learn more about variables

Token: lasjflasjdfasljk com.api.customer.r lsdjfkljd

__13. Click Send to execute the Customer API request:

```
Type: GET
URL: localhost:8080/customers
```

__14. You should get Customer data and no error.

Part 4 - Update the JWT Utility to Manage Real Tokens

We now have a token service that we can call to get tokens and we have a Customer data service that is protected by a Servlet filter that checks for the token. But the JWTUtil implmentation we have been using does not create real JWT tokens, only mock tokens. To fix this we will add a real implementation that uses the **com.auth0.jwt.JWT** library to create and manage real JWT tokens.

__1. Copy the following file from LabFiles into the project:

```
Copy Files:
    JWTHelper.java

From Dir: LabFiles/jwt-lab/
To Dir: LabWork/jwt-lab/src/main/java/com/example/api/
```

__2. Open JWTHelper.java in a text editor:

```
package com.example.api;

import java.util.Date;
import java.util.Map;

import com.auth0.jwt.JWT;
import com.auth0.jwt.JWTVerifier;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.exceptions.JWTCreationException;
import com.auth0.jwt.exceptions.JWTVerificationException;
import com.auth0.jwt.interfaces.Claim;
import com.auth0.jwt.interfaces.DecodedJWT;

public class JWTHelper implements JWTUtil {

    @Override
    public Token createToken(String scopes) {

      try {
          Algorithm algorithm = Algorithm.HMAC256("secret");
          long fiveHoursInMillis = 1000 * 60 * 60 * 5;
          Date expireDate =
            new Date(System.currentTimeMillis() + fiveHoursInMillis);
          String token = JWT.create()
            .withSubject("apiuser")
              .withIssuer("me@me.com")
              .withClaim("scopes", scopes)
              .withExpiresAt(expireDate)
              .sign(algorithm);
          return new Token(token);
      } catch (JWTCreationException exception){
        return null;
      }
    }

    @Override
    public boolean verifyToken(String token) {

      try {
          Algorithm algorithm = Algorithm.HMAC256("secret");
          JWTVerifier verifier = JWT.require(algorithm)
```

```
                .withIssuer("me@me.com")
                .build();
            DecodedJWT jwt = verifier.verify(token);
            return true;
        } catch (JWTVerificationException exception){
          return false;
        }

    }

    public Map<String, Claim> getClaims(String token) {
        try {
            Algorithm algorithm = Algorithm.HMAC256("secret");
            JWTVerifier verifier = JWT.require(algorithm)
                .withIssuer("me@me.com")
                .build(); //Reusable verifier instance
            DecodedJWT jwt = verifier.verify(token);
            return jwt.getClaims();
        } catch (JWTVerificationException exception){
          return null;
        }
    }

    @Override
    public String getScopes(String token) {
        try {
            Algorithm algorithm = Algorithm.HMAC256("secret");
            JWTVerifier verifier = JWT.require(algorithm)
                .withIssuer("me@me.com")
                .build(); //Reusable verifier instance
            DecodedJWT jwt = verifier.verify(token);
            return jwt.getClaim("scopes").asString();
        } catch (JWTVerificationException exception){
          return null;
        }
    }
  }
```

The JWTHelper class implements the JWTUtil interface:

```
public class JWTHelper implements JWTUtil { … }
```

And it includes the interface methods:

```
public Token createToken(String scopes) {...}
public boolean verifyToken(String token) {...}
public String getScopes(String token) {…}
```

We will take a look at these one at a time.

__3. The **createToken** Method accepts a scopes string and returns a Token object:

```
public Token createToken(String scopes) {
  try {
      Algorithm algorithm = Algorithm.HMAC256("secret");
      long fiveHoursInMillis = 1000 * 60 * 60 * 5;
      Date expireDate =
              new Date(System.currentTimeMillis() + fiveHoursInMillis);
      String token = JWT.create()
        .withSubject("apiuser")
          .withIssuer("me@me.com")
          .withClaim("scopes", scopes)
          .withExpiresAt(expireDate)
          .sign(algorithm);
      return new Token(token);
  } catch (JWTCreationException exception){
    return null;
  }
}
```

The method calculates an expiration date for the token and then uses the com.auth0.jwt.JWT.create() builder to assemble and sign the token. JWT.create creates a String which is converted to a Token object which is then returned to the caller.

__4. The **verifyToken** Method accepts a token String and returns true or false depending if the token is valid:

```
public boolean verifyToken(String token) {
  try {
      Algorithm algorithm = Algorithm.HMAC256("secret");
      JWTVerifier verifier = JWT.require(algorithm)
          .withIssuer("me@me.com")
          .build();
      DecodedJWT jwt = verifier.verify(token);
      return true;
  } catch (JWTVerificationException exception){
    return false;
  }
}
```

The verifyToken method creates an instance JWTVerifier and calls its 'verify' method with the token String. When successful the 'verify' method returns a DecodedJWT object and *true* is returned from verifyToken. If 'verify' is not successful an exception is thrown and *false* is returned from verifyToken.

__5. The **getScopes** Method accepts a token String and returns the scopes String that was embedded in the token:

```
public String getScopes(String token) {
  try {
      Algorithm algorithm = Algorithm.HMAC256("secret");
      JWTVerifier verifier = JWT.require(algorithm)
          .withIssuer("me@me.com")
          .build(); //Reusable verifier instance
      DecodedJWT jwt = verifier.verify(token);
      return jwt.getClaim("scopes").asString();
  } catch (JWTVerificationException exception){
    return null;
  }
}
```

The **getScopes** method is similar to the previous method in that it creates a DecodedJWT object. But then it calls the 'getClaims' method of the DecodedJWT object to return a value for the “scopes” claim. If anything goes wrong getScopes returns null instead.

In order to use JWTHelper we just need to update a few lines in the TokenAPI and AuthFilter classes.

__6. Open **AuthFilter.java** in an editor.

__7. The following lines should appear near the top of the file:

```
JWTUtil jwtUtil = new JWTMockUtil();
// JWTUtil jwtUtil = new JWTHelper();
```

__8. Comment out the first line and uncomment the second line so that they look like this:

```
// JWTUtil jwtUtil = new JWTMockUtil();
JWTUtil jwtUtil = new JWTHelper();
```

__9. Save the AuthFilter.java file

Now we will make the same change in the TokenAPI.java file.

__10. Open **TokenAPI.java** in an editor.

__11. The following lines should appear near the top of the file:

```
JWTUtil jwtUtil = new JWTMockUtil();
// JWTUtil jwtUtil = new JWTHelper();
```

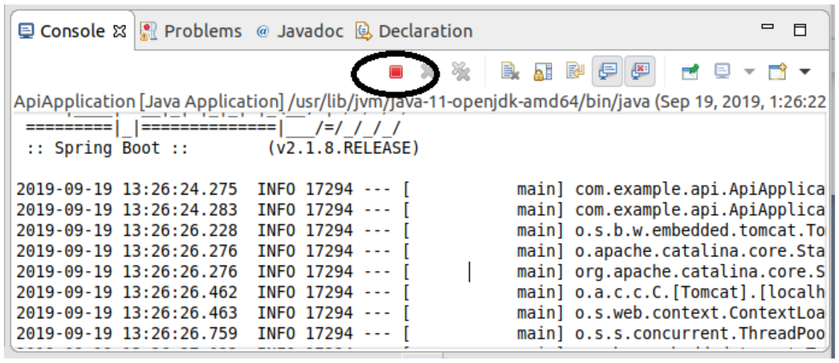
__12. Comment out the first line and uncomment the second line so that they look like this:

```
// JWTUtil jwtUtil = new JWTMockUtil();
JWTUtil jwtUtil = new JWTHelper();
```

__13. Save the TokenAPI.java file

__14. Close the editor windows.

__15. Shut down the existing instance of the api server by clicking on the red 'Terminate' box on the 'Console' tab.



```
ApiApplication [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Sep 19, 2019, 1:26:22
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.8.RELEASE)

2019-09-19 13:26:24.275  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:24.283  INFO 17294 --- [           main] com.example.api.ApiApplica
2019-09-19 13:26:26.228  INFO 17294 --- [           main] o.s.b.w.embedded.tomcat.To
2019-09-19 13:26:26.276  INFO 17294 --- [           main] o.apache.catalina.core.Sta
2019-09-19 13:26:26.276  INFO 17294 --- [           main] org.apache.catalina.core.S
2019-09-19 13:26:26.462  INFO 17294 --- [           main] o.a.c.c.C.[Tomcat].[localh
2019-09-19 13:26:26.463  INFO 17294 --- [           main] o.s.web.context.ContextLoa
2019-09-19 13:26:26.759  INFO 17294 --- [           main] o.s.s.concurrent.ThreadPoo
```

Part 5 - Test the JWT Protected API

In this part we will call the Token service to obtain a JWT token and use the token to make Customer API requests. We'll start by running the APIs.

__1. Right click on the project in the 'Package Explorer” and choose **Run-As/Maven Install** to retrieve dependencies and compile the project.

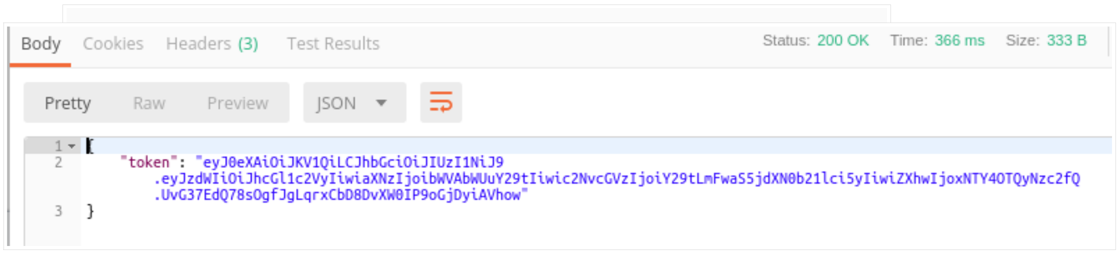
__2. Right-Click on '**ApiApplication.java**' and choose **Run-As/Run As Java Application**.

__3. Open the Postman REST client and execute a POST request against the Token endpoint:

```
  Type: POST
  URL: localhost:8080/token
  BODY:
  {"username": "john","password":"pass", "scopes":"com.api.customer.r"}
```

Make sure the body is in JSON format as shown above.

__4. You should get the following response:

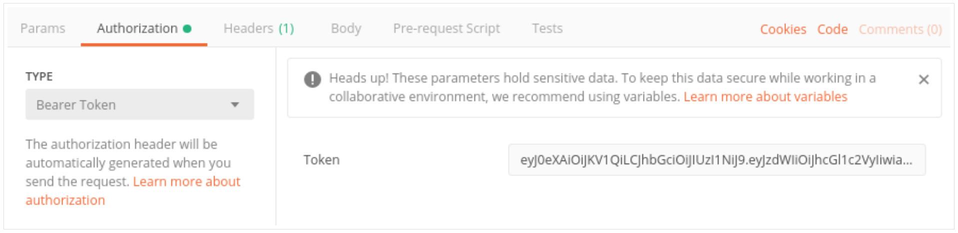


```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
        .eyJzdWIiOiJhcGl1c2VyIiwiaXNzIjoibWVAbWUuY29tIiwic2NvcGVzIjoiY29tLmFwaS5jdXN0b21lci5yIiwiZXhwIjoxNTY4OTQyNzc2fQ
        .UvG37EdQ78sOgfJgLqrxCbD8DvXW0IP9oGjDyiAVhow"
}
```

The token displayed in your response will differ from the one above but should look similar.

This time the returned token string was generated by the real JWT token implementation class JWTHelper. The token has three parts separated by two '.' characters and is base64 encoded.

__5. Update the authentication token on the Customer request with the one that was just returned:



Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

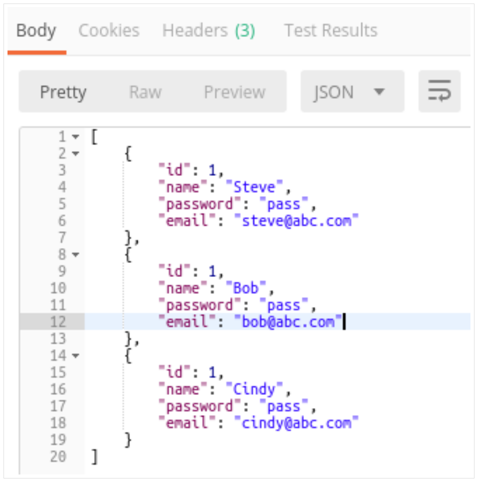
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhcGl1c2VyIiwia...

__6. Click Send to execute the Customer API request:

```
Type: GET
URL: localhost:8080/customers
```

__7. You should get Customer data and no error:



```
[
    {
        "id": 1,
        "name": "Steve",
        "password": "pass",
        "email": "steve@abc.com"
    },
    {
        "id": 1,
        "name": "Bob",
        "password": "pass",
        "email": "bob@abc.com"
    },
    {
        "id": 1,
        "name": "Cindy",
        "password": "pass",
        "email": "cindy@abc.com"
    }
]
```

Now that the services are working, you can try some additional tests.

__8. Try creating a token with a different scopes string and using it to call the Customer API. You should see that the Customer API returns an error because the scope in the token no longer matches the one in the Customer API.

__9. Try creating a token with a different usernames or passwords. You should see that the Token API returns an error because of the invalid users or passwords.

Part 6 - Review

In this lab we followed the steps listed below to create Customer and Token web services (APIs), obtain tokens from the Token API and use them to access the Customer API.

1. Installing and Running the Starter Application
2. Add a Servlet Filter that Checks the Authorization Header
3. Create an API Service for Retrieving JWT Tokens
4. Update the JWT Utility to Manage Real Tokens
5. Test the JWT Protected API