How to Add Two-Factor Authentication (2FA) Using Java and Spark

Last updated on May 11, 2021

Two-factor authentication (2FA) is an increasingly popular method of protecting against fraudulent signups. It works by adding an extra layer of security that requires not only the traditional username and password but also something that the user has in their possession. That's typically their mobile phone.

In this tutorial, you will learn how to implement 2FA for your web apps and services. We'll do this by creating a simple web page that asks a user to register using their mobile phone number. We'll use the Verify API to generate a confirmation code and send it to the number via SMS.

If the user has access to the number that they registered with, they will receive the code. Our application will then prompt them to enter it to complete the registration process.

Prerequisites

We'll build this application using Java and the Spark web framework and create some simple views for the UI

elements using the Handlebars templating engine.

You'll need the following:

- The JDK or its open-source equivalent OpenJDK. This tutorial was written using OpenJDK 11, but version 8 or above of either should be fine.
- Gradle (version 3.4 or later) to build your project and manage its dependencies.

You can find the source code for this tutorial on GitHub.

Vonage API Account

To complete this tutorial, you will need a Vonage API account. If you don't have one already, you can sign up today and start building with free credit. Once you have an account, you can find your API Key and API Secret at the top of the Vonage API Dashboard.

Create Your Project

Create a directory for your project called `two-factor-auth`, change into that directory and then use `gradle` to

initialize the project:

```
mkdir two-factor-auth
cd two-factor-auth
gradle init --type=java-application
```

Accept all the defaults, then open the generated project in your IDE.

Initialize Dependencies

Locate the `build.gradle` file and replace the `dependencies` section with the following:

```
dependencies {
    // Use JUnit test framework
    testImplementation 'junit:junit:4.12'

    // Spark framework
    implementation 'com.sparkjava:spark-core:2.8.0'

    // Nexmo client library
    implementation 'com.nexmo:client:5.3.0'

    // Templating engine
    implementation 'com.sparkjava:spark-template-handlebars
}
```

We won't write any unit tests in this example, but you can

leave `JUnit` in there for now. However, to stop it complaining about a missing greeting method later on, comment out the test in `src/test/java/two/factor/auth/AppTest.java` as follows:

```
public class AppTest {

}
```

Create a Spark Web Application

`gradle` created the `App` class in the `src/main/java/two/factor/auth/App.java` folder.

Open `App.java` in your IDE. Remove the `getGreeting()` method that `gradle` created for you and add the necessary `import` statements for the `spark` package.

Then, call Spark's `port` method to indicate that your application is listening for requests on port 3000.

Your `App.java` should look like this:

```
package two.factor.auth;

import static spark.Spark.*;

public class App {
```

```
  public static void main(String[] args) {
    port(3000);


  }
}
```

Initialize the Java Client Library

To access the Verify API, you're going to want to use the REST API Client Library for Java.

Instantiate it as shown below, replacing `YOUR_API_KEY` and `YOUR_API_SECRET` with your API key and secret from the developer dashboard. Make sure that you include the necessary `import` statements required to work with the Verify API:

```
package two.factor.auth;

import static spark.Spark.*;

import com.nexmo.client.NexmoClient;
import com.nexmo.client.verify.VerifyResponse;
import com.nexmo.client.verify.VerifyStatus;
import com.nexmo.client.verify.CheckResponse;

public class App {

  static String API_KEY = "YOUR_API_KEY";
```

```
  static String API_SECRET = "YOUR_API_KEY";

  public static void main(String[] args) {
    port(3000);

    NexmoClient client = NexmoClient.builder().apiKey(API_K

  }
}
```

Create the Views

Your application will have three pages:

- An initial **registration page**, where your users will register for your service by entering their mobile number.
- A **confirmation page**, where they will be asked to enter the confirmation code sent to their mobile device by the Verify API.
- A **results page**, where the application will say either that they have registered successfully (if they entered the correct confirmation code) or that registration failed (if they haven't).

Spark supports many different templating engines, which enables you to insert content into your HTML pages dynamically and also re-use blocks of HTML. In this tutorial, we'll use Handlebars.

Because we want to focus on teaching you how to use the Verify API here, we won't describe how these work in this post, but instead ask you to download the content that you need from our GitHub repo.

First, include the following imports in your `App.java` file that will enable you to work with Handlebars:

```
import java.util.HashMap;
import java.util.Map;
import spark.template.handlebars.HandlebarsTemplateEngine;
import spark.ModelAndView;
```

Then, create the `src/main/resources/public` and `src/main/resources/templates` directories.

Copy the contents of the styles.css file into `src/main/resources/public/styles.css`.

Then copy the `*.hbs` template files in the resources folder on GitHub into `src/main/resources/templates`.

Ensure that your application knows about the `styles.css` static CSS file by specifying the location of its parent folder (`public`) in the `main` method in `App.java`:

```
public static void main(String[] args) {
  port(3000);
  staticFiles.location("/public");
```

```
  NexmoClient client = NexmoClient.builder().apiKey(API_KEY
}
```

Display the Initial Registration Page

When your user first visits your site, you want to display the registration page. Do this by defining the default route (`/`) using `spark` and rendering the `register.hbs` template as shown:

```
public static void main(String[] args) {
  port(3000);
  staticFiles.location("/public");

  NexmoClient client = NexmoClient.builder().apiKey(API_KEY

  get("/", (request, response) -> {
    Map<String, Object> model = new HashMap<>();
    return new ModelAndView(model, "register.hbs");
  }, new HandlebarsTemplateEngine());

}
```

Test your application by executing `gradle run` and then visiting `http://localhost:3000` in your browser. If you have set up everything correctly, you will see the following page:

Submit the Verification Request

The user must enter their cellphone number into the text box on the registration page and then click **Register** to start the verification process.

The Verify API expects this number to include the international dialing code but omit any leading zeroes. For example, the UK number `07700 900001` should be represented as `447700900001`.

In a production application, you might want to determine the correct locale and country code programmatically and we have an API for that! Check out the Number Insight API. For now though, let's keep things simple.

When the user clicks **Register** we want to capture the number they entered and submit the verification request.

Each verification request is associated with a verification ID. We need to keep a record of this too so that we can use it to check that the user entered the correct confirmation code later.

So add two class-level variables to store this information, underneath the `API_KEY` and `API_SECRET` variables you populated earlier:

```
static String number = "";
static String requestId = "";
```

We'll submit the verification request from the `/register` route, so define the route as follows:

```
post("/register", (request, response) -> {
  number = request.queryParams("number");

  VerifyResponse verifyResponse = client.getVerifyClient().
  if (verifyResponse.getStatus() == VerifyStatus.OK) {
    requestId = verifyResponse.getRequestId();
    System.out.printf("RequestID: %s", requestId);

  } else {
    System.out.printf("ERROR! %s: %s", verifyResponse.getSt
  }

  Map<String, Object> model = new HashMap<>();
  return new ModelAndView(model, "verify.hbs");
}, new HandlebarsTemplateEngine());
```

This code triggers the verification request by first retrieving an instance of `VerifyClient` and then calling its `verify` method, passing in the number we want to verify and an alphanumeric string that is used to identify the sender in the SMS message body.

It returns a `VerifyResponse` object which we can use to examine if the request was issued successfully. If so, we retrieve the verification request ID and use it to check the code sent to the user for that specific verification attempt

in the next step.

Once we have submitted the verification request, the user will receive a verification code via SMS:

We render the `verify.hbs` view, to allow them to enter the code that they received:

By default, after sending the SMS, the Verify API waits for a code for 125 seconds. If it doesn't receive it within that time period, it follows up with two text-to-speech phone calls before finally giving up and failing the verification attempt. You can find out more about the default workflow and how to enable different workflows by reading the documentation.

Check the Confirmation Code

We now need to provide the necessary logic to verify the code that they entered. Create the `/check` route for this:

```
post("/check", (request, response) -> {
  CheckResponse checkResponse = client.getVerifyClient().ch

  Map<String, Object> model = new HashMap<>();

  if (checkResponse.getStatus() == VerifyStatus.OK) {
    model.put("status", "Registration Successful");
    System.out.println("Verification successful");
  } else {
```

```
    model.put("status", "Verification Failed");
    System.out.println("Verification failed: " + checkRespo
  }
  model.put("number", number);

  return new ModelAndView(model, "result.hbs");
}, new HandlebarsTemplateEngine());
```

This code uses the `VerifyClient.check` method, passing it the request ID that we stored from the verification request step and the code that the user entered in the `verify.hbs` view.

The `check` method returns a `CheckResponse` object. We use its `getStatus` method to determine whether the user entered the correct code and display the appropriate response in the `result.hbs` view. If the user entered the code correctly, we receive the following message:

Try it Out!

1. Execute **`gradle run`** in your terminal.
2. Visit **`http://localhost:3000`** in your browser.
3. Enter your mobile phone number and click **Register**. In a moment or two, you will receive an SMS that contains a verification code.
4. Enter the verification code and click **Check**.
5. If you entered the code successfully, you will receive a "Registration successful" message.

Conclusion

That's the basics of using the Verify API to implement two-factor authentication in your Java web applications. To learn more, see the links to the documentation provided below.

Further Reading

- General information about the Verify API
- Verify API Documentation
- Verify API reference
- Number Insight API