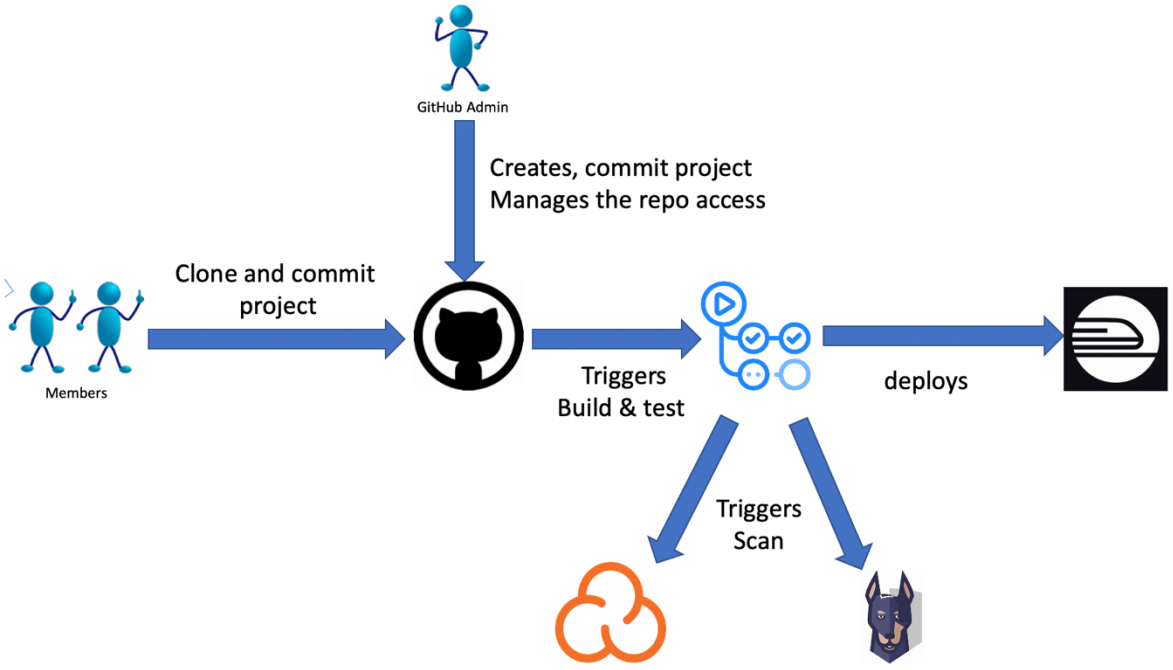
NICF – Essential Practices for Agile Teams

Continuous Integration Workshop with Git Workflow

Table of Contents

Overview

The following diagram sets the context for this workshop.



There are two roles. You may customize it to suit your preference if necessary.

- Member: All the team members play this role.
- GitHub Admin: One member/pair is assigned to administer the GitHub repository. This member/pair also sets up the Unit Testing build/batch file.

Pre-requisite (Estimate: 1.5 hours)

(Prior to EPAT Day 3, you should have the following requirements setup, verified and completed.)

You are required to setup the following prior to walkthrough this CI workshop exercise.

1. Java Developer Kit 17
   https://www.geeksforgeeks.org/download-and-install-java-development-kit-jdk-on-windows-mac-and-linux/

2. Github account
   https://github.com/signup

3. VSCode
   https://code.visualstudio.com/download

4. Postman
   https://www.postman.com/downloads/

5. Snyk account
   https://snyk.io

6. Sonarcloud account
   https://sonarcloud.io/projects

7. Railway account
   https://railway.app

8. Docker
   https://www.appsdeveloperblog.com/how-to-install-docker-on-macos/
   https://docs.docker.com/desktop/install/mac-install/
   https://docs.docker.com/desktop/install/windows-install/

Spring Boot takes the Java Spring platform to a new level that enables developers to build stand-alone, production-ready applications with little effort possible focusing on the business requirements instead of technical configuration and complexity.

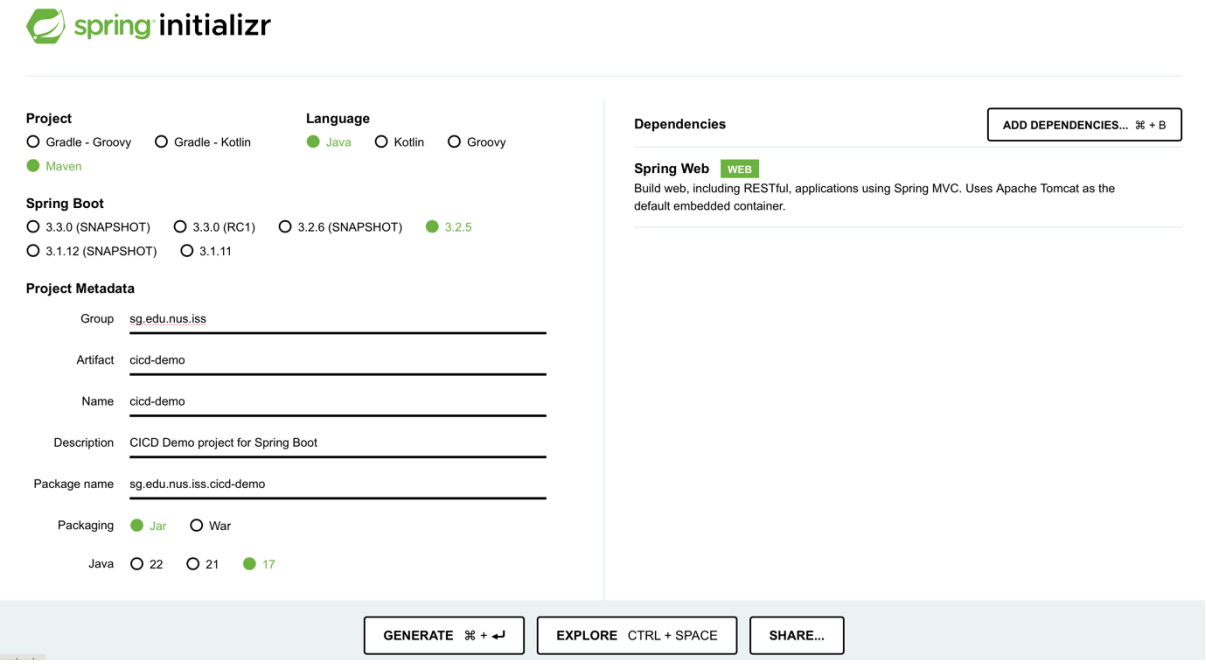
The key driver to the success of Spring boot in Java and Spring framework adoption is the convention over configuration. Instead of the hassle, tedious and length steps of configuring Spring or 3rd-party components, Spring provides out-of-the-box auto configurations that follows common configuration conventions. These are shipped as part of Spring Boot Starters. It has sort of become the de-factor standard for Java and Spring MVC framework to leverage on Spring Boot Web Starter to create Java or Spring applications.

Part One

Spring Initializr (*Estimate: 10 minutes*)

Spring Initializr is the web starter that enables developers to create new Spring Boot development projects. It allows developers to configure certain aspects of the project, such as Maven coordinates or base package, and provides developer with an extensive list of dependencies to choose from.

1. Use Spring Initializer to setup a new Springboot project



Maven Repositories are web servers which provide simple HTTP endpoints which allow GET and PUT requests for publishing and retrieving Maven Artifacts. An artifact can be any type of file that is used in the software development process. The most common of these are Java libraries, also known as 'JAR files'. These are used by Java programs. We will use the Maven repo to help us install a Java library dependency. The repositroy contains a wealth of open source libraries for public usage.

This brings us to Software Engineering 101: "Don't Reinvent the Wheel"- ie. find a reliable library and use it. One of the reasons why Java is popular today is because of the wealth of existing libraries that exist for you to use today.

2. Search for Java Faker library in maven repository from this URL https://mvnrepository.com .

3. Use Maven repo to add the Java Faker library to get random data.

Add the following dependency to the POM file of your Springboot project.

```
        <dependency>
            <groupId>com.github.javafaker</groupId>
            <artifactId>javafaker</artifactId>
            <version>1.0.2</version>
        </dependency>
```

Constructing project code (*Estimate: 30 minutes*)

1. Add a ***Controller folder*** to your springboot project.

2. In the ***Controller folder***, Create a new ***DataController.java*** class file.

3. In the ***DataController.java*** class file, add the following code.

```
package sg.edu.nus.iss.cicddemo.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.github.javafaker.Faker;

@RestController
public class DataController {
    @GetMapping("/")
    public String healthCheck() {
        return "HEALTH CHECK OK!";
    }

    @GetMapping("/version")
    public String version() {
        return "The actual version is 1.0.0";
    }

    @GetMapping("/nations")
    public JsonNode getRandomNations() {
        var objectMapper = new ObjectMapper();
        var faker = new Faker();
        var nations = objectMapper.createArrayNode();
        for (var i = 0; i < 10; i++) {
            var nation = faker.nation();
            nations.add(objectMapper.createObjectNode()
                    .put("nationality", nation.nationality())
                    .put("capitalCity", nation.capitalCity())
                    .put("flag", nation.flag())
                    .put("language", nation.language()));
        }
        return nations;
    }

    @GetMapping("/currencies")
    public JsonNode getRandomCurrencies() {
        var objectMapper = new ObjectMapper();
        var faker = new Faker();
        var currencies = objectMapper.createArrayNode();
        for (var i = 0; i < 20; i++) {
            var currency = faker.currency();
            currencies.add(objectMapper.createObjectNode()
```

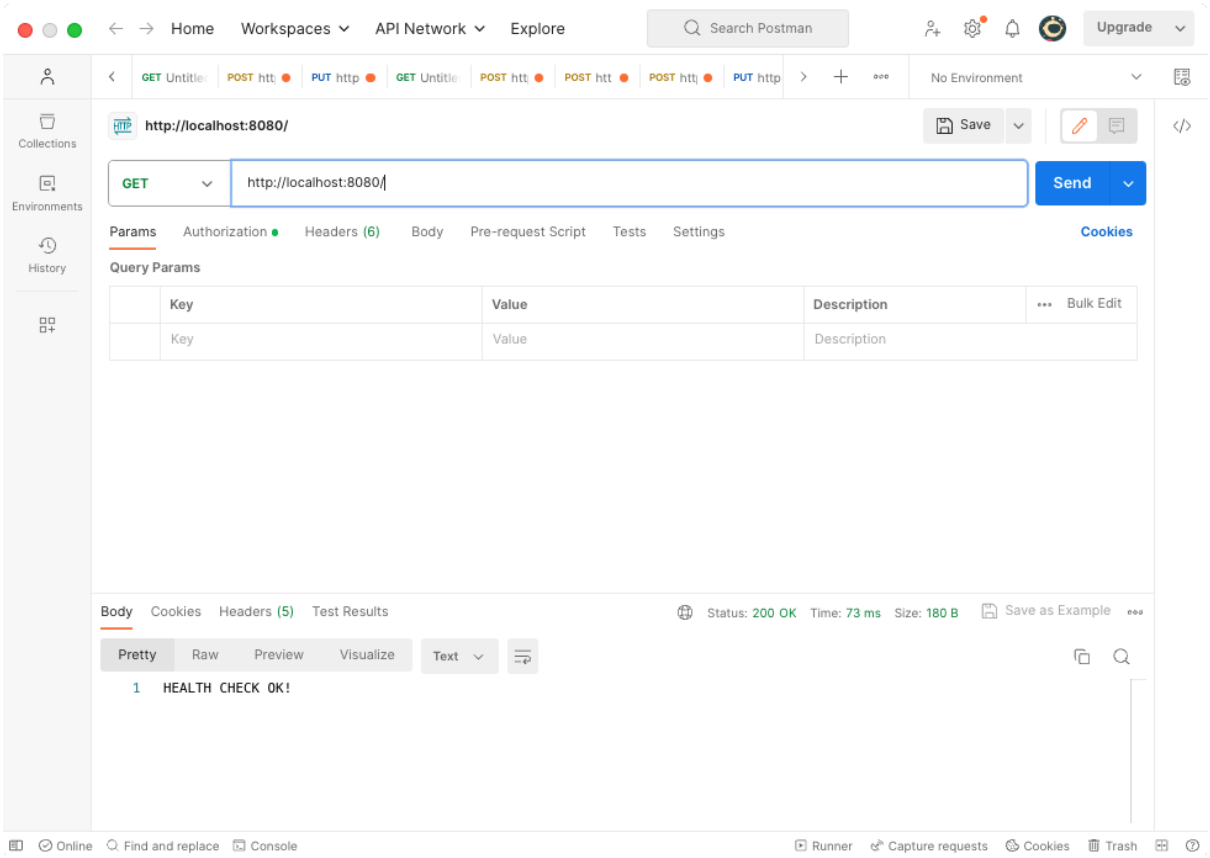
```
                    .put("name", currency.name())
                    .put("code", currency.code()));
        }
        return currencies;
    }
}
```

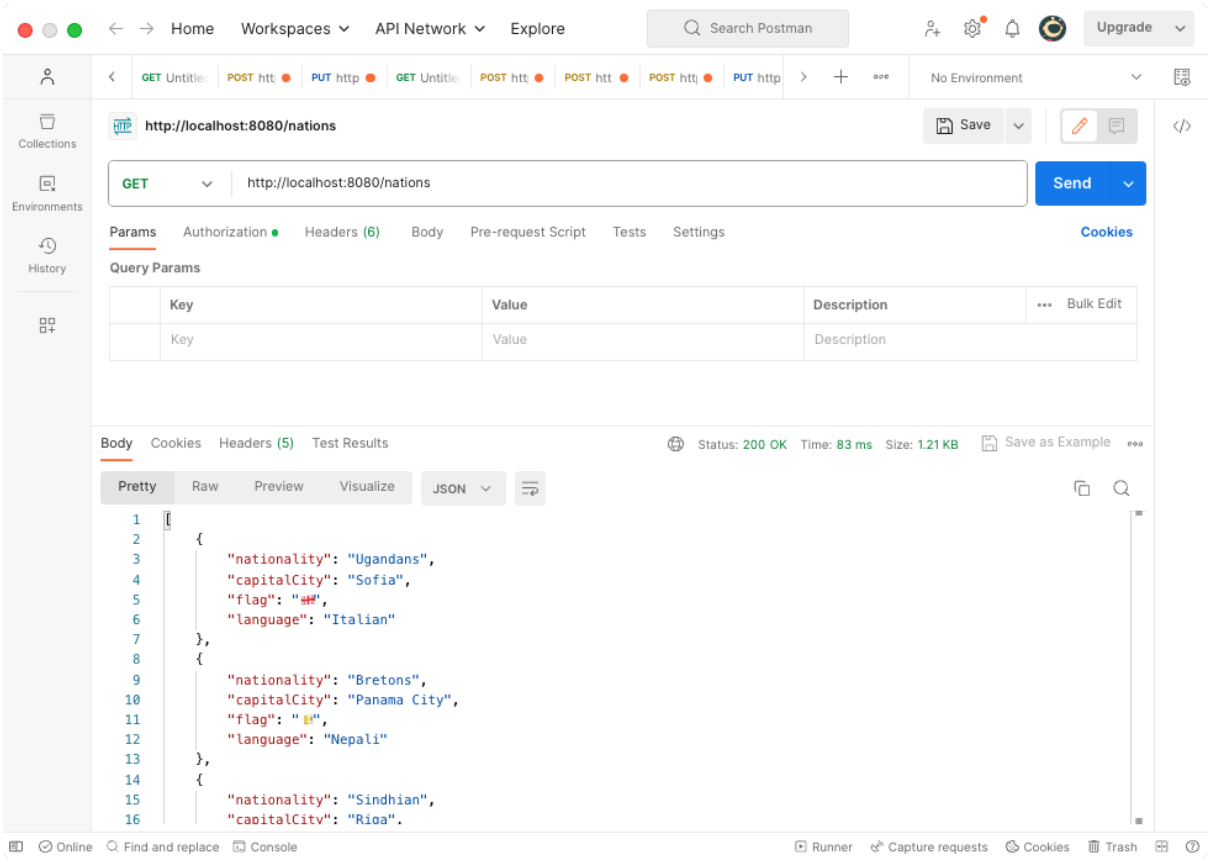
4. Build and run the application using the embedded apache Tomcat server with the command:
   ./mvnw spring-boot:run

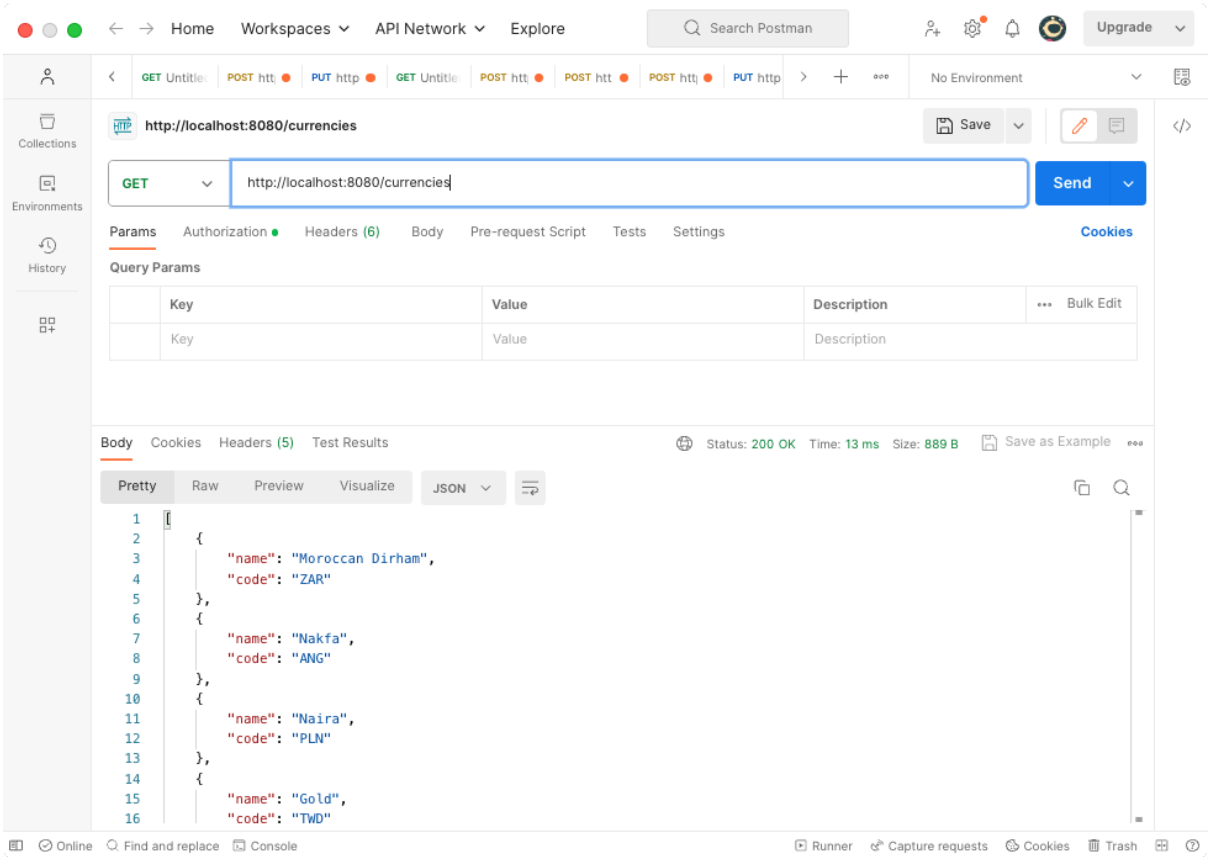
```
2023-08-22T14:22:23.843+08:00  INFO 47348 --- [           main] s.e.n.iss.cicddemo.CicdDemoApplication   : Starting CicdDemoApplication using Java 20.0.1 with PID 47348 (/Users/Darry
l/Downloads/cicd-demo/target/classes started by Darryl in /Users/Darryl/Downloads/cicd-demo)
2023-08-22T14:22:23.844+08:00  INFO 47348 --- [           main] s.e.n.iss.cicddemo.CicdDemoApplication   : No active profile set, falling back to 1 default profile: "default"
2023-08-22T14:22:24.154+08:00  INFO 47348 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-08-22T14:22:24.158+08:00  INFO 47348 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-08-22T14:22:24.158+08:00  INFO 47348 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.11]
2023-08-22T14:22:24.200+08:00  INFO 47348 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-08-22T14:22:24.200+08:00  INFO 47348 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 337 ms
2023-08-22T14:22:24.328+08:00  INFO 47348 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-08-22T14:22:24.333+08:00  INFO 47348 --- [           main] s.e.n.iss.cicddemo.CicdDemoApplication   : Started CicdDemoApplication in 0.616 seconds (process running for 0.731)
```

Testing the APIs (*Estimate: 15 minutes*)

1. Use Postman to test your API application endpoints.




Home
Workspaces
API Network
Explore
Search Postman
Upgrade
No Environment
Collections
Environments
History
http://localhost:8080/nations
Save
GET
http://localhost:8080/nations
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 83 ms
Size: 1.21 KB
Save as Example
Pretty
Raw
Preview
Visualize
JSON
[
{
"nationality": "Ugandans",
"capitalCity": "Sofia",
"flag": "
"language": "Italian"
},
{
"nationality": "Bretons",
"capitalCity": "Panama City",
"flag": "
"language": "Nepali"
},
{
"nationality": "Sindhian",
"capitalCity": "Riga",
Online
Find and replace
Console
Runner
Capture requests
Cookies
Trash


Home
Workspaces
API Network
Explore
Search Postman
Upgrade
No Environment
Collections
Environments
History
http://localhost:8080/currencies
Save
GET
http://localhost:8080/currencies
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 13 ms
Size: 889 B
Save as Example
Pretty
Raw
Preview
Visualize
JSON
[
{
"name": "Moroccan Dirham",
"code": "ZAR"
},
{
"name": "Nakfa",
"code": "ANG"
},
{
"name": "Naira",
"code": "PLN"
},
{
"name": "Gold",
"code": "TWD"
Online
Find and replace
Console
Runner
Capture requests
Cookies
Trash

Adding Unit test to the project (*Estimate: 20 minutes*)

1. In the project test folder, add a new ***DataControllerTest.java*** file.

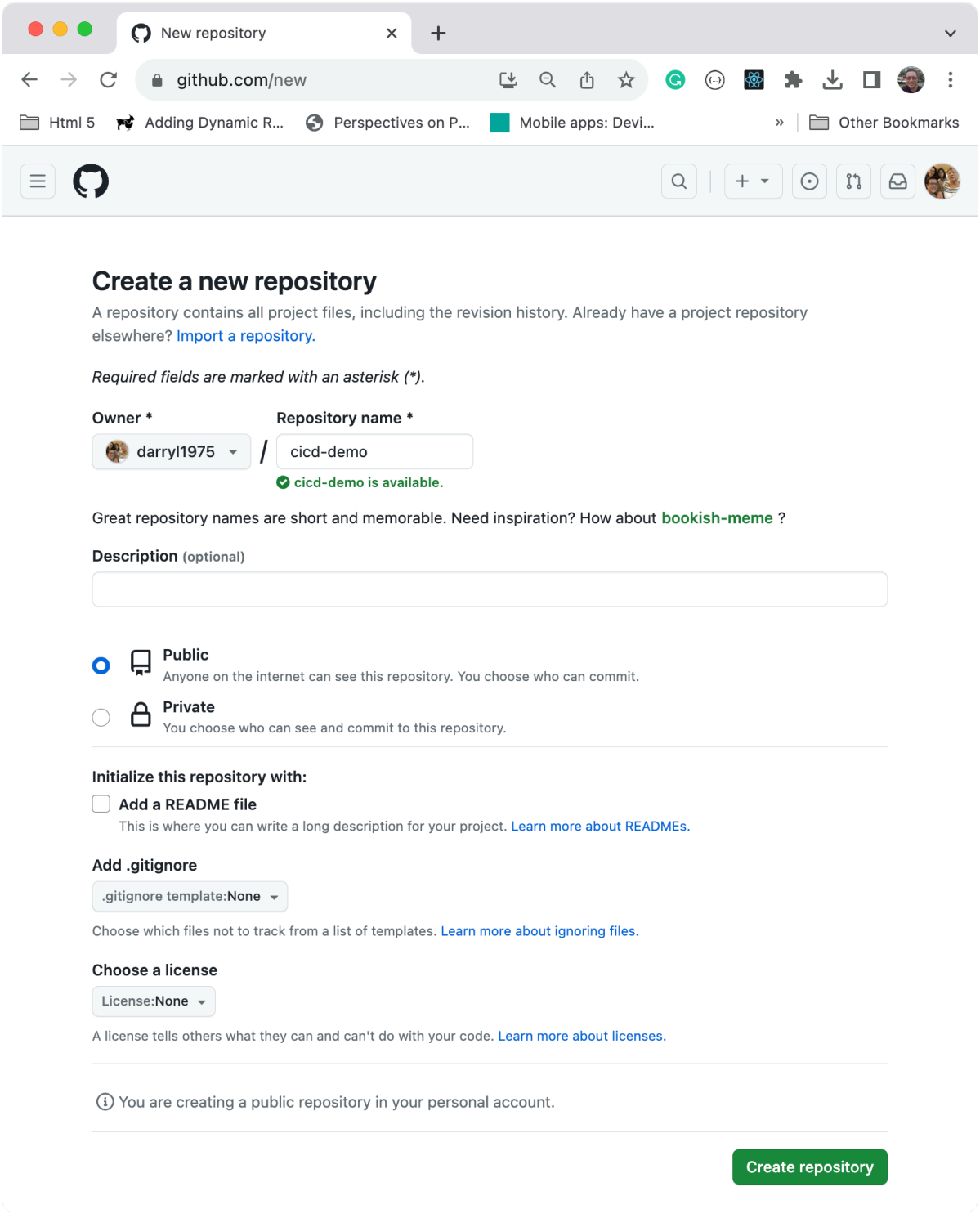
2. In DataControllerTest.java file, add the following Unit Test code below to test our RestController.

```
package sg.edu.nus.iss.cicddemo;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import sg.edu.nus.iss.cicddemo.Controller.DataController;

@SpringBootTest
public class DataControllerTest {
    @Autowired
    DataController dataController;

    @Test
    void health() {
        assertEquals("HEALTH CHECK OK!", dataController.healthCheck());
    }

    @Test
    void version() {
        assertEquals("The actual version is 1.0.0", dataController.version());
    }

    @Test
    void nationsLength() {
        Integer nationsLength = dataController.getRandomNations().size();
        assertEquals(10, nationsLength);
    }

    @Test
    void currenciesLength() {
        Integer currenciesLength = dataController.getRandomCurrencies().size();
        assertEquals(20, currenciesLength);
    }
}
```

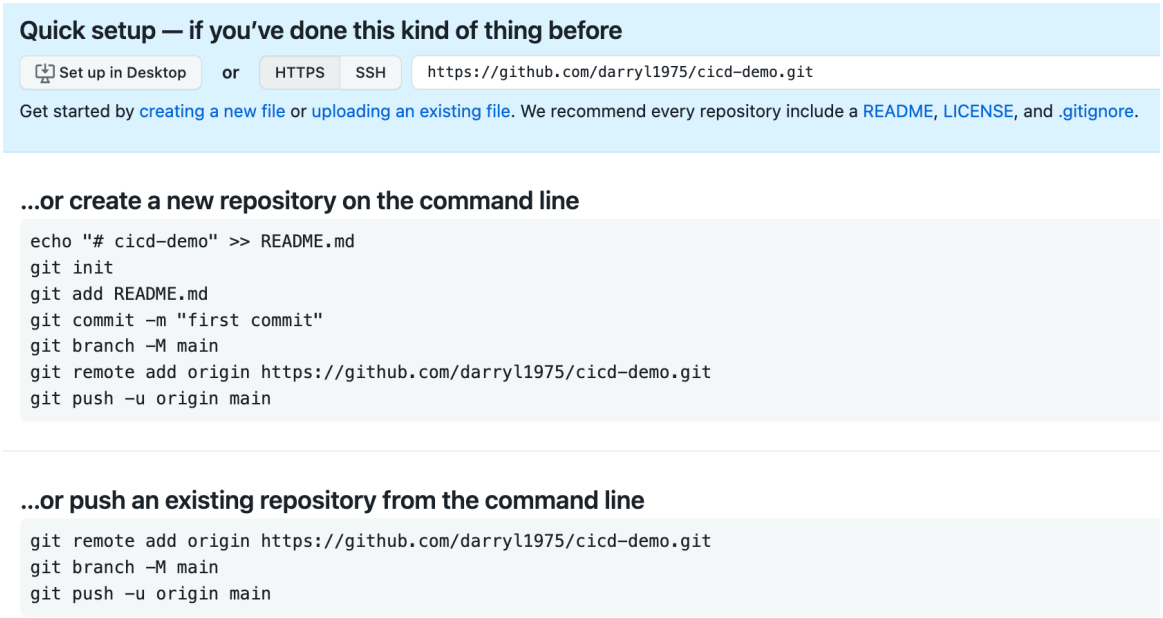
Part Two

Setup project repo (*Estimate: 15 minutes*)

1. Create a new GitHub repo for the project



2. Commit your project to GitHub



Concept of Git Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. It is integrated with Github. Therefore making it easier to use and manage.

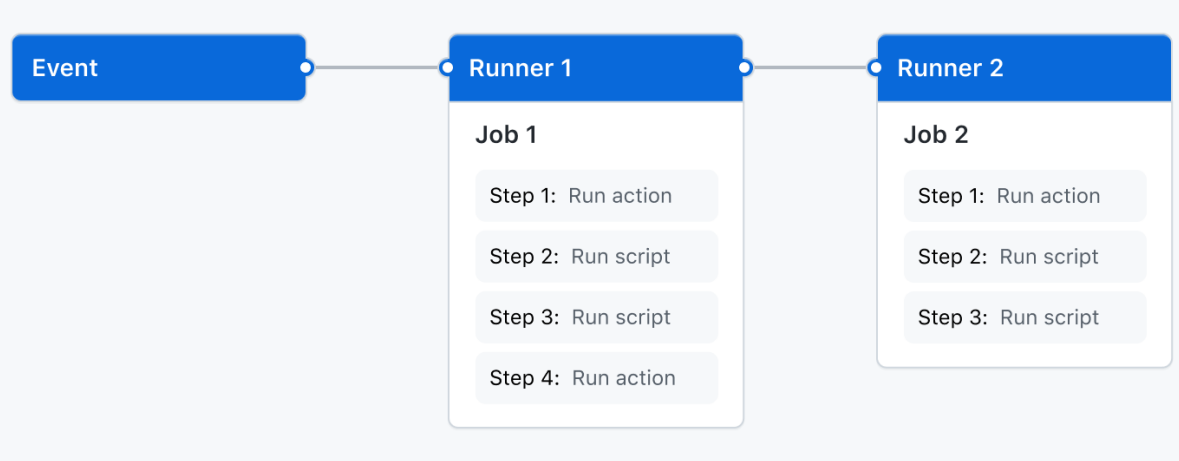
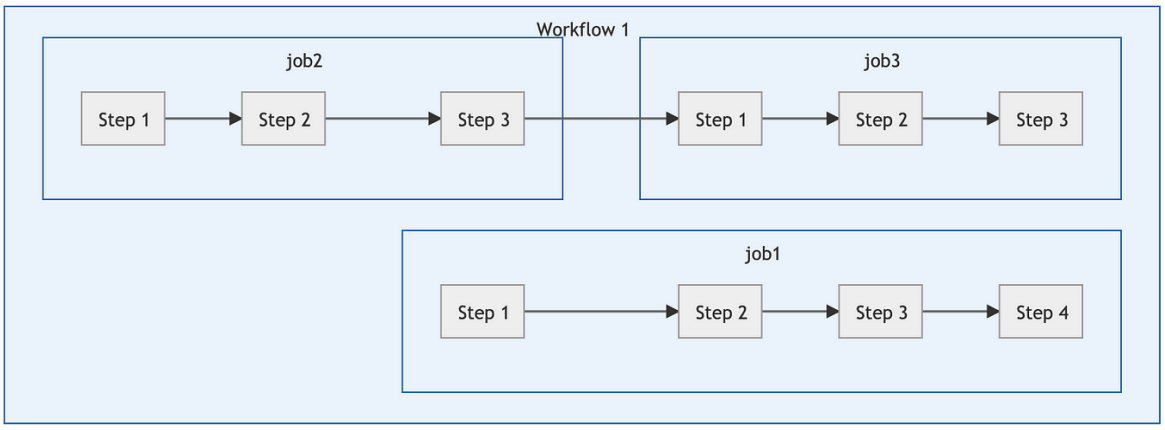


Figure from https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

The above pipeline is a simple description of the CI flow. We also called this workflow on GitHub. To put it simply, a workflow is a collection of job definition executed concurrently as well as sequentially. A job consists of one or more steps which will be executed sequentially as shown.



In the above diagram, we have a workflow named workflow1. It can be triggered when any push event happens. There are 3 jobs in this workflow. And each job hava an arbitrary numbers of steps. So when a developer commits and pushes any code to the repo, job1 and job2 will be fired and run concurrently. Job3 will be triggered only when job2 is completed successfully. If any of the steps failed, the job fails which also make the workflow fails.

To understand how to create the workflow, it follows the following process:

1. ***Create the workflow file into workflows directory***. The file must have a .yml or .yaml extension. For example: ***.github/workflows/main.yml***

2. ***Set workflow name:*** On the first line of your file, start with ***name: Whatever***. This is value must be a string that is unique to your workflow.
3. ***Set event trigger:*** On the third line you'll set the event that will trigger your workflow to run. This is set with the parameter ***on:***
4. ***Set the jobs:*** After the event, you have to set the tasks you want it to run and it's set by the parameter ***jobs:***

Components of a workflow file

name

```
# Each workflow must have a name
# Here the workflow will be named as Sample Workflow
name: Sample workflow
```

on

```
# Each workflow is triggered on some events
# will be triggered on push
on: push
# will be triggered on push and create pull_request
on: [push, pull_request]
# You can also have some configuration like this
on:
  push: # will be triggered for any push
  pull_request:
    types: [opened] # will be triggered when a pull request is created
  issues:
    types: [opened] # will be triggered when a issue is created
  issue_comment:
    types: [deleted] # will be triggered when a comment in a issue is deleted

# You can trigger any workflow using schedule too
on:
  schedule: 0 12 * * * # this will be triggered on 12pm (UTC) each day
# You can filter out branches and tags too
on:
  push:
    tags: # will be triggered when v1.0 or v2.0 is pushed.
      - v1.0
      - v2.0
    branches: # will be triggered when code is pushed to master or any branch
starting with release/
      - master
      - release/*
  pull_request:
    branches: # will be triggered when a pull request is submitted to develop or
master branch
     - develop
     - master
# you can also ignore branches and tags to by simply using branches-ignore instead
of branches
# same goes for tags-ignore in place of tags
# you can not use branches-ignore and branches together and same of tags
on:
```

```
  push:
    branches-ignore:
      - feature/* # will not trigger for any push to feature/ branches
    tags-ignore:
      - v1.* # will not trigger any version like v1.1, v1.5, v1.8
# you can find more in here https://help.github.com/en/actions/reference/events-
that-trigger-workflows#external-events-repository_dispatch

env
# You can have environment variables in the workflow
# this will be available in each job and steps inside it.
env:
  PROJECT_ID: sample-project-id
  USERNAME: sample-username

# So now in each steps you can access them by $PROJECT_ID or $USERNAME
# Normally the environment variables will show their value when printed in logs
# You might want to hide sensitive information in logs like password
# For this use,  secrets.
# You can set secrets under Variables in each repository settings
# Repo --> Settings --> Variables
env:
  PROJECT_ID: sample-project-id
  USERNAME: sample-username
  PASSWORD: ${{secrets.PASSWORD}} # you have to set PASSWORD in variables under
repo settings

# You can have env variables in jobs and steps too.

jobs
# there might be multiple jobs definition under jobs.
# this is the job annotation. It can contain multiple job
# in each job definition runs-on and steps are required.
jobs:
  # name of the job to be shown in GitHub
  first-job:
    # os in which the job will run
    # Possible values are ubuntu-latest, windows-latest, ubuntu-16.04
    # there might be many version.
    runs-on: ubuntu-latest # required
    # job specific env variables
    # this env variables won't be available to second-job
    env:
      ENV_VAR: hello-world
      SECOND_VAR: hello-world
    # I will explain the steps separately. Generally steps are list of objects.
```

```
    steps: # required
      -
  second-job:
    runs-on: windows-latest
    steps:
      -



# One more interesting thing you can do is to run a job in different os
# For this, you need to strategy under each job config
jobs:
  first-job:
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
    # This will allow the job to run in 3 different os
    runs-on: ${{ matrix.os }}
    steps:
      -


# Apart from this you can also run all the steps in a docker container
jobs:
  first-job:
    runs-on: ubuntu-latest
    container:
      # this will run all the steps in this job inside a python:3 docker container
      image: "python:3"
    steps:
     -


# By default all the jobs that are defined under jobs keywords
# Sometime we need to start a job after a job is finished.
# for this we need the needs keyword
jobs:
  # as first-job and fourth-job don't have any needs keywords
  # they will start together
  first-job:
  second-job:
    # only run when first-job is successful
    needs: [first-job]
  third-job:
    # only run when first-job and second-job is successful
    needs: [first-job, second-job]
```

```
  fourth-job:



# also sometimes you need docker container to be started as sidecar for a job to
run
jobs:
  first-job:
    services:
      postgres:
        image: postgres # docker image name
        ports:
          - "5432:5432" #port exposed
      redis:
        image: redis
```

steps

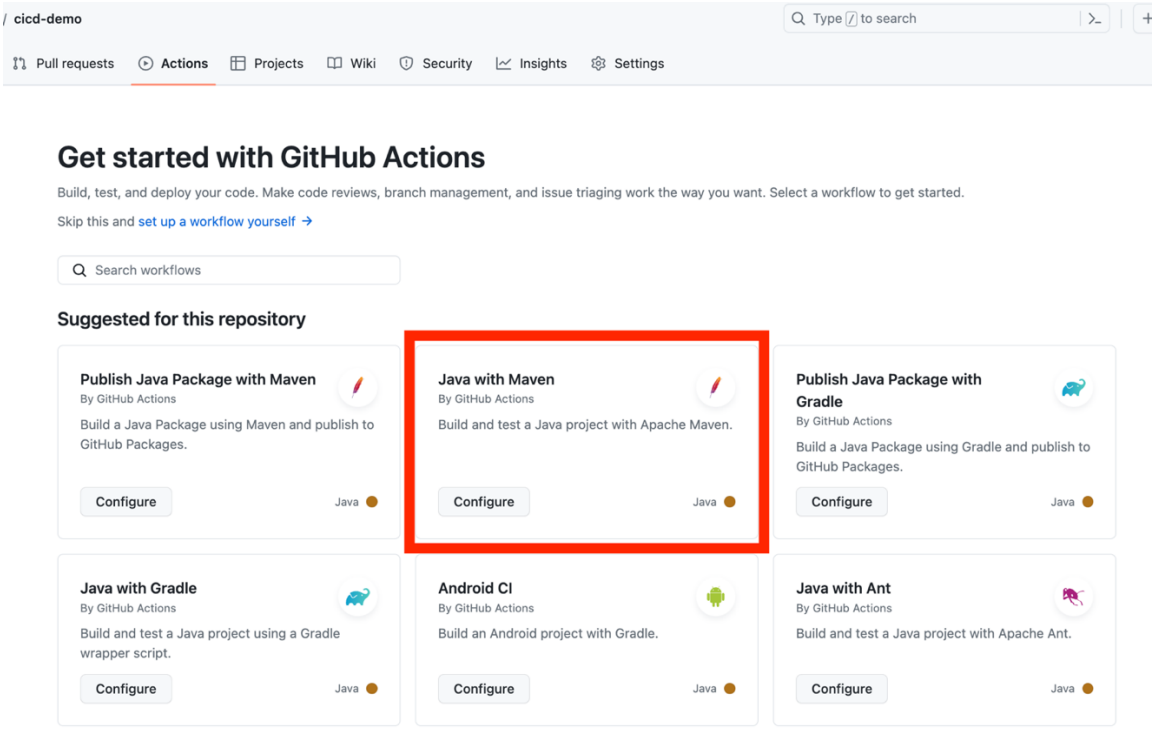
```
# Under each job there is a keyword called steps
# steps contains list of step
# In here will simply explain single step block
# as steps are list, if one fails, the next steps are skipped. There are ways to
trigger it to.
# name of the step
name: name of the step
id: sample_id # needed if this step has output to be used in other steps
run: echo 'Hello World' # command we want to run
# why GitHub Actions are so powerful
# because it has a action marketplace that will let you
# use other's actions and publish yours
name: checkout code at current push
uses: actions/checkout@v1
# here actions is the username, checkout is the repo name
# v1 is the tag
# you can use branch name, sha or tag after @
# but always safe to use tag as it is static
# sometimes some action might take input from outside to act on
# you can pass this using with keywords
name: Running simple steps
uses: actions/hello-world-javascript-action@master
with:
  who-to-greet: 'Mona the Octocat'


## Other than using public actions you can also use your own actions that lives in
your repo
name: Local Action
```

```
uses: ./.github/actions/hello
# for this to succeed make sure you have a file named
# ./.github/actions/hello/action.yaml in your repo
# I will go more into it in my subsequent blogs
# You might want to run a docker container in your steps
# simply use this
name: Running docker container
uses: docker://python:3
# this will bootup a python:3 docker container
# You can have env variables scoped to our steps only
name: Env Variable Steps
run: echo $HELLO_WORLD
env:
  HELLO_WORLD: hello-world # will not be available to other steps
# sometimes you might want to run a step always
name: Run only in case of Failure
run: echo the workflow failed
if: ${{ failed() }} # possible values are failed(), always(), success()
# also you can run a steps based on event type and branches too
name: Run only when push to master
run: echo the code is pushed to master branch
if: ${{ github.event_name == 'push' && github.ref == 'master' }}

# a job might also have output that we can use in subsequent steps too
# let's assume we have a step with id second_step in any step before this step and
it has a output named time.
name: Print Previous Step Output
run: echo ${{steps.second_step.outputs.time}}
```

Create a new Git Workflow for Java project (Estimate: *10 minutes*)

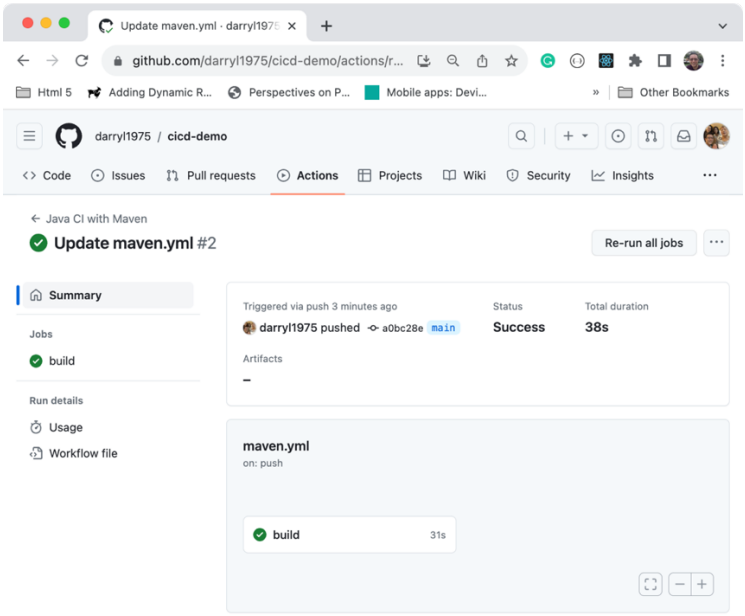
1. Click the “Actions” tab in your repository.

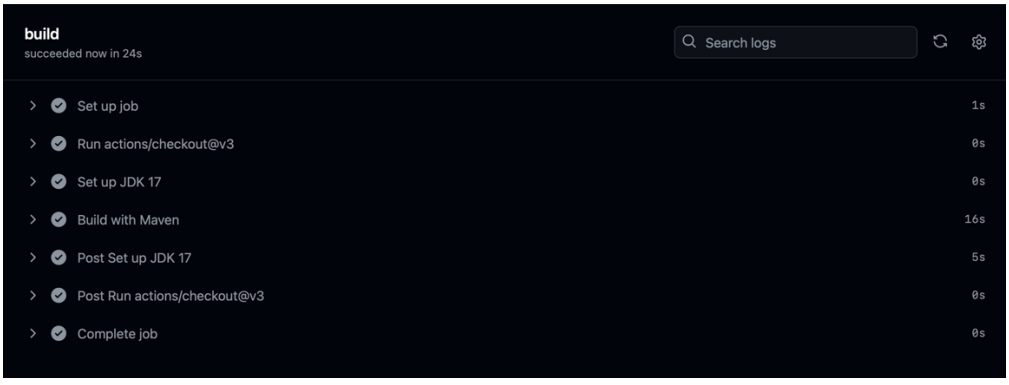
2. Choose the workflow that’s best for your type of project.



3. Analyse the workflow

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 20
      uses: actions/setup-java@v3
      with:
        java-version: '20'
        distribution: 'temurin'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file pom.xml
```





Factor Code Coverage in testing to ensure completeness and code quality (*Estimate: 10 minutes*)

One of the challenges we faced was adding new features or endpoints to existing APIs/microservices and introducing new APIs/microservices for new features without breaking previously released features. This required us to ensure that our code was thoroughly tested before each release. Jacoco plugin for code coverage allow us to measure the effectiveness of our testing efforts by ensuring that the conditional and line coverage was always above 90% during the build phase of our pipeline. It helps you keep an eye on your codebase by measuring how much of it is covered by automated tests. With the Jacoco plugin integrated into your build process, you can be confident that your tests are thorough and effective, and that your code is reliable and performant.

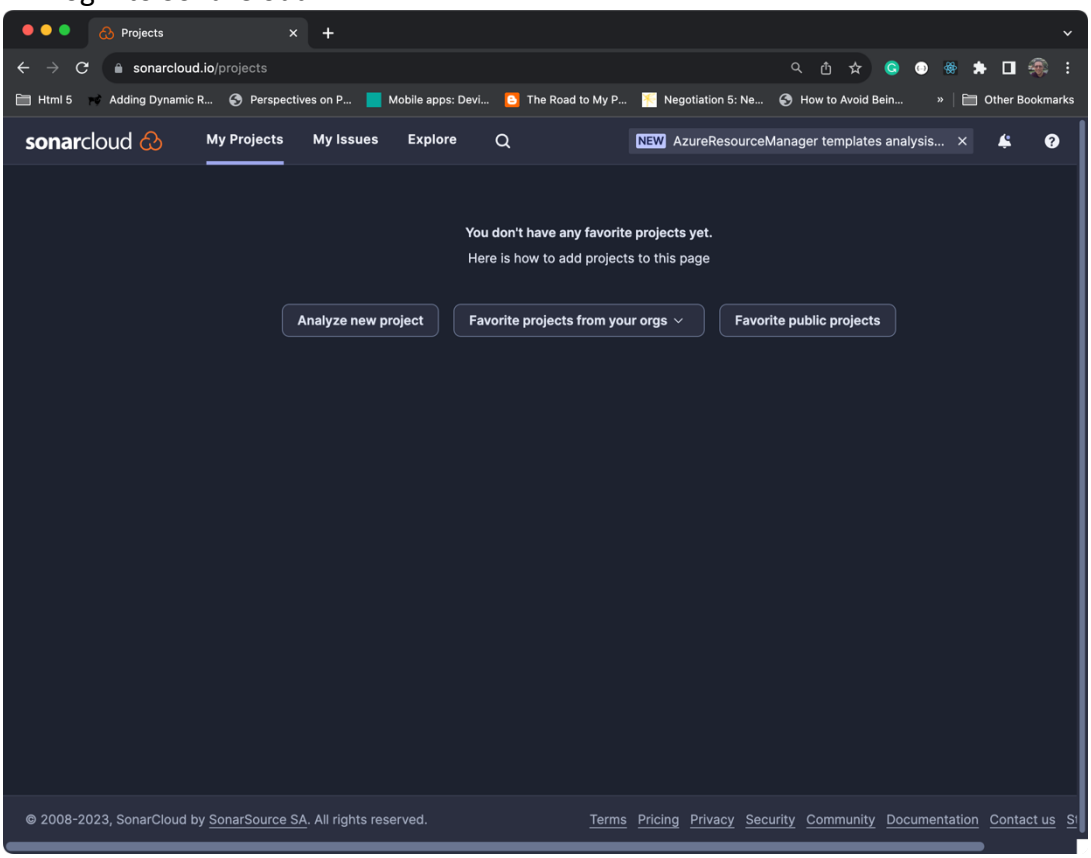
1. Add Jacoco plugin to your Spring Boot project build configuration ***pom.xml*** file.

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.10</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <!-- attached to Maven test phase -->
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

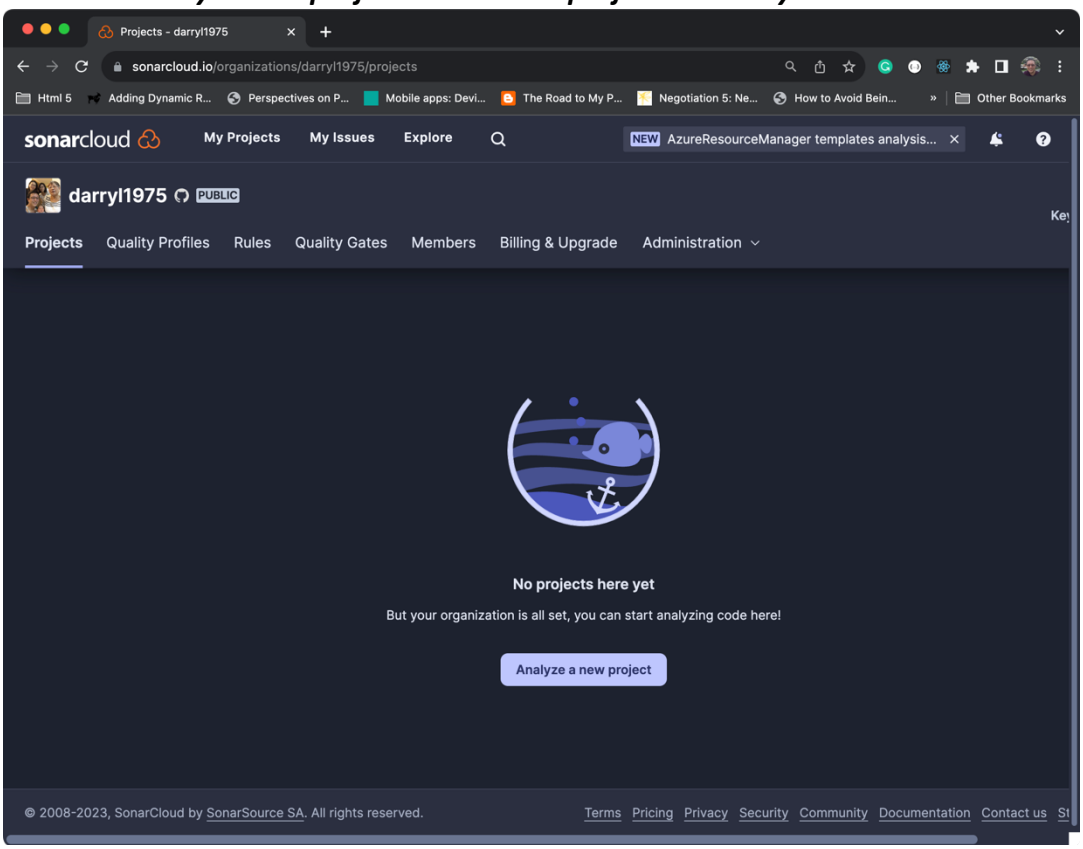
Integrating SonarCloud in your CI pipeline (*Estimate: 1 hour*)

Every CI pipeline is not complete with som form of continuous inspection of code quality. Here, we will integrate SonarQube (open-source platform) for inspection of code quality. Sonarcloud will performs automated code analysis and provides detailed reports on code quality, code coverage, code-related issues, including bugs, vulnerabilities, and code smells. Maintain high code quality and detecting issues early in the software development process ensure timeliness in delivery and mitigates potential technical debts.

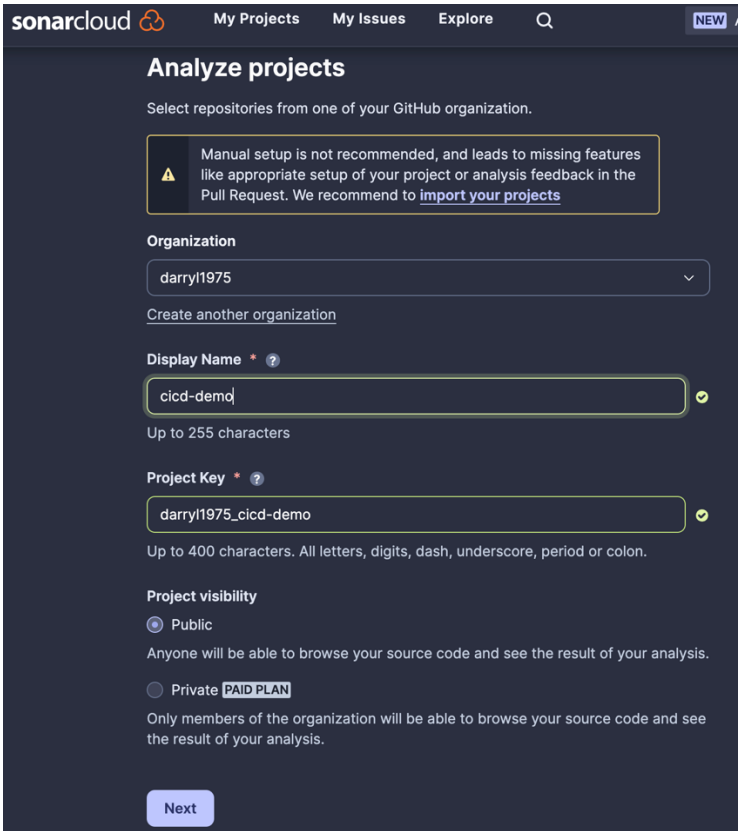
1. Login to SonarCloud



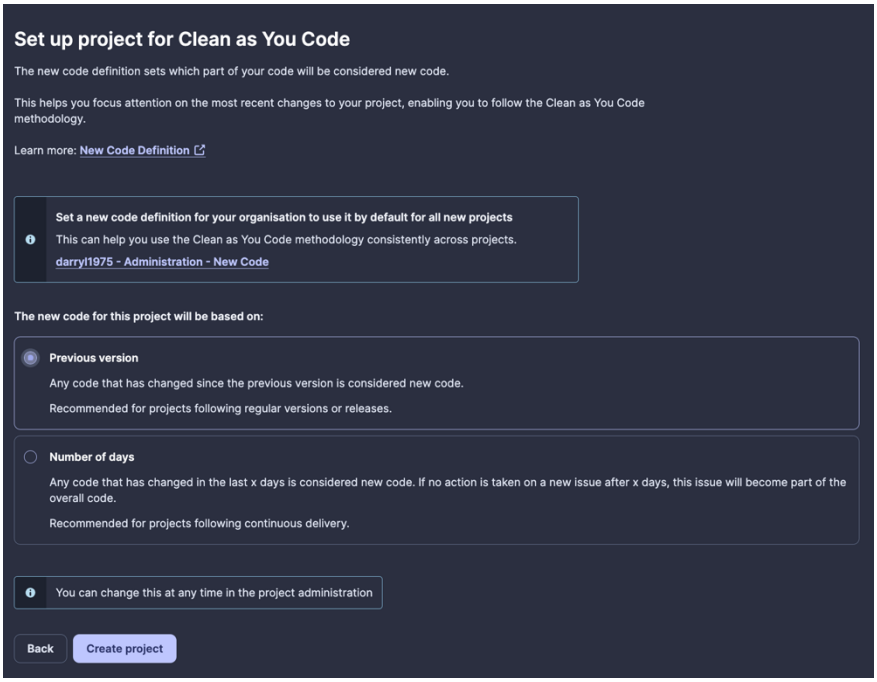
2. Select ***Analyse new project → Create a project manually***.



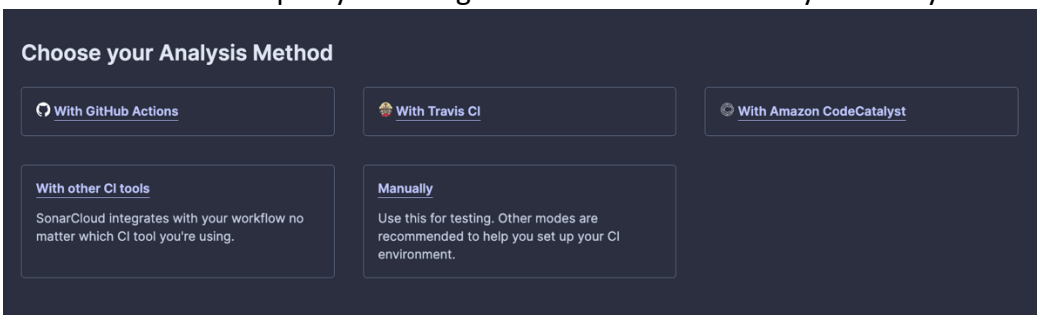
3. Select and fill in the necessary detaisl for the project (similar to the following).



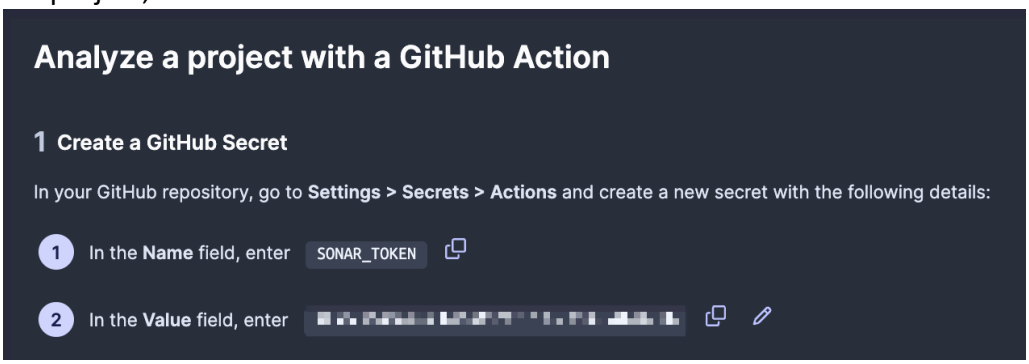
4. Define criteria for new code trigger. Click ***Create project*** to complete the project creation.



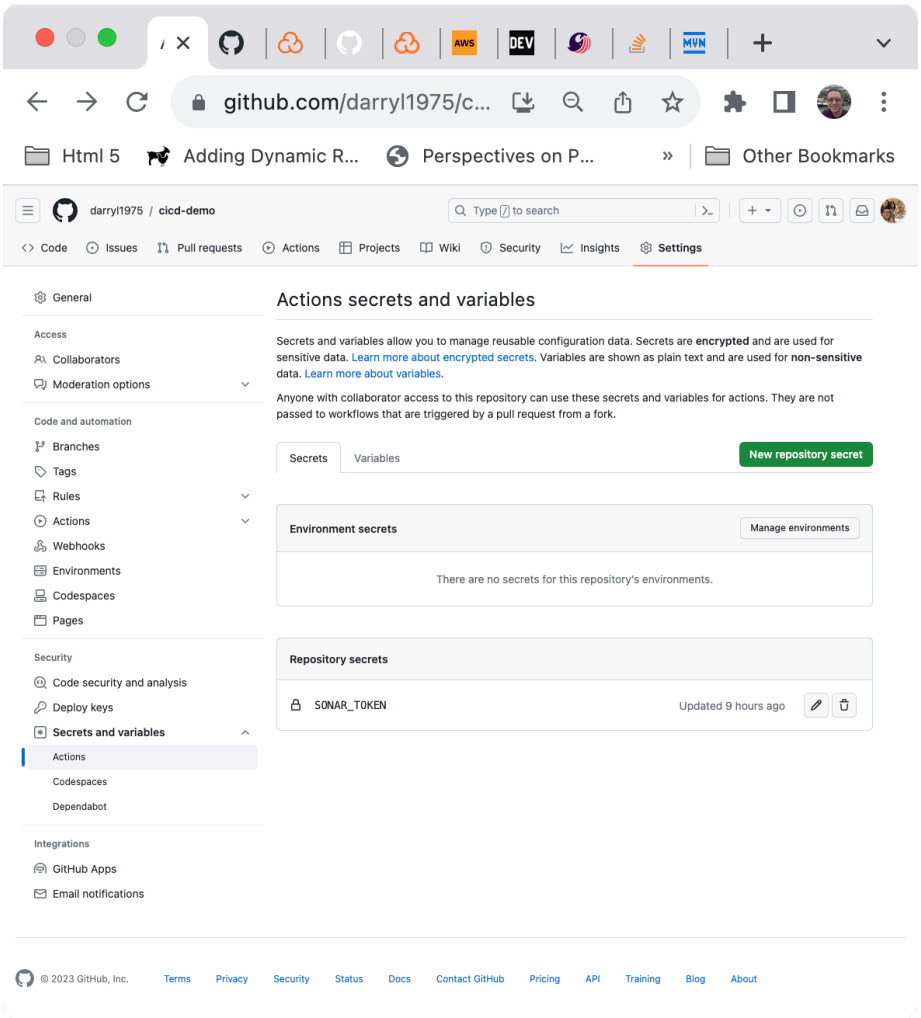
5. Bind to GitHub repo by selecting ***With GitHub Actions*** for your Analysis Method.



6. Create GitHub Secret for Sonar to integrate with Github. On your Github repo for the project, select ***Secrets and variables > Actions*** as shown.



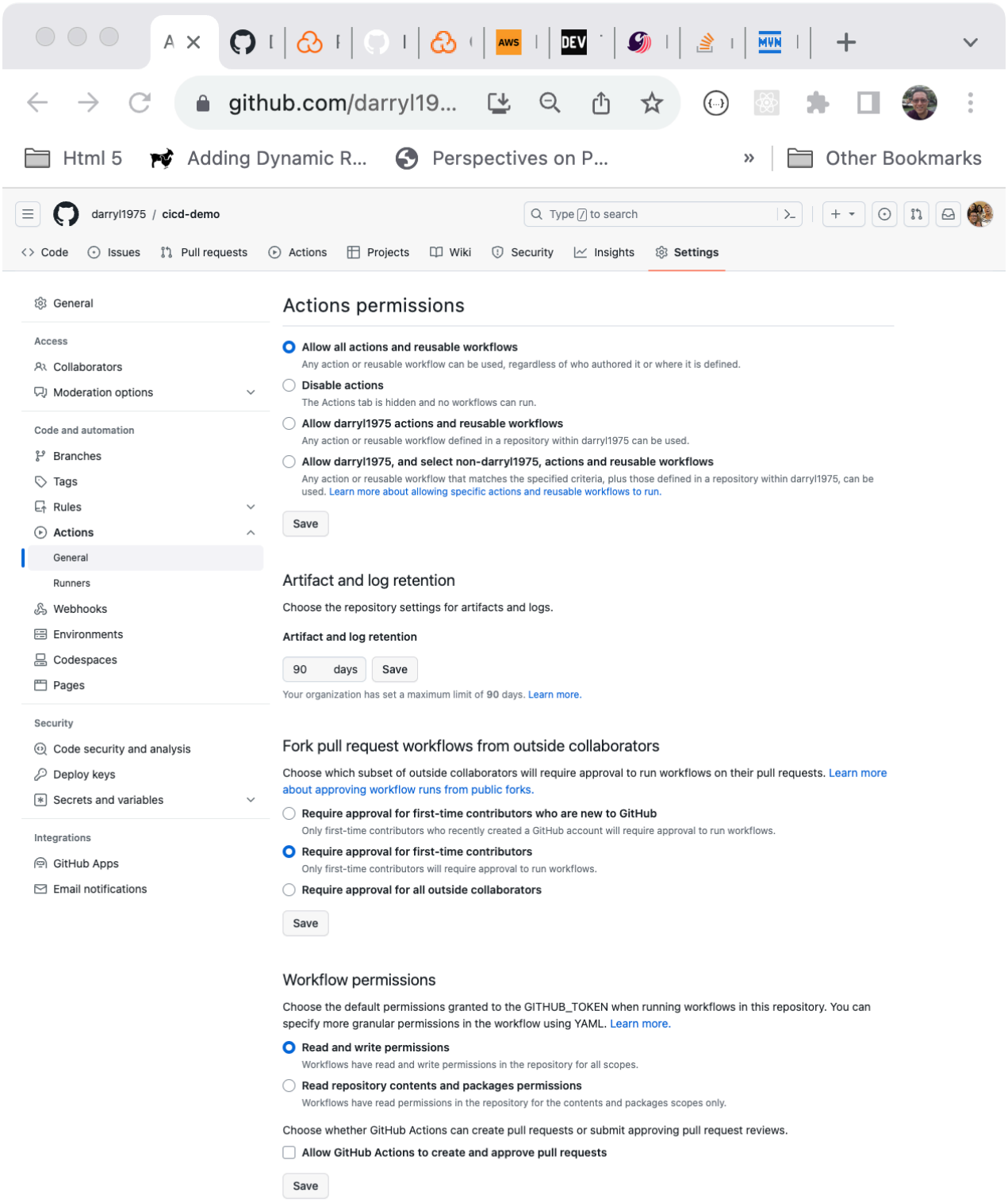
7. Your github repo Actions secrets and variables page should look similar to the following.



8. Update your Spring Boot project build configuration ***pom.xml*** file.

```
<properties>
  <sonar.organization>darryl1975</sonar.organization>
  <sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

9. In your github repo, update Github ***Actions Permissions*** for your project as shown.



10. Modify build workflow to test workflow

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
    types: [opened, synchronize, reopened]

jobs:
  test:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 20
      uses: actions/setup-java@v3
      with:
        java-version: '20'
        distribution: 'temurin'
        cache: maven
    - name: Run Test with Maven
      run: mvn -B test
```

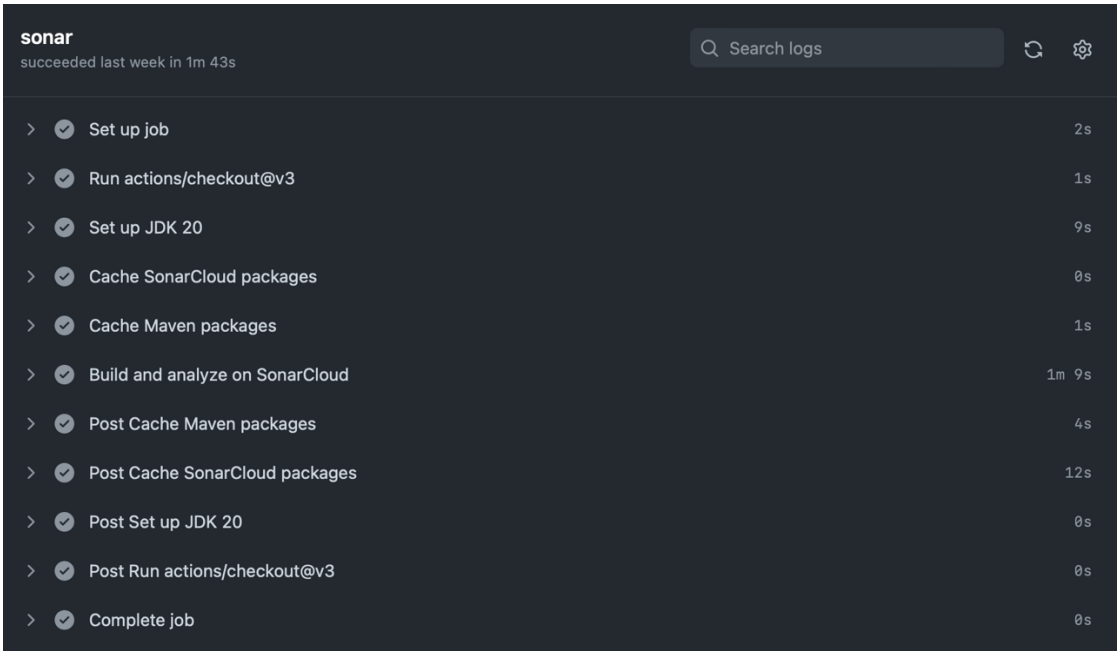
11. Create or update your .github/workflows/maven.yml. Add in SonarQube workflow.

```yaml
  sonar:
    needs: test
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 20
      uses: actions/setup-java@v3
      with:
        java-version: '20'
        distribution: 'temurin'
        cache: maven
    - name: Cache SonarCloud packages
      uses: actions/cache@v3
      with:
        path: ~/.sonar/cache
        key: ${{ runner.os }}-sonar
        restore-keys: ${{ runner.os }}-sonar
    - name: Cache Maven packages
      uses: actions/cache@v3
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2
    - name: Build and analyze on SonarCloud
```

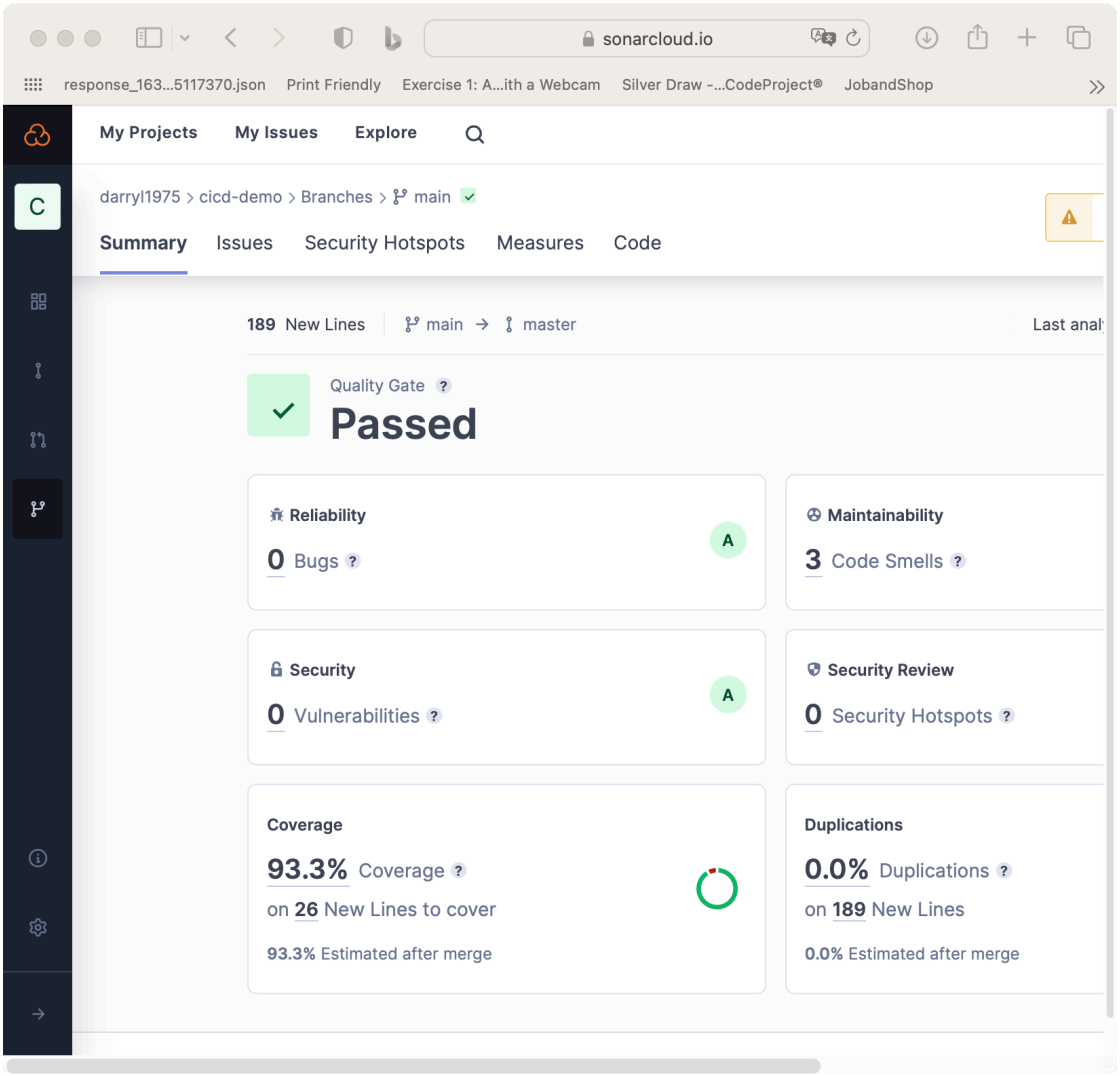
```
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information,
if any
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
      run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -
Dsonar.projectKey=darryl1975_cicd-demo
```

With references from https://github.com/SonarCloud-Spring-Boot/

12. Analyse your workflow

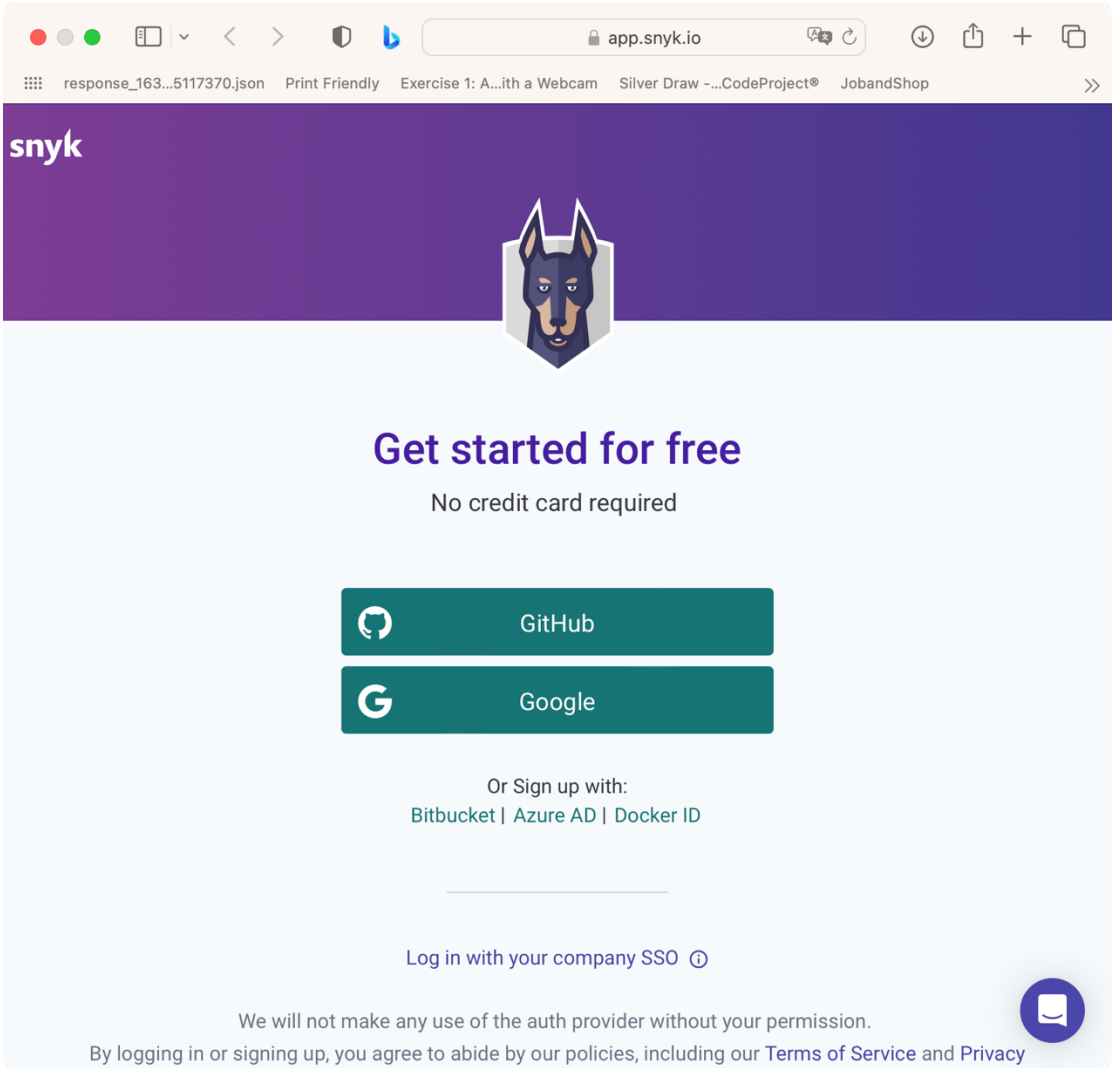


13. Analyse your sonar results

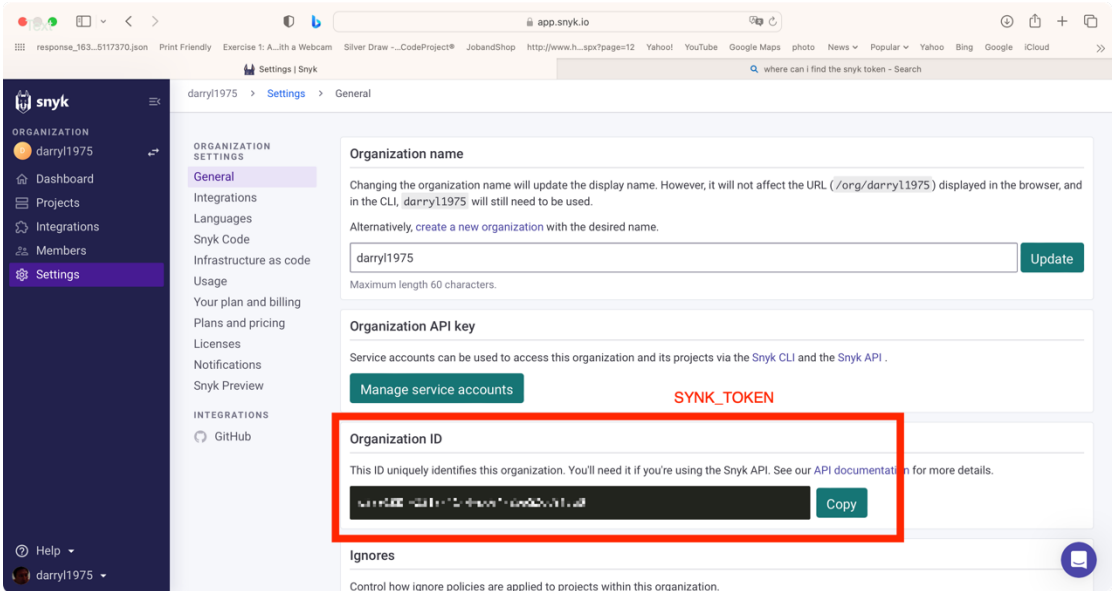


Integrating Snyk into your Github CI/CD (*Estimate: 45 minutes*)

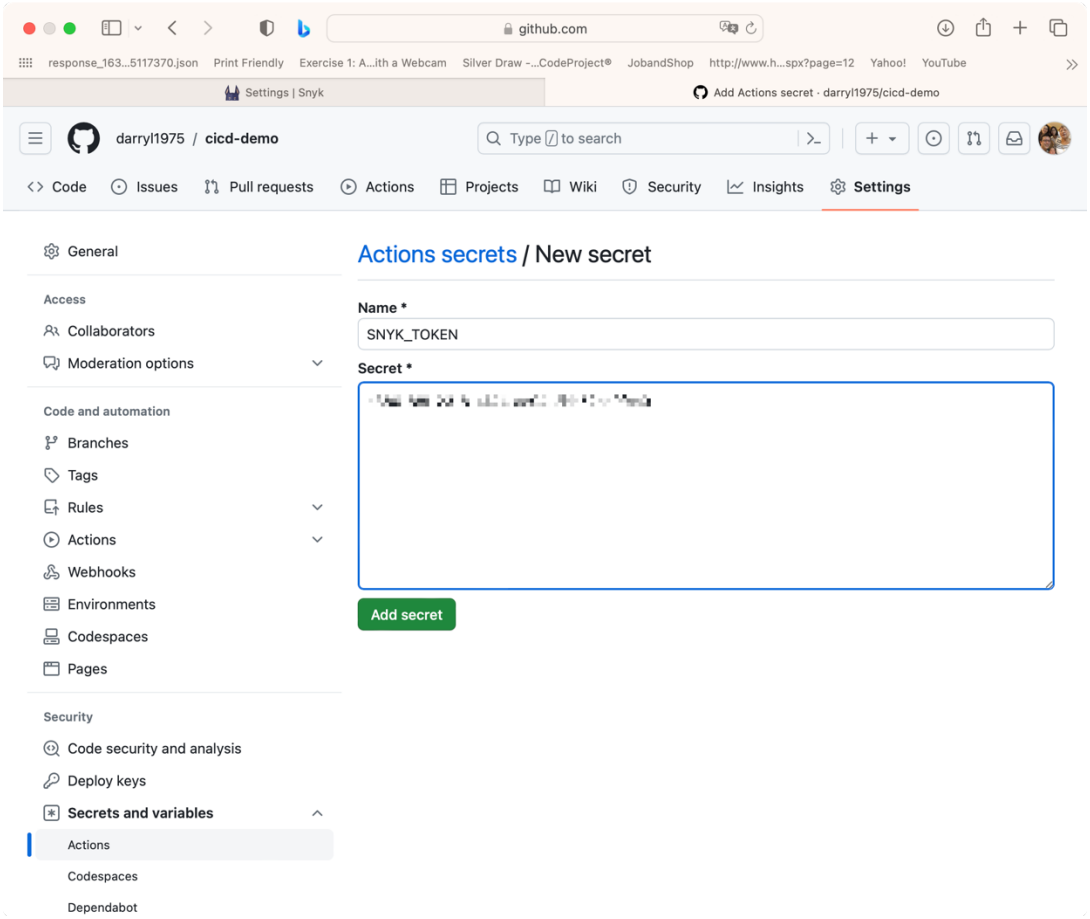
1. Login to your snyk account



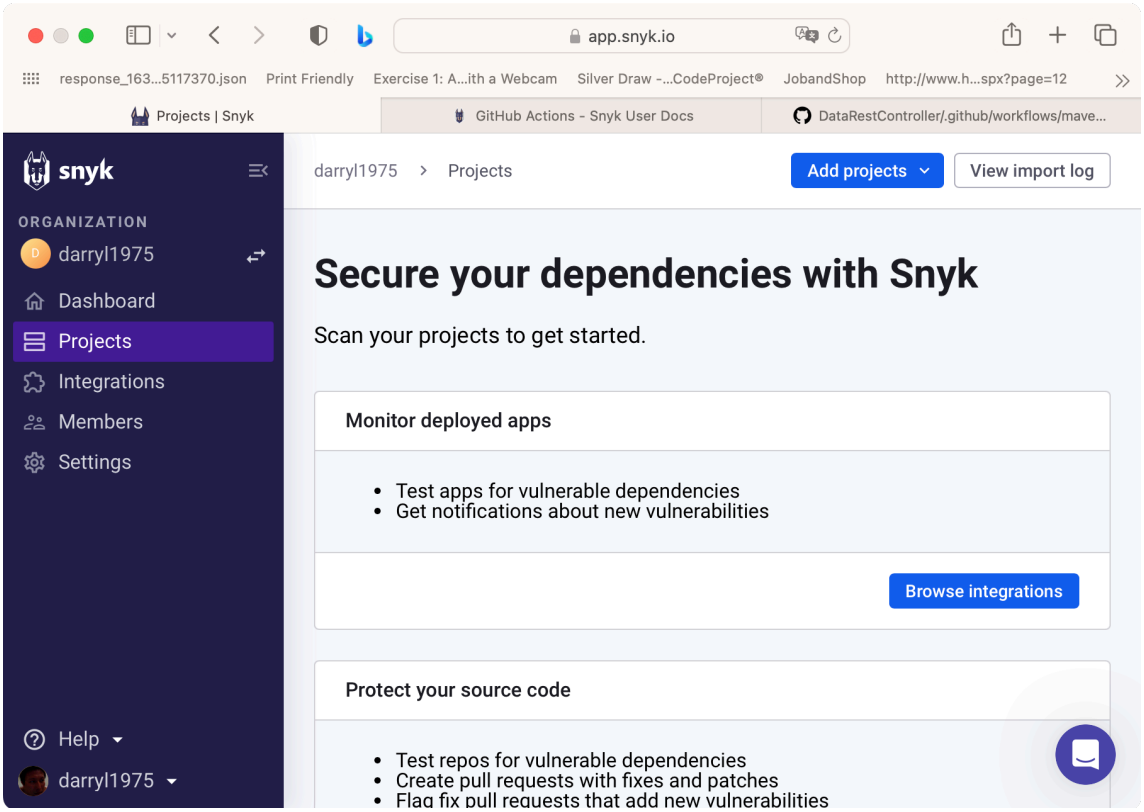
2. Retrieve your SNYK Token required for integrating with GitHub.



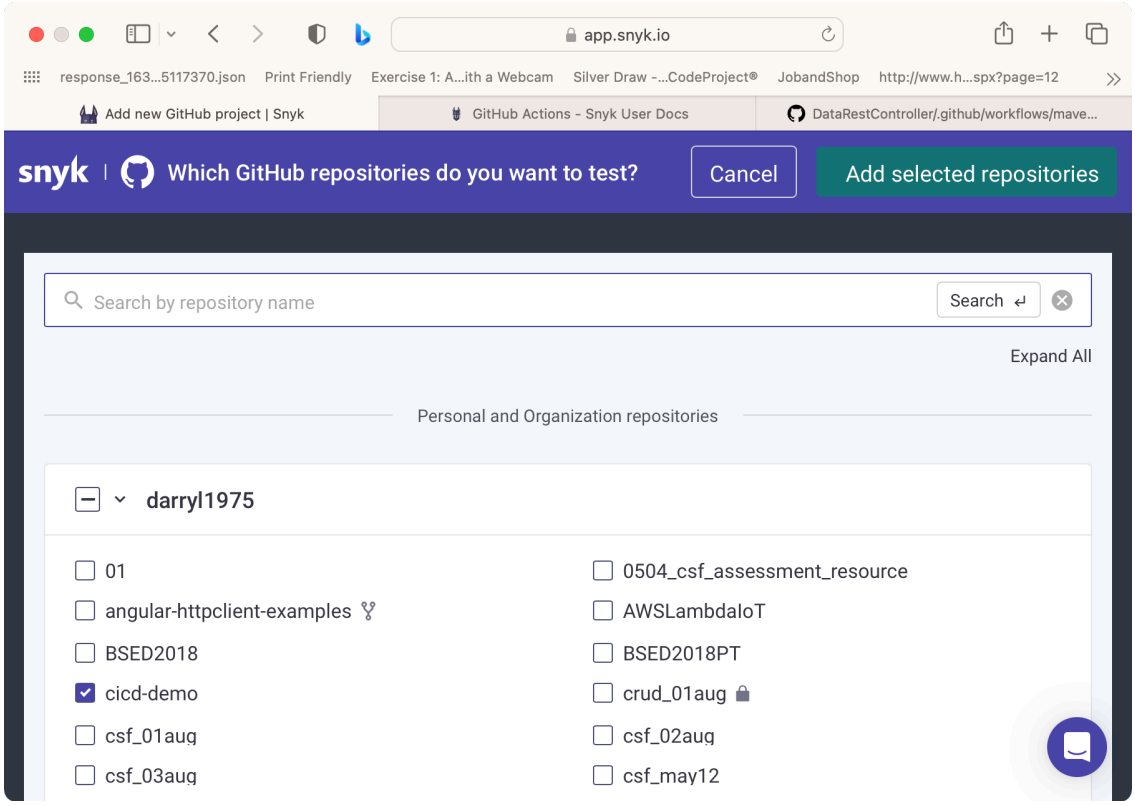
3. Under ***Secrets and variables > Actions*** on your github repo, setup **SNYK_TOKEN** in github project settings.



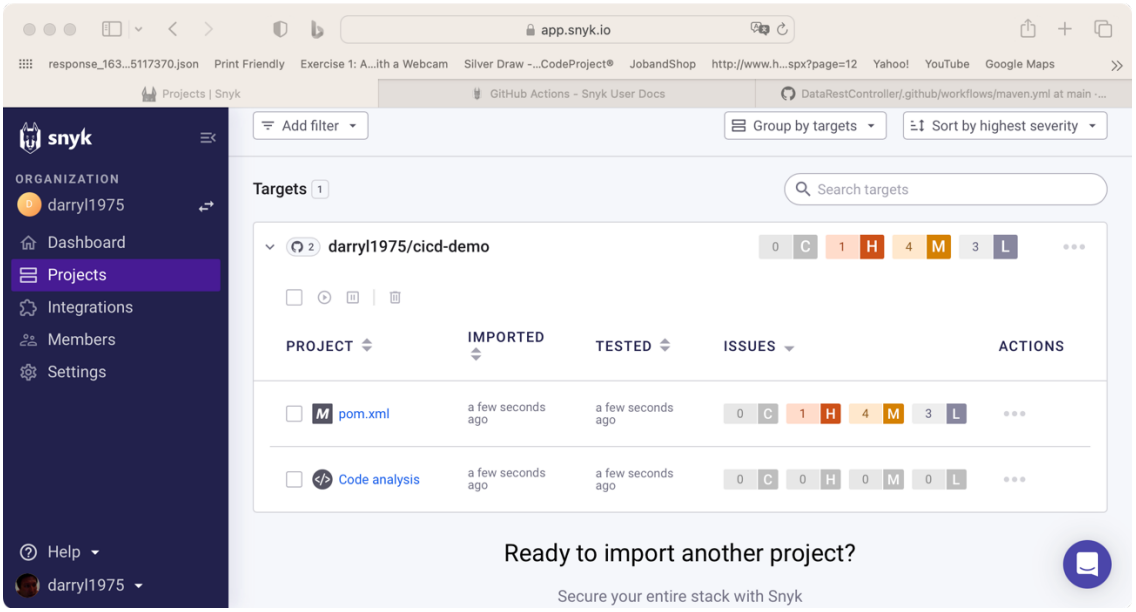
4. Add a new Synk project. Under ***Projects***, click ***Add projects***.



5. Select the github project to add to snyk.



6. Ensure your project has been added and scanned.



7. Add the Snyk configuration to your git workflow.

```yaml
  snyk:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
      - name: Run Snyk to check for vulnerabilities
        uses: snyk/actions/maven@master
        continue-on-error: true # To make sure that SARIF upload gets called
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
```

Part Three

Deploying Spring Boot application to Railway (*Estimate: 30 minutes*)

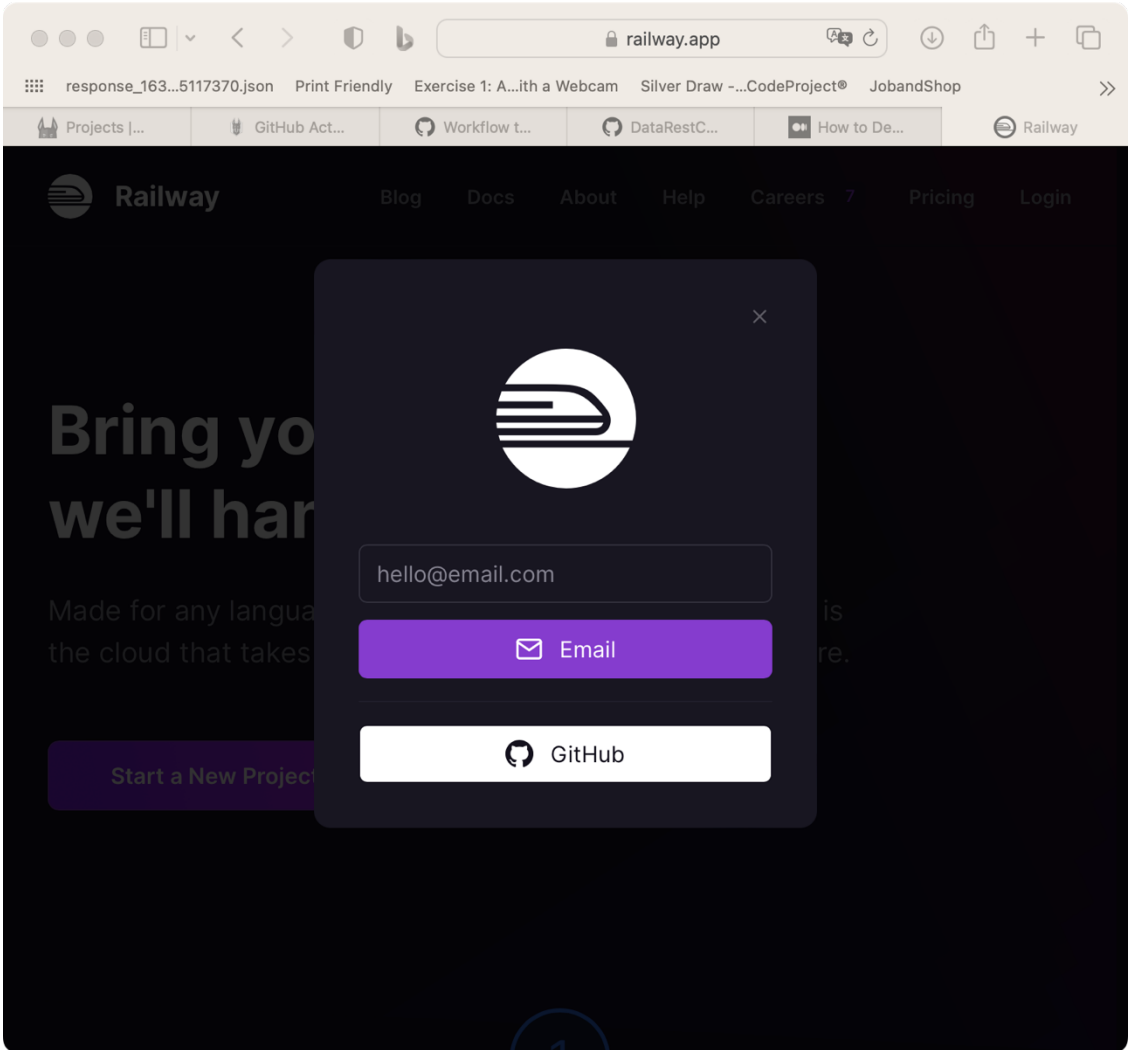
(Optional: For informaiton & references only.)

To integrate this project with railway you will need to do two things.

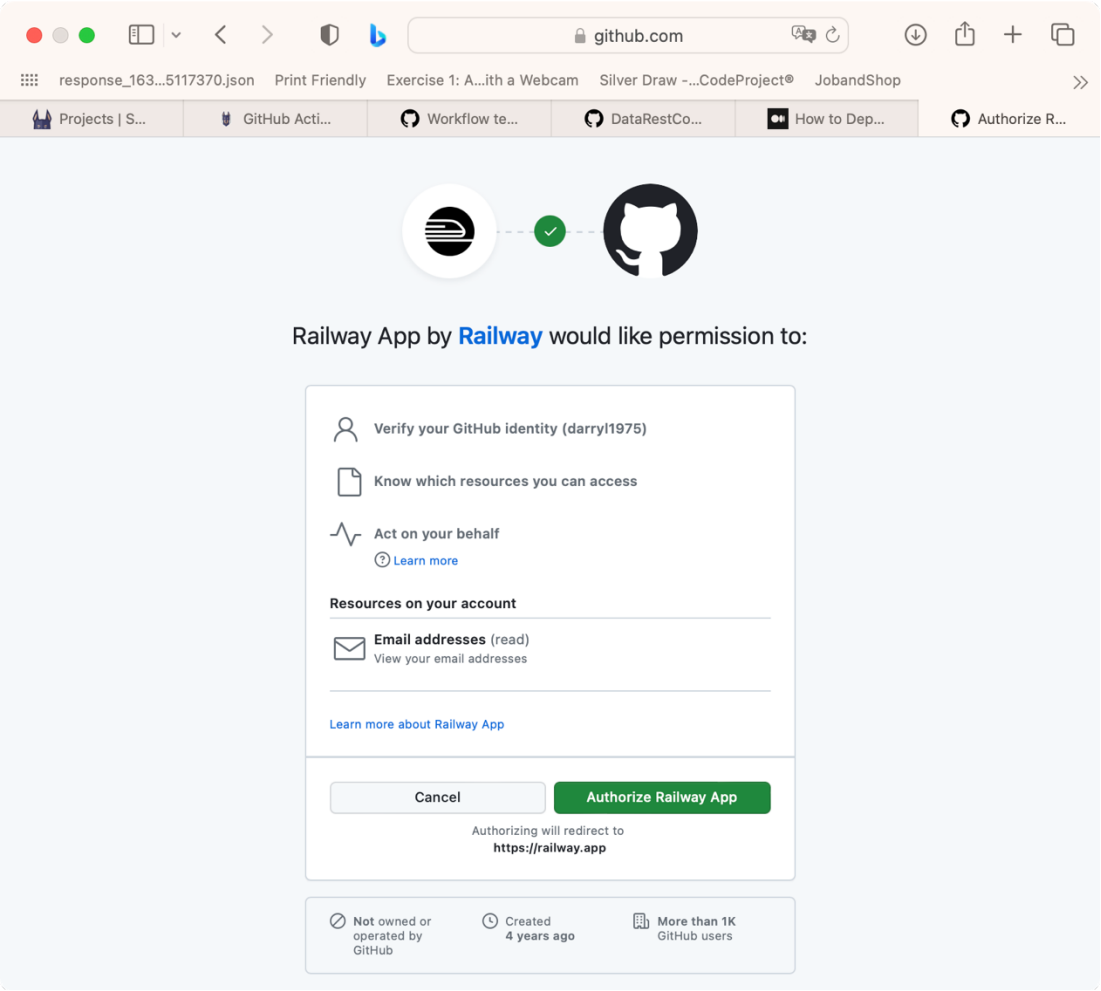
6. Create a Railway.app account
7. Connect Railway.app to your github account app

After completing this section of the workshop, Github triggers will be setup at Railway app so that your app can be automatically deployed when you push to a selected branch within the connected repository.

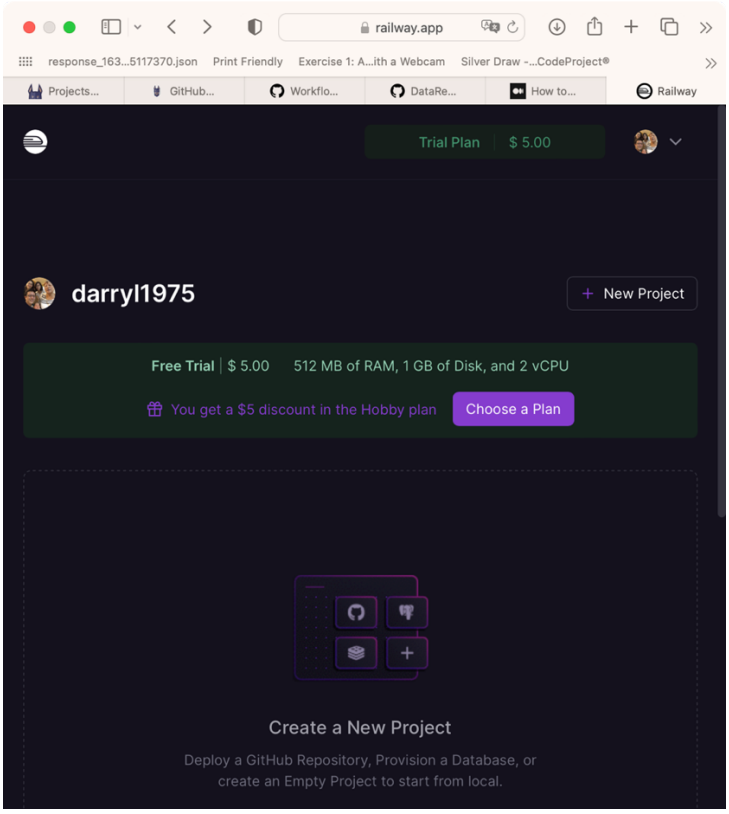
1. Login Railways using Github account.



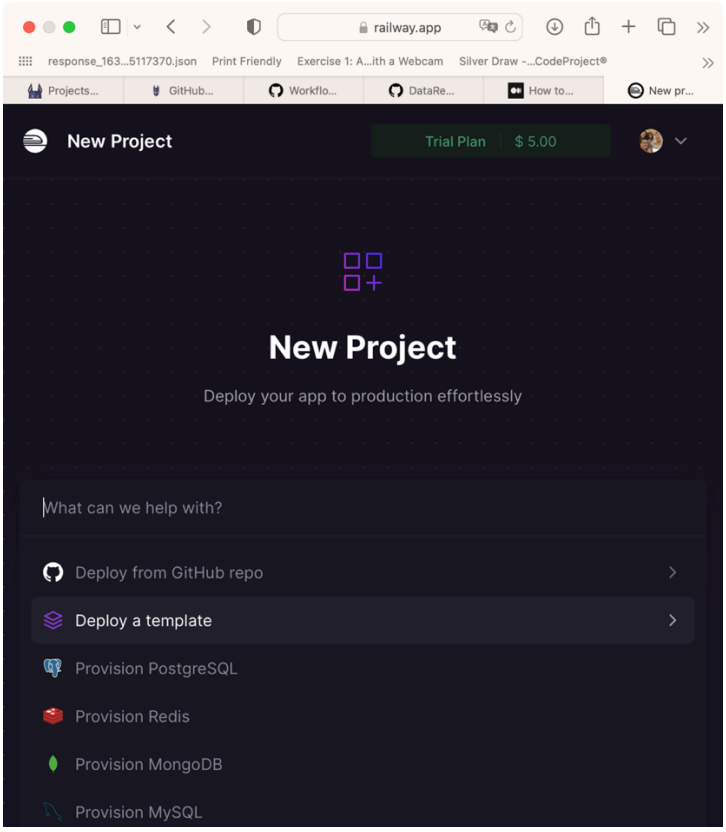
2. If you are logging in via Github (recommended), authorize Railway app.



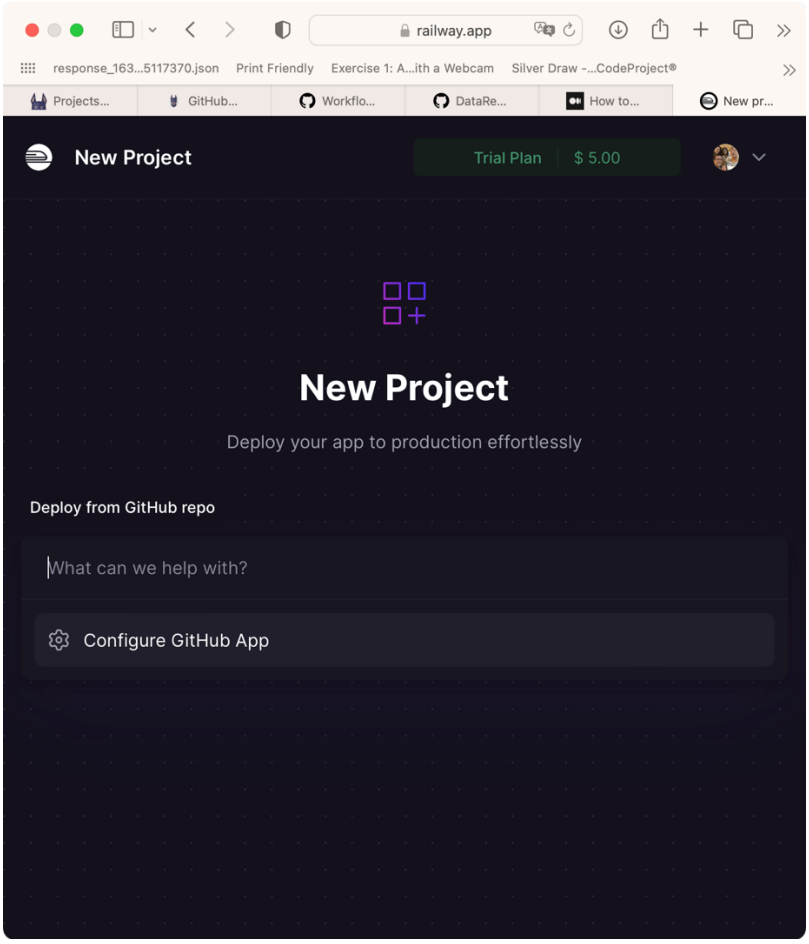
3. Create a ***New Project*** on Railway.



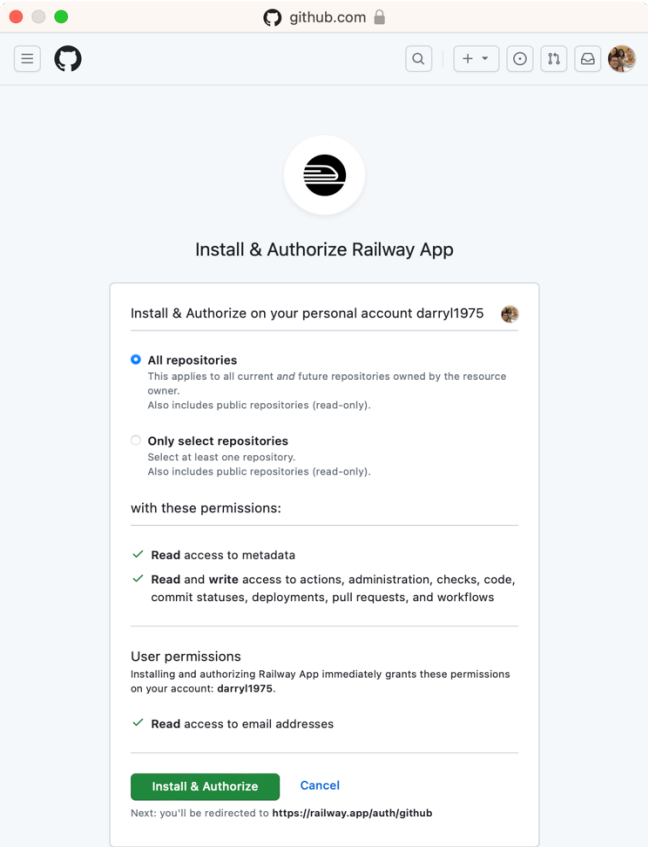
4. Deploy from Github Repo.

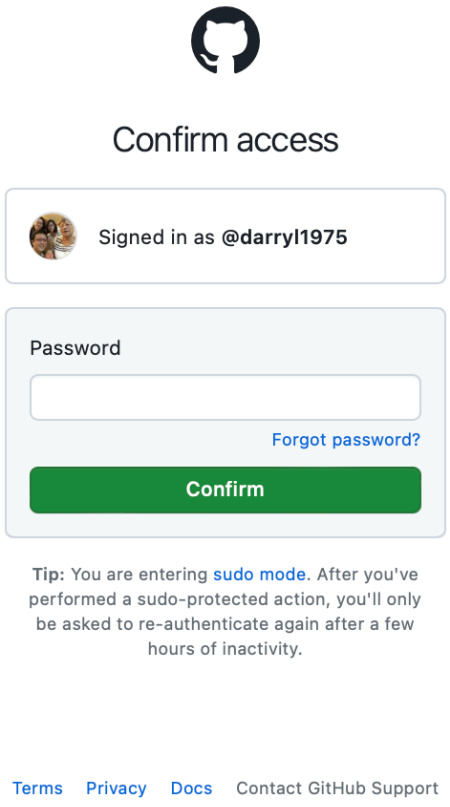


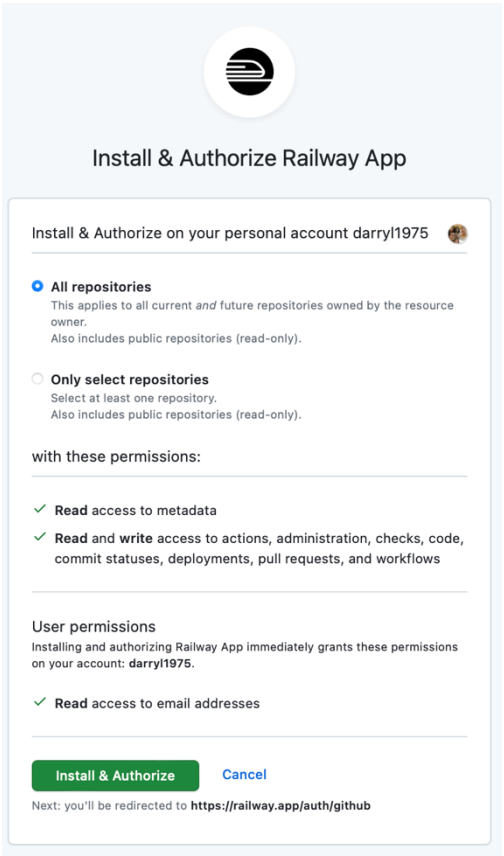
5. ***Configure GitHub App*** on Railway.



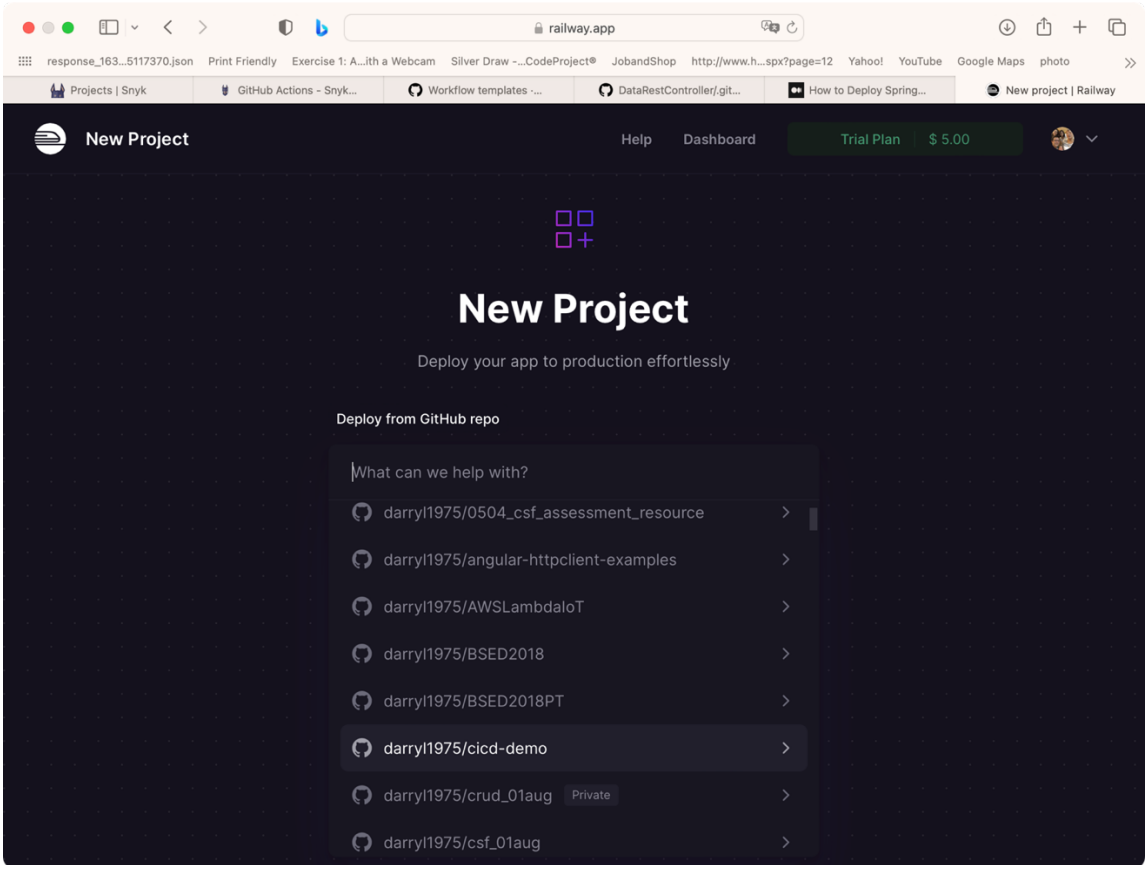
6. Install and authorize Railway app.



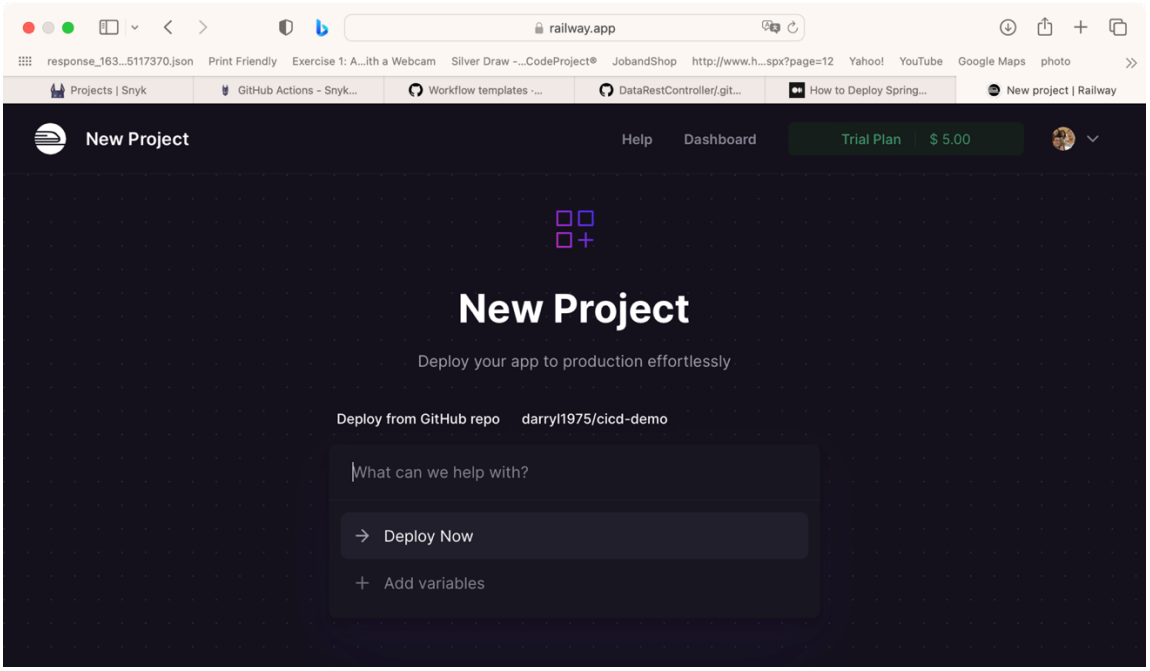




7. Choose your project on Railway.

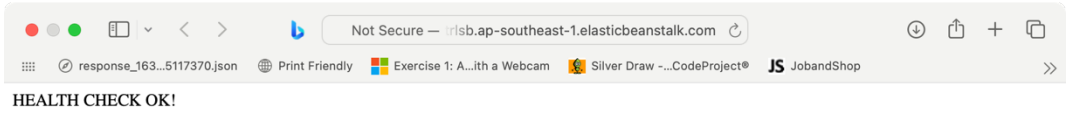


8. Deploy your project onto Railway.

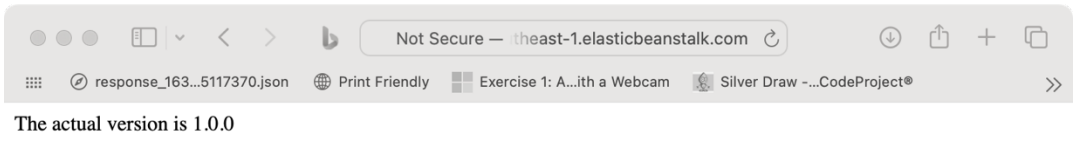


9. You may proceed to test and check your deployed application over the Internet on your browser/Postman as shown.

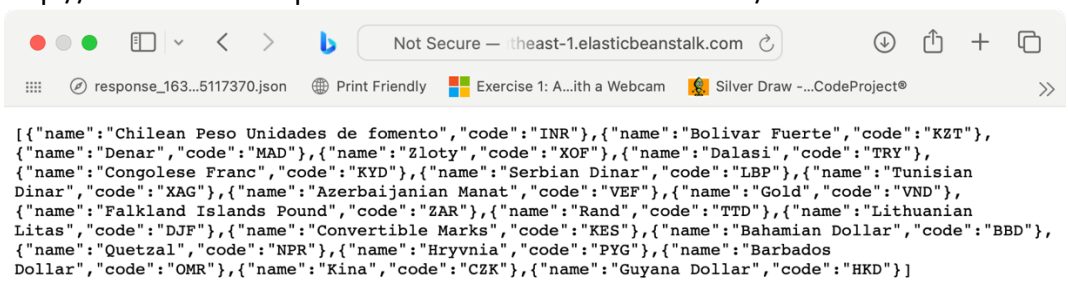
http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/



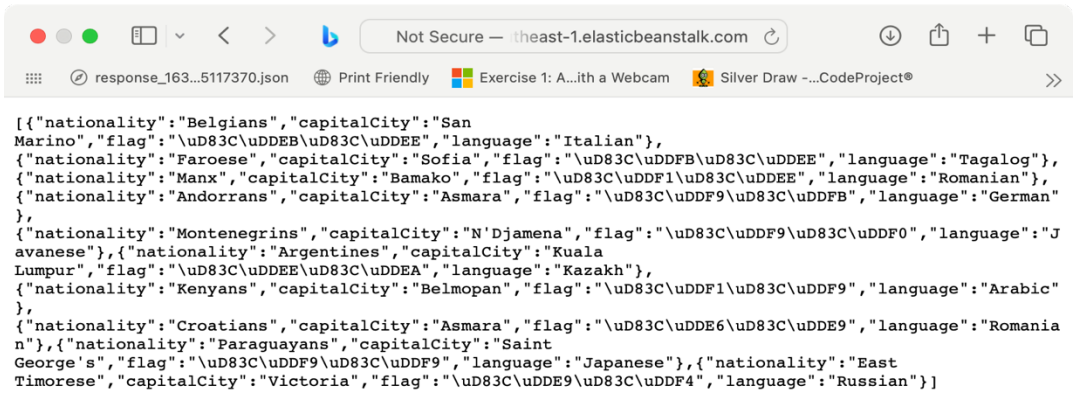
http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/version



http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/currencies



http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/nations



Dockerizing Springboot application (*Estimate: 30 minutes*)

This workshop is optional. One of the key reason why we want to dockerize our application is to make it easy to create and deploy applications in a standardized way by packaging our application and all its dependencies into a single "container" that can be run anywhere, regardless of the underlying system.

To dockerize an application, we will need a dockerfile, which is a simple text file that describe how to build a Docker container. In the dockerfile, you will specify the base image to use, files to include, commands to run, and more in order to automate the process of building a docker container for your application.

1. Create a ***dockerfile*** in the root folder of your Spring Boot project.

```
# Use a Java base image
FROM openjdk:17-oracle

# Set the working directory to /app
WORKDIR /app

# Copy the Spring Boot application JAR file into the Docker image
COPY target/cicd-demo-0.0.1-SNAPSHOT.jar /app/cicd-demo-0.0.1-SNAPSHOT.jar

# Expose the port that the Spring Boot application is listening on
EXPOSE 5000

# Run the Spring Boot application when the container starts
CMD ["java", "-jar", "cicd-demo-0.0.1-SNAPSHOT.jar"]
```

The ***FROM*** command specifies the base Java image that we'll be using:***openjdk:17-oracle***. This reference to version 17 of the open-source reference implementation of Java SE platform. This command specifies the base image that we are using to build our Docker image and must correspond to the version specified in the POM file.

The ***WORKDIR*** command specifies the working directory inside the Docker image to ***/app*** where the Spring Boot application JAR file will be copied to.

The ***COPY*** command copies files from our local machine into the Docker image. In this workshop, it will copy the ***cicd-demo-0.0.1-SNAPSHOT.jar*** from the target directory to the /app directory in the Docker image.

The ***EXPOSE*** command is used to expose ports in the Dockerfile to listen to specific ports when the Docker container runs. The application will listen on port ***5000***.

The ***CMD*** command runs a command in the Docker image. In this workshop, we use the ***java -jar cicd-demo-0.0.1-SNAPSHOT.jar*** to start our Spring Boot application.

2. In the project root folder, build the Spring Boot application to generate the JAR file needed for the Docker image. The following command will generate the JAR file in the 'target' folder.
   ./mvnw clean install

3. In the project root folder, build the Docker image.
   docker build -t cicd-demo-app .

   -t flag is used to tag your image with a name
   . at the end of the command tells Docker to look for the Dockerfile in the current directory

4. Check your Docker image has been created.
   docker images

5. Run the Spring Boot app image in a container.
   docker run -it -p 8090:5000 cicd-demp-app

   This command will tell Docker to run the container and forward the exposted port 5000 to port 8090. To access your application, you should be able to visit http://localhost:8090/ in your browser.

6. Tag your Docker image using the following format:
   docker tag [Image-Id] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

   e.g. ***docker tag ae7c8a26aad7 darryl1975/cicd-demo***

7. Share your image publicly by pushing the image to Docker hub using the tag created in the previous step.
   docker push darryl1975/cicd-demo