

NICF – Essential Practices for Agile Teams

Continuous Integration Workshop with Git Workflow

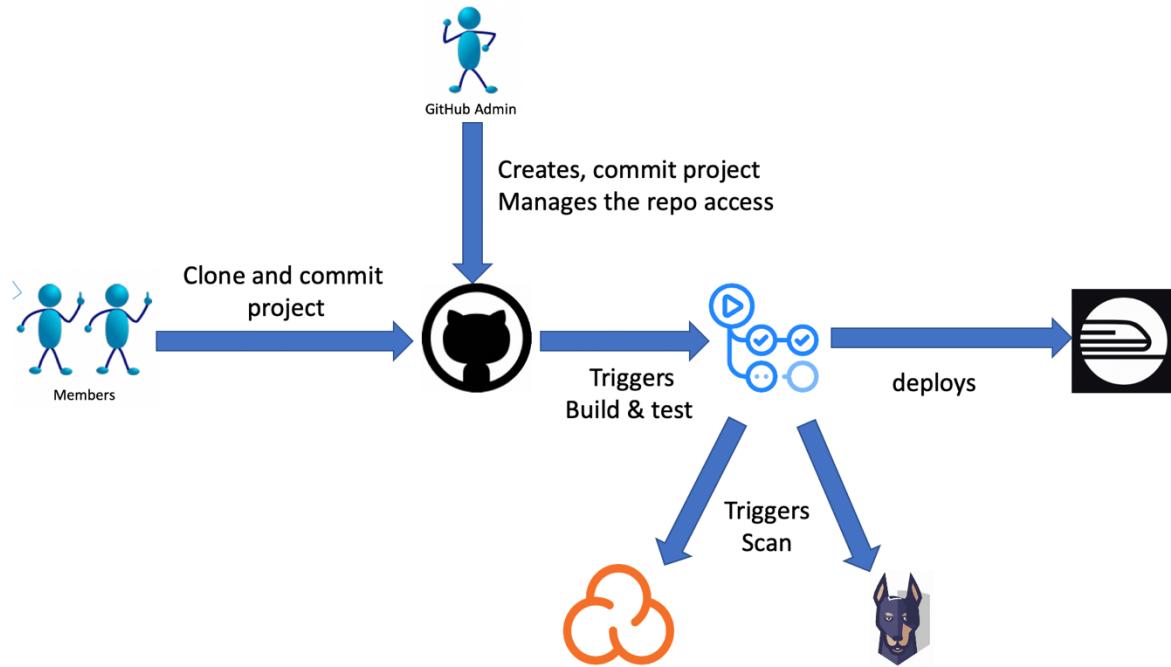
Copyright 2023 NUS-ISS. The contents contained in this document may not be reproduced in any form or by any means, without the written permission of ISS and NUS, other than for the purpose for which it has been supplied.

Table of Contents

Overview.....	3
Pre-requisite (Estimate: 1.5 hours)	4
Part One.....	5
Spring Initializr (<i>Estimate: 10 minutes</i>)	5
Constructing project code (<i>Estimate: 30 minutes</i>)	7
Testing the APIs (<i>Estimate: 15 minutes</i>)	9
Adding Unit test to the project (<i>Estimate: 20 minutes</i>)	11
Part Two	12
Setup project repo (<i>Estimate: 15 minutes</i>).....	12
Concept of Git Actions	14
Components of a workflow file.....	16
name	16
on	16
env.....	17
jobs	17
steps	19
Create a new Git Workflow for Java project (<i>Estimate: 10 minutes</i>).....	21
Factor Code Coverage in testing to ensure completeness and code quality (<i>Estimate: 10 minutes</i>).....	23
Integrating SonarCloud in your CI pipeline (<i>Estimate: 1 hour</i>)	24
Integrating Snyk into your Github CI/CD (<i>Estimate: 45 minutes</i>)	34
Part Three	39
Deploying Spring Boot application to Railway (<i>Estimate: 30 minutes</i>).....	39
Dockerizing Springboot application (<i>Estimate: 30 minutes</i>).....	46

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.

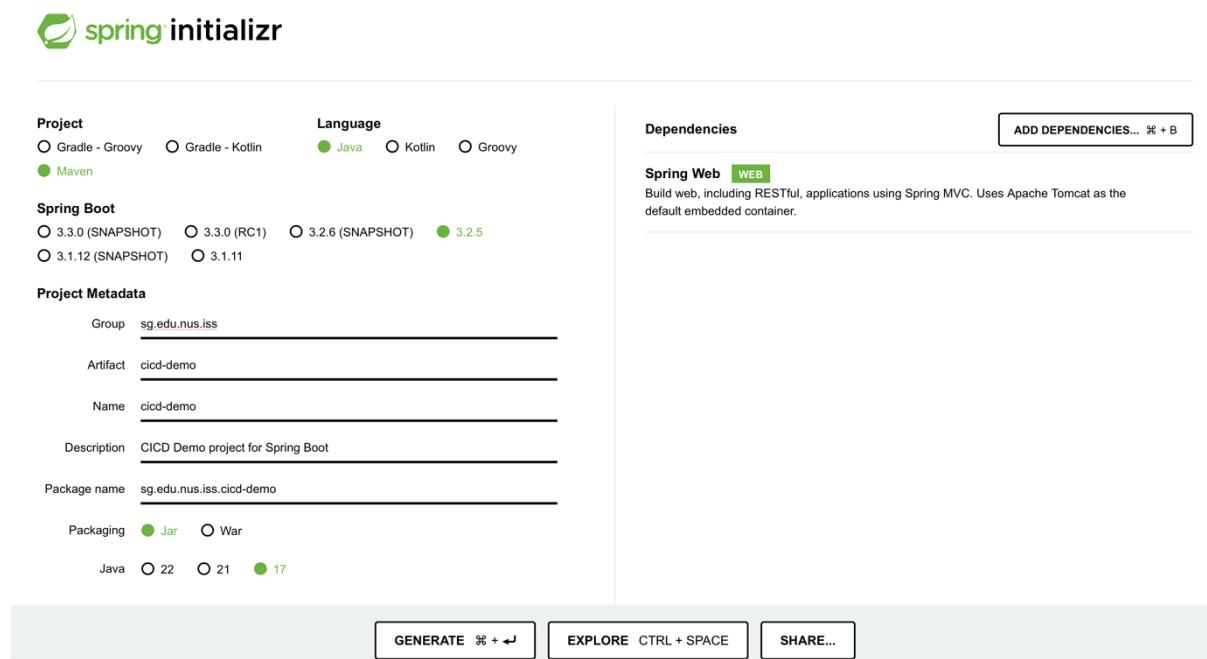
The key driver to the success of Spring boot in Java and Spring framework adoption is the convention over configuration. Instead of the hassle, tedious and length steps of configuring Spring or 3rd-party components, Spring provides out-of-the-box auto configurations that follows common configuration 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



The screenshot shows the Spring Initializr web application interface. The configuration is set up for a "Spring Web" project:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 3.2.5 (selected)
- Dependencies:** Spring Web (selected)
- Project Metadata:**
 - Group: sg.edu.nus.iss
 - Artifact: cicd-demo
 - Name: cicd-demo
 - Description: CICD Demo project for Spring Boot
 - Package name: sg.edu.nus.iss.cicd-demo
 - Packaging: Jar (selected)
 - Java version: 22 (selected)
- Buttons at the bottom:** GENERATE, EXPLORE, SHARE...

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

```

        .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

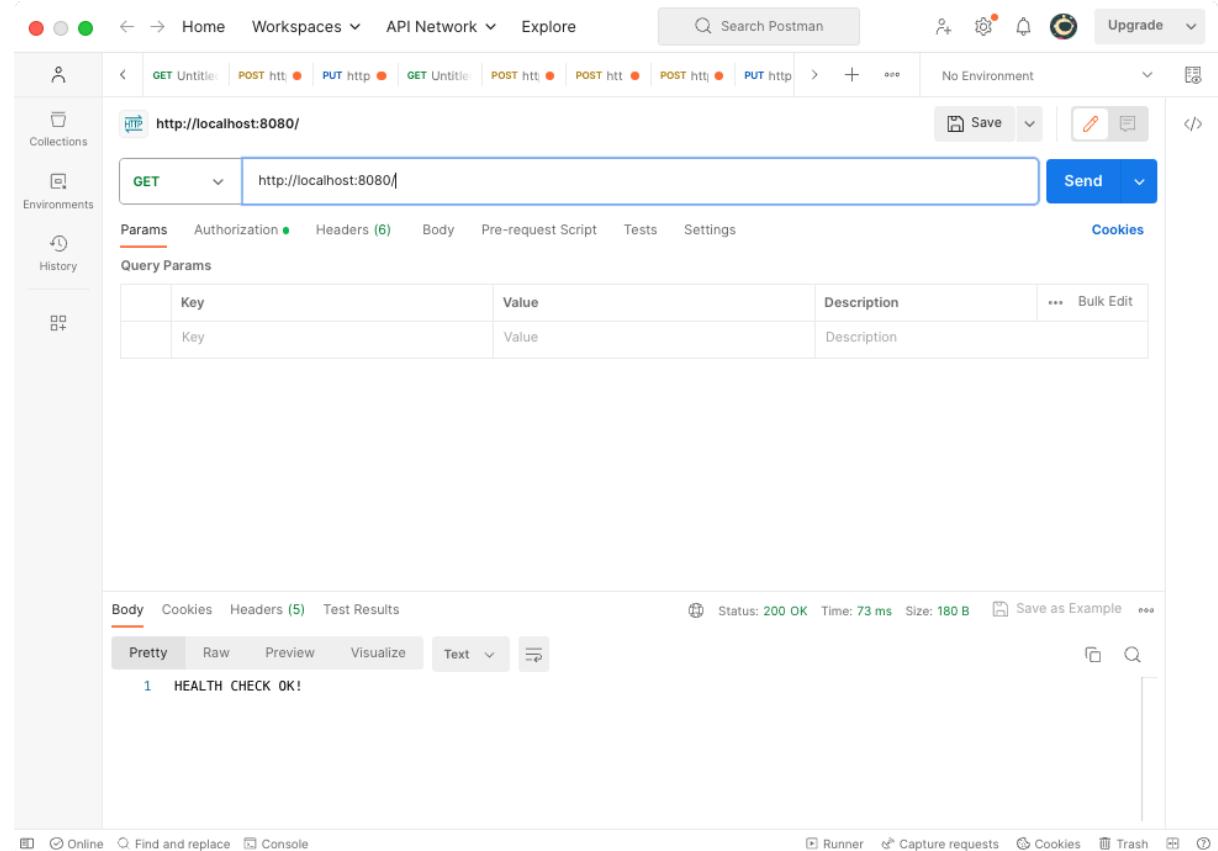
```

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/Darryl
U/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
2023-08-22T14:22:24.328+08:00 INFO 47348 — [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 337 ms
2023-08-22T14:22:24.338+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)

```

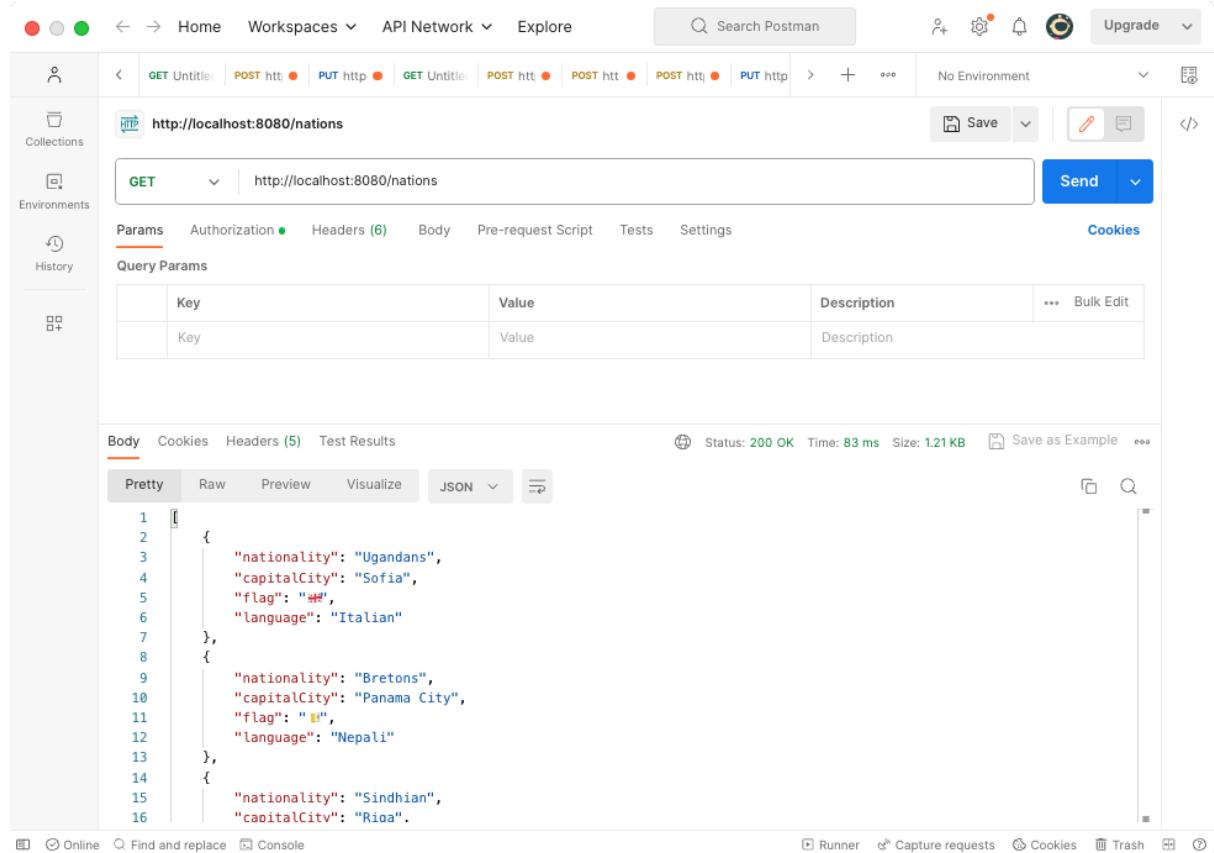
Testing the APIs (*Estimate: 15 minutes*)

1. Use Postman to test your API application endpoints.



The screenshot shows the Postman application interface. On the left, there's a sidebar with tabs for Collections, Environments, and History. The main area shows a request configuration for a GET method to `http://localhost:8080/`. The 'Params' tab is selected, showing a single entry with 'Key' and 'Value'. Below the request, the 'Body' tab is selected, displaying the response content: `1 HEALTH CHECK OK!`. The status bar at the bottom indicates a `200 OK` status, `73 ms` time, and `180 B` size.

NICF – Essential Practices for Agile Teams



HTTP <http://localhost:8080/nations>

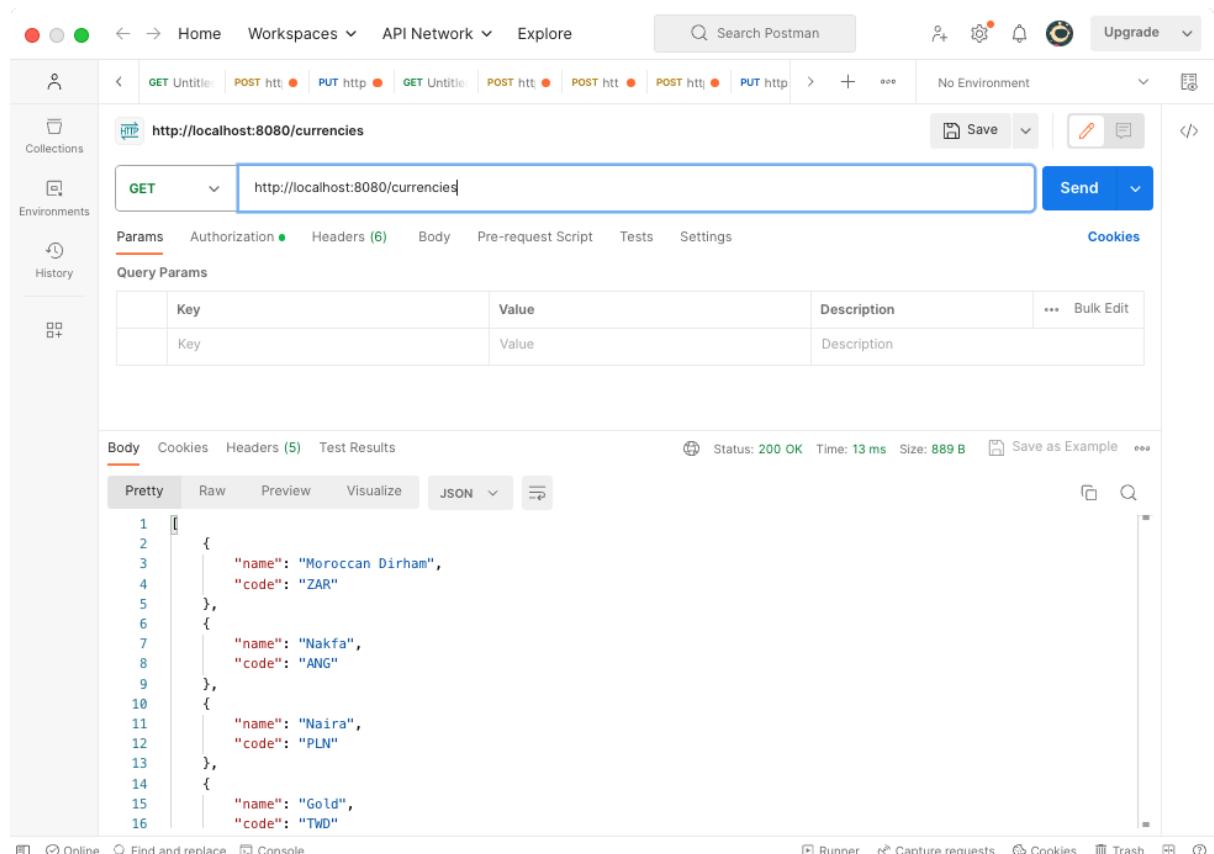
GET <http://localhost:8080/nations>

Key	Value	Description
Key	Value	Description

```

1 [
2   {
3     "nationality": "Ugandans",
4     "capitalCity": "Sofia",
5     "flag": "#",
6     "language": "Italian"
7   },
8   {
9     "nationality": "Bretons",
10    "capitalCity": "Panama City",
11    "flag": "#",
12    "language": "Nepali"
13  },
14  {
15    "nationality": "Sindhian",
16    "capitalCity": "Rios".

```



HTTP <http://localhost:8080/currencies>

GET <http://localhost:8080/currencies>

Key	Value	Description
Key	Value	Description

```

1 [
2   {
3     "name": "Moroccan Dirham",
4     "code": "ZAR"
5   },
6   {
7     "name": "Nakfa",
8     "code": "ANG"
9   },
10  {
11    "name": "Naira",
12    "code": "PLN"
13  },
14  {
15    "name": "Gold",
16    "code": "TWD"

```

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);
    }

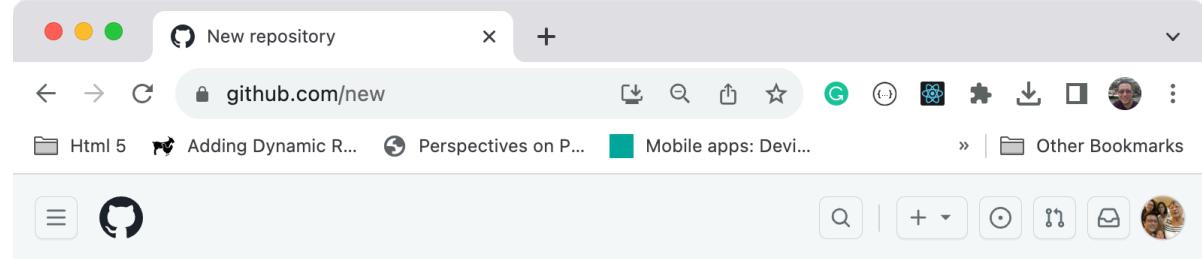
    @Test
    void currenciesLength() {
        Integer currenciesLength = dataController.getRandomCurrencies().size();
        assertEquals(20, currenciesLength);
    }
}

```

Part Two

Setup project repo (*Estimate: 15 minutes*)

1. Create a new GitHub repo for the project



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Required fields are marked with an asterisk (*).

Owner * Repository name *



darryl1975

/ cicd-demo

cicd-demo is available.

Great repository names are short and memorable. Need inspiration? How about [bookish-meme](#) ?

Description (optional)

 Public

Anyone on the internet can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file

This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

.gitignore template:None

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

License:None

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

 You are creating a public repository in your personal account.

Create repository

2. Commit your project to GitHub

Quick setup — if you've done this kind of thing before

or <https://github.com/darryl1975/cicd-demo.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# cicd-demo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/darryl1975/cicd-demo.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/darryl1975/cicd-demo.git
git branch -M main
git push -u origin main
```

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.

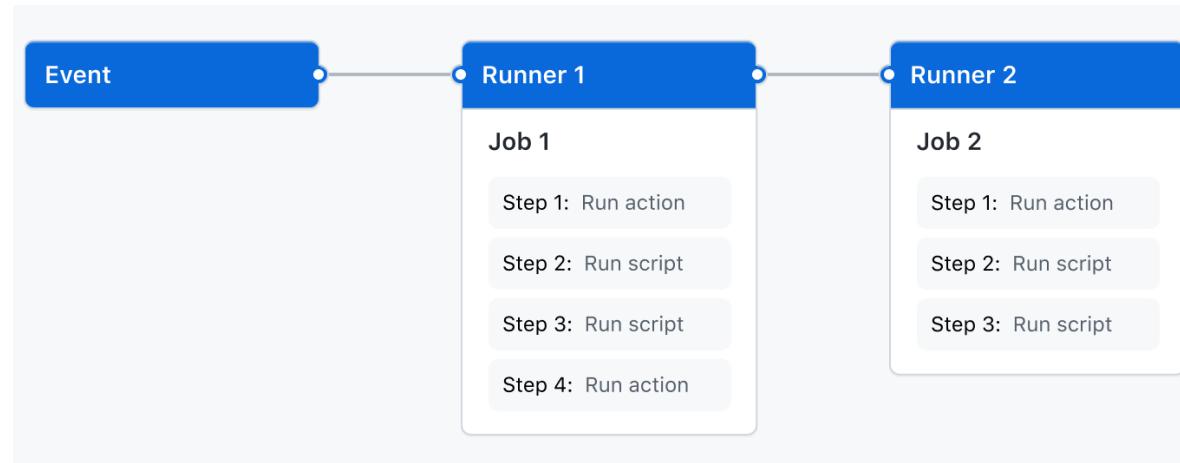
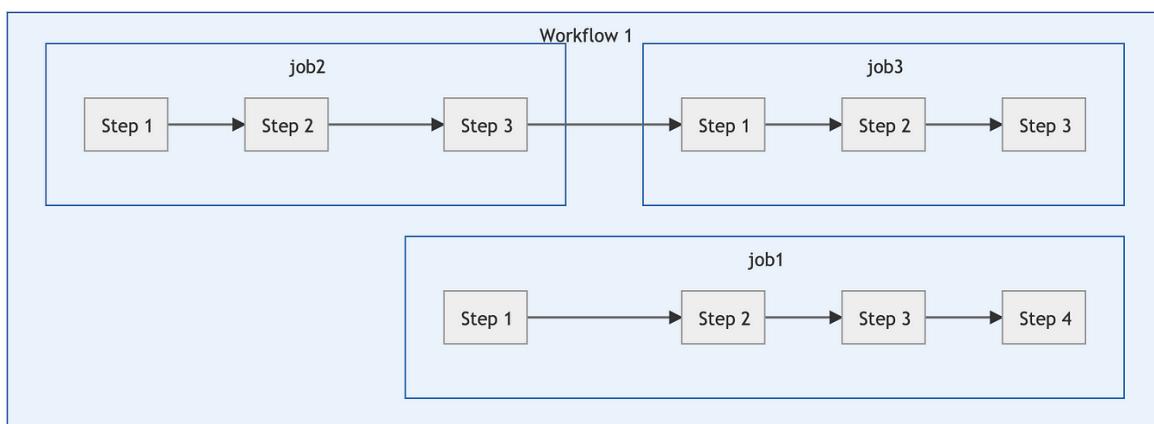


Figure from <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>

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 have 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
```

```

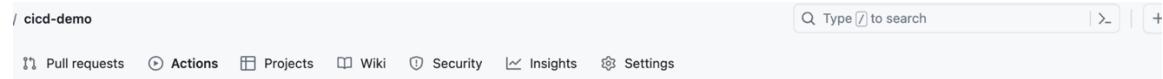
uses: ./github/actions/hello
# for this to succeed make sure you have a file named
# ./github/actions/hello/action.yaml in your repo
# I will go more into it in my subsequent blogs
# You might want to run a docker container in your steps
# simply use this
name: Running docker container
uses: docker://python:3
# 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.



A screenshot of a GitHub repository page for 'ciid-demo'. The top navigation bar shows tabs for Pull requests, Actions (which is highlighted in red), Projects, Wiki, Security, Insights, and Settings. A search bar is at the top right. Below the tabs, the repository name 'ciid-demo' is shown. The main content area is titled 'Get started with GitHub Actions' with the sub-instruction 'Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.' A link 'Skip this and set up a workflow yourself →' is present. A search bar labeled 'Search workflows' is at the top left of the workflow cards. The workflow cards are arranged in a grid under the heading 'Suggested for this repository'. One card, 'Java with Maven' by GitHub Actions, is highlighted with a red box around its entire card.

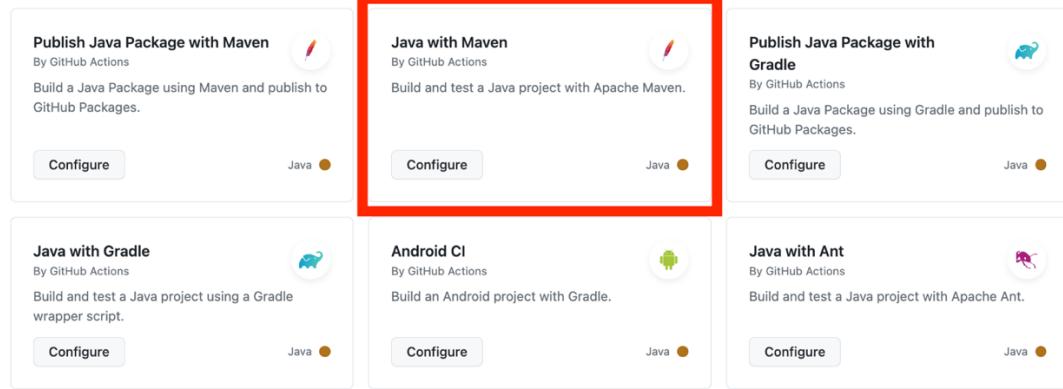
Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

[Skip this and set up a workflow yourself →](#)

Search workflows

Suggested for this repository



The screenshot shows a grid of GitHub Actions workflow cards:

- Publish Java Package with Maven** By GitHub Actions: Build a Java Package using Maven and publish to GitHub Packages. **Configure** button, Java icon.
- Java with Maven** By GitHub Actions: Build and test a Java project with Apache Maven. **Configure** button, Java icon. This card is highlighted with a red box.
- Publish Java Package with Gradle** By GitHub Actions: Build a Java Package using Gradle and publish to GitHub Packages. **Configure** button, Java icon.
- Java with Gradle** By GitHub Actions: Build and test a Java project using a Gradle wrapper script. **Configure** button, Java icon.
- Android CI** By GitHub Actions: Build an Android project with Gradle. **Configure** button, Java icon.
- Java with Ant** By GitHub Actions: Build and test a Java project with Apache Ant. **Configure** button, Java icon.

3. Analyse the workflow

```

name: Java CI with Maven

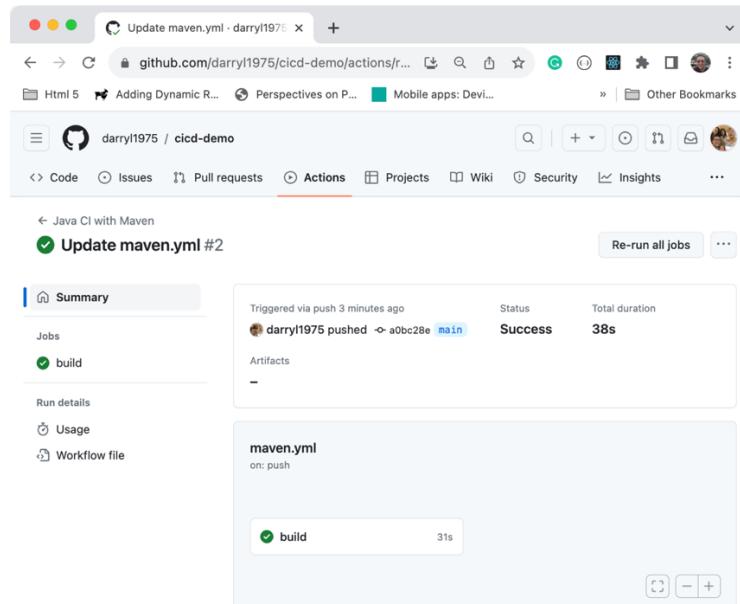
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

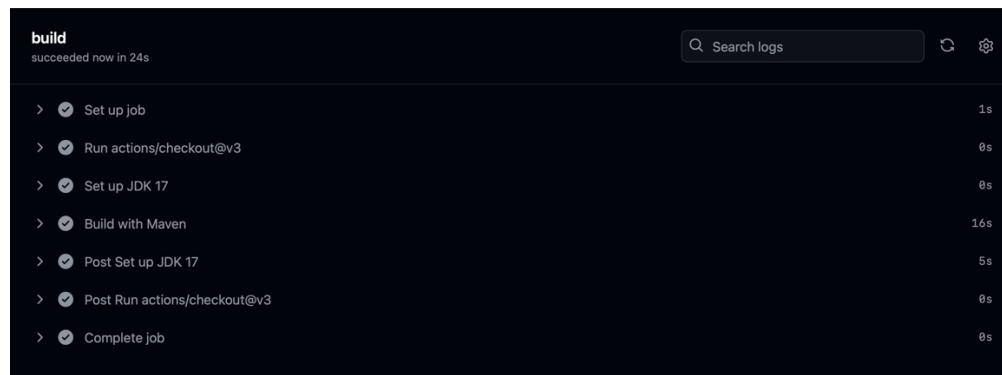
    runs-on: ubuntu-latest

    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

```



The screenshot shows the GitHub Actions interface for a repository named 'cicd-demo'. A green checkmark indicates a successful run of the 'build' job. The summary shows the job was triggered via a push 3 minutes ago, the status is 'Success', and the total duration was 38s. The logs section for the 'maven.yml' file shows the 'build' step completed successfully in 31s.



This screenshot provides a detailed view of the GitHub Actions logs for the 'build' job. It lists the following steps and their execution times:

- > Set up job: 1s
- > Run actions/checkout@v3: 0s
- > Set up JDK 17: 0s
- > Build with Maven: 16s
- > Post Set up JDK 17: 5s
- > Post Run actions/checkout@v3: 0s
- > Complete job: 0s

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.

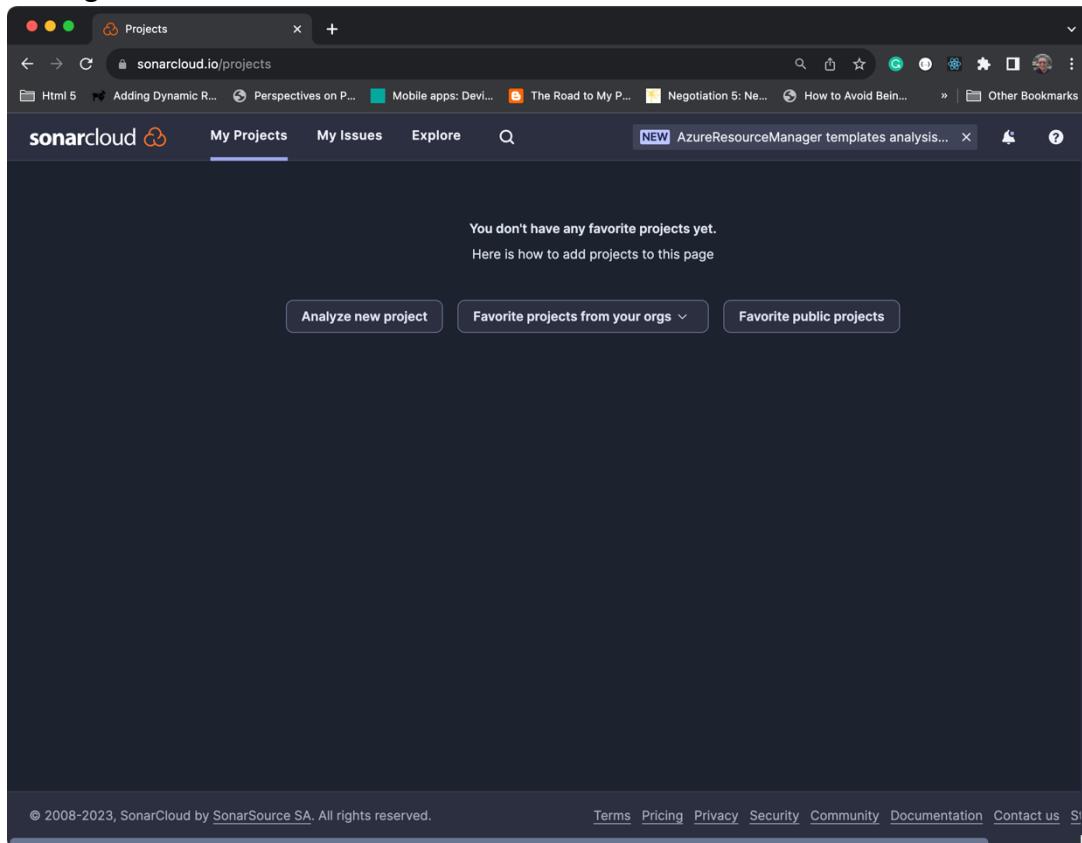
1. Add Jacoco plugin to your Spring Boot project build configuration **pom.xml** file.

```
<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*)

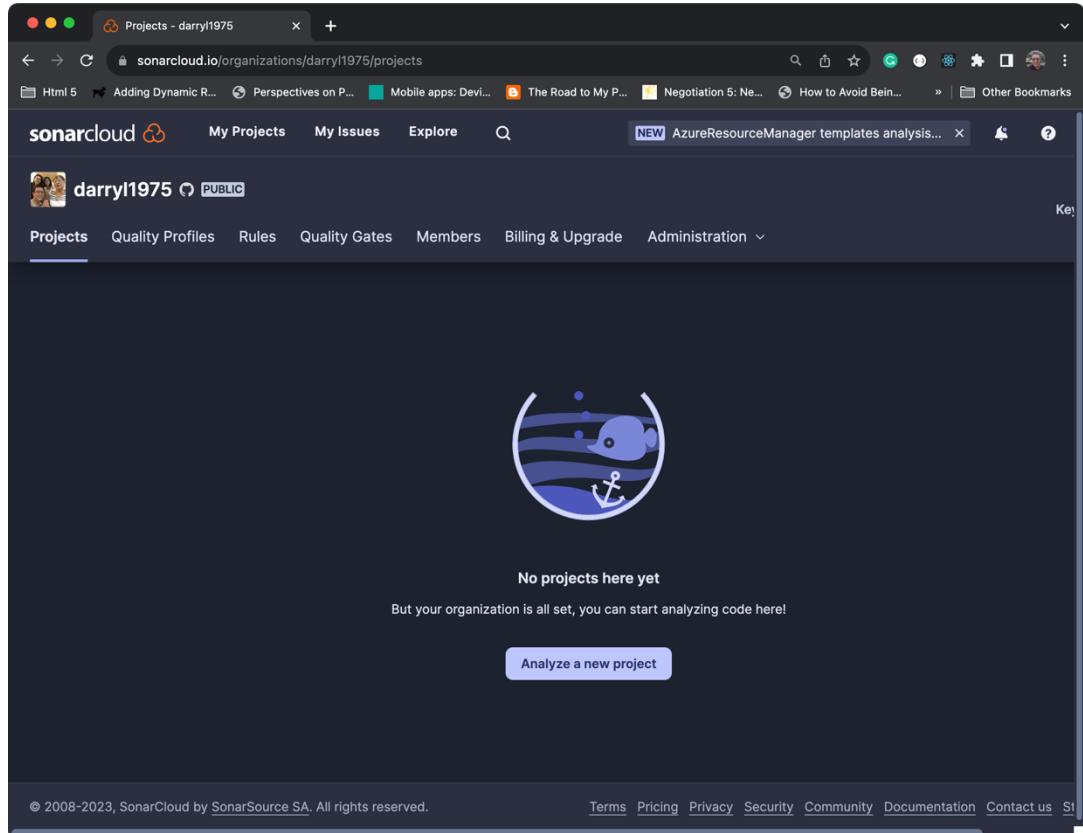
Every CI pipeline is not complete with some form of continuous inspection of code quality. Here, we will integrate SonarQube (open-source platform) for inspection of code quality. Sonarcloud will perform automated code analysis and provides detailed reports on code quality, code coverage, code-related 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



The screenshot shows the SonarCloud 'Projects' page. The URL in the address bar is `sonarcloud.io/projects`. The page has a dark theme. At the top, there are navigation links: 'sonarcloud' (with a cloud icon), 'My Projects' (which is underlined to indicate it's the active tab), 'My Issues', 'Explore', and a search bar. Below the navigation, a message says 'You don't have any favorite projects yet.' followed by a link 'Here is how to add projects to this page'. At the bottom of the main content area, there are three buttons: 'Analyze new project', 'Favorite projects from your orgs', and 'Favorite public projects'. At the very bottom of the page, there is a footer with links: '© 2008-2023, SonarCloud by SonarSource SA, All rights reserved.', 'Terms', 'Pricing', 'Privacy', 'Security', 'Community', 'Documentation', 'Contact us', and a partially visible 'S'.

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



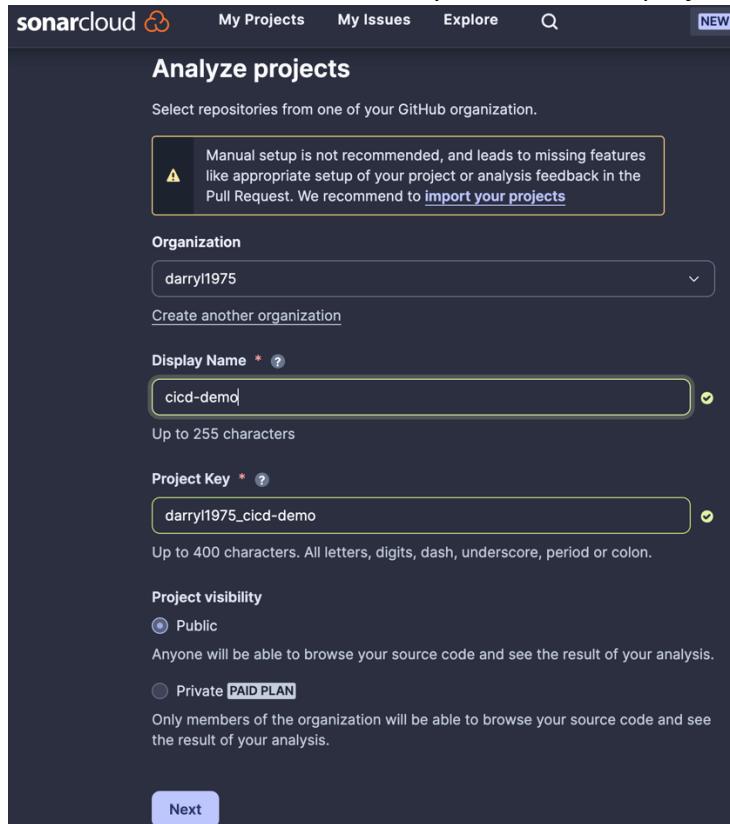
No projects here yet

But your organization is all set, you can start analyzing code here!

Analyze a new project

© 2008-2023, SonarCloud by SonarSource SA. All rights reserved.

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



Manual setup is not recommended, and leads to missing features like appropriate setup of your project or analysis feedback in the Pull Request. We recommend to [import your projects](#)

Organization

darryl1975

Create another organization

Display Name * ?

cicd-demo

Up to 255 characters

Project Key * ?

darryl1975_cicd-demo

Up to 400 characters. All letters, digits, dash, underscore, period or colon.

Project visibility

Public

Anyone will be able to browse your source code and see the result of your analysis.

Private PAID PLAN

Only members of the organization will be able to browse your source code and see the result of your analysis.

Next

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

Set up project for Clean as You Code

The new code definition sets which part of your code will be considered new code.

This helps you focus attention on the most recent changes to your project, enabling you to follow the Clean as You Code methodology.

Learn more: [New Code Definition ↗](#)

Set a new code definition for your organisation to use it by default for all new projects

- This can help you use the Clean as You Code methodology consistently across projects.
`darryl1975 - Administration - New Code`

The new code for this project will be based on:

Previous version

Any code that has changed since the previous version is considered new code.

Recommended for projects following regular versions or releases.

Number of days

Any code that has changed in the last x days is considered new code. If no action is taken on a new issue after x days, this issue will become part of the overall code.

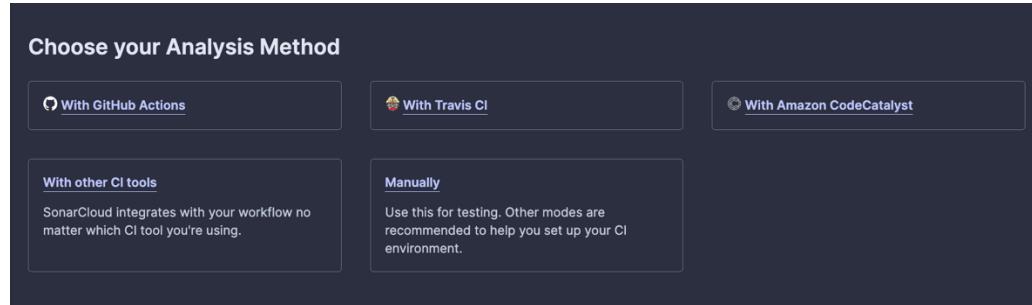
Recommended for projects following continuous delivery.

You can change this at any time in the project administration

[Back](#)

[Create project](#)

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



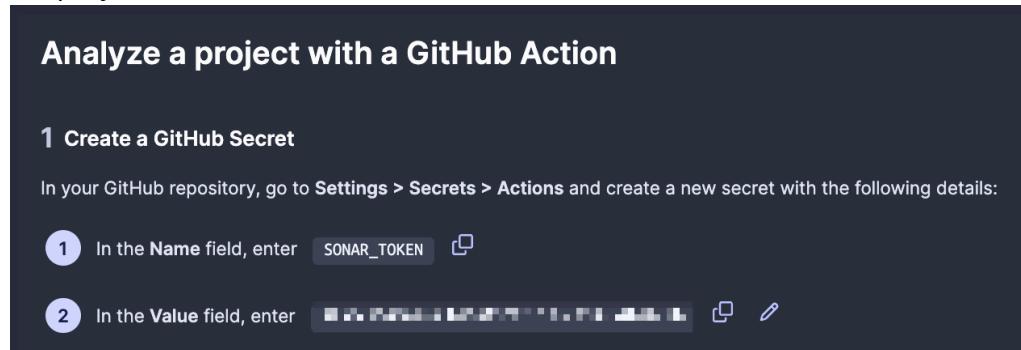
Choose your Analysis Method

- With GitHub Actions
- With Travis CI
- With Amazon CodeCatalyst
- With other CI tools

SonarCloud integrates with your workflow no matter which CI tool you're using.
- Manually

Use this for testing. Other modes are recommended to help you set up your CI environment.

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



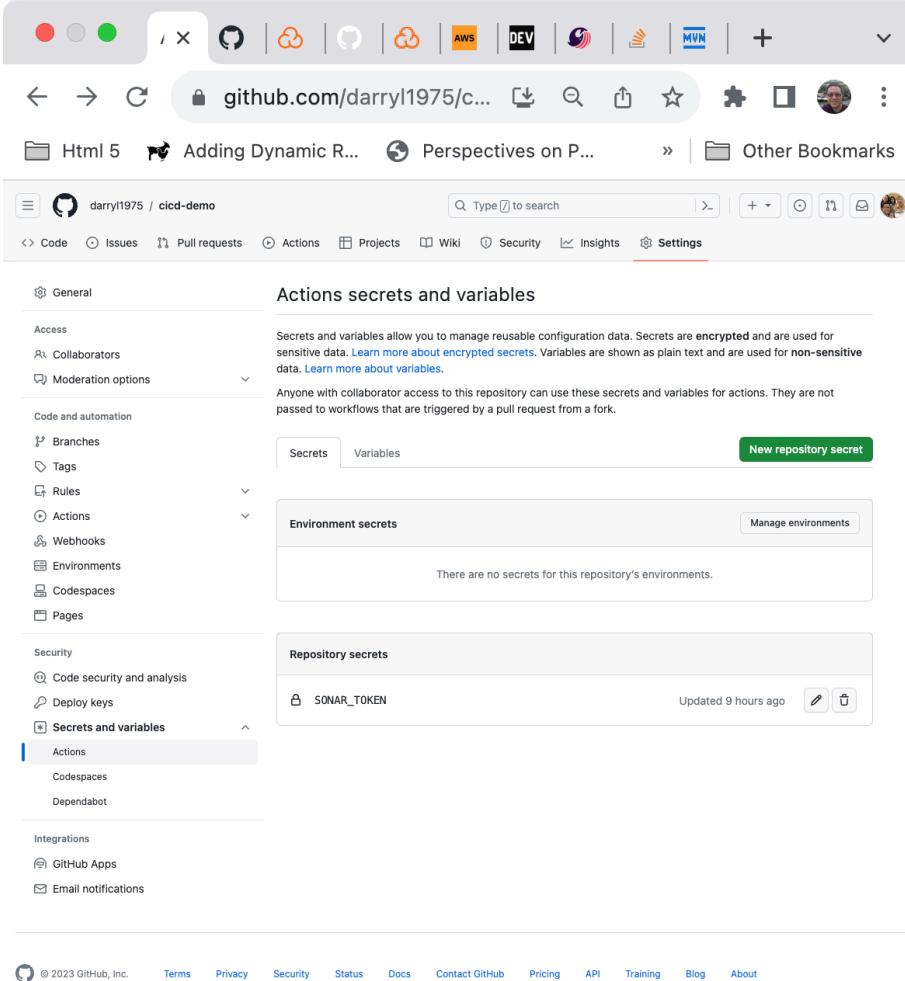
Analyze a project with a GitHub Action

1 Create a GitHub Secret

In your GitHub repository, go to **Settings > Secrets > Actions** and create a new secret with the following details:

- 1 In the **Name** field, enter `SONAR_TOKEN` 
- 2 In the **Value** field, enter   

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

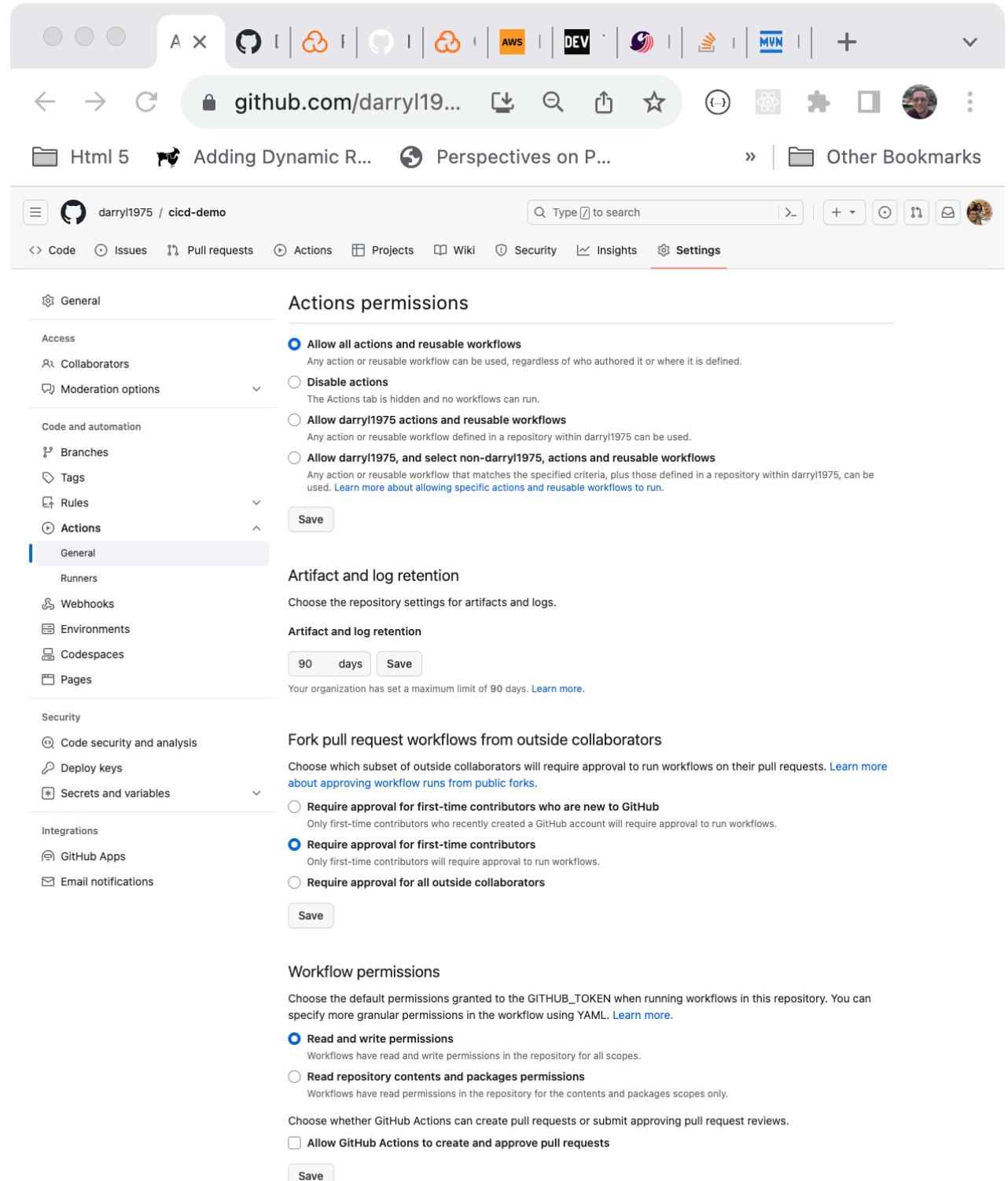


The screenshot shows the GitHub repository settings page for 'darryl1975/cicd-demo'. The 'Actions' tab is selected. On the left, there's a sidebar with sections like General, Access, Code and automation, Security, and Integrations. Under Security, the 'Secrets and variables' section is expanded, showing 'Actions' selected. The main content area is titled 'Actions secrets and variables'. It has tabs for 'Secrets' (selected) and 'Variables'. Under 'Environment secrets', it says 'There are no secrets for this repository's environments.' Under 'Repository secrets', there's one entry: 'SONAR_TOKEN' (updated 9 hours ago), with edit and delete icons. At the bottom, there's a footer with links to Terms, Privacy, Security, Status, Docs, Contact GitHub, Pricing, API, Training, Blog, and About.

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.



The screenshot shows the GitHub repository settings for 'cicd-demo'. The left sidebar shows the 'Actions' section is selected. The main area is titled 'Actions permissions' and contains the following configuration:

- Allow all actions and reusable workflows** (selected): Any action or reusable workflow can be used, regardless of who authored it or where it is defined.
- Disable actions**: The Actions tab is hidden and no workflows can run.
- Allow darryl1975 actions and reusable workflows**: Any action or reusable workflow defined in a repository within darryl1975 can be used.
- Allow darryl1975, and select non-darryl1975, actions and reusable workflows**: Any action or reusable workflow that matches the specified criteria, plus those defined in a repository within darryl1975, can be used. A link to learn more about allowing specific actions and reusable workflows to run is provided.

A 'Save' button is located below this section.

Artifact and log retention: Set to 90 days. A 'Save' button is located below this section.

Fork pull request workflows from outside collaborators: Set to 'Require approval for first-time contributors'. A 'Save' button is located below this section.

Workflow permissions: Set to 'Read and write permissions'. A 'Save' button is located below this section.

10. Modify build workflow to test workflow

```
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.

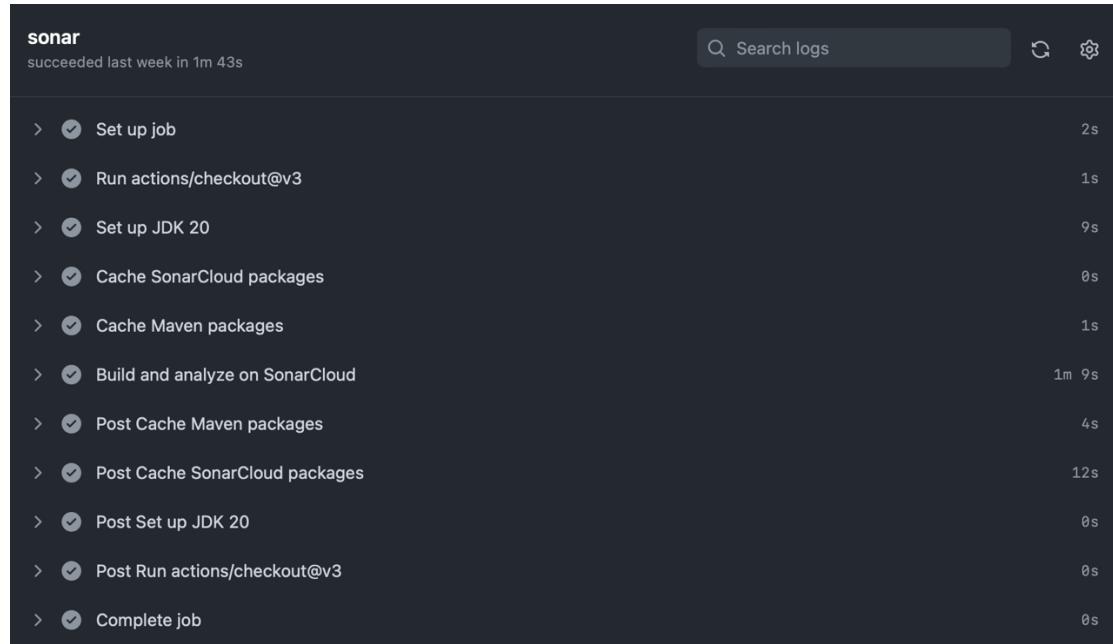
```
sonar:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 20
      uses: actions/setup-java@v3
      with:
        java-version: '20'
        distribution: 'temurin'
        cache: maven
    - name: Cache SonarCloud packages
      uses: actions/cache@v3
      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/>

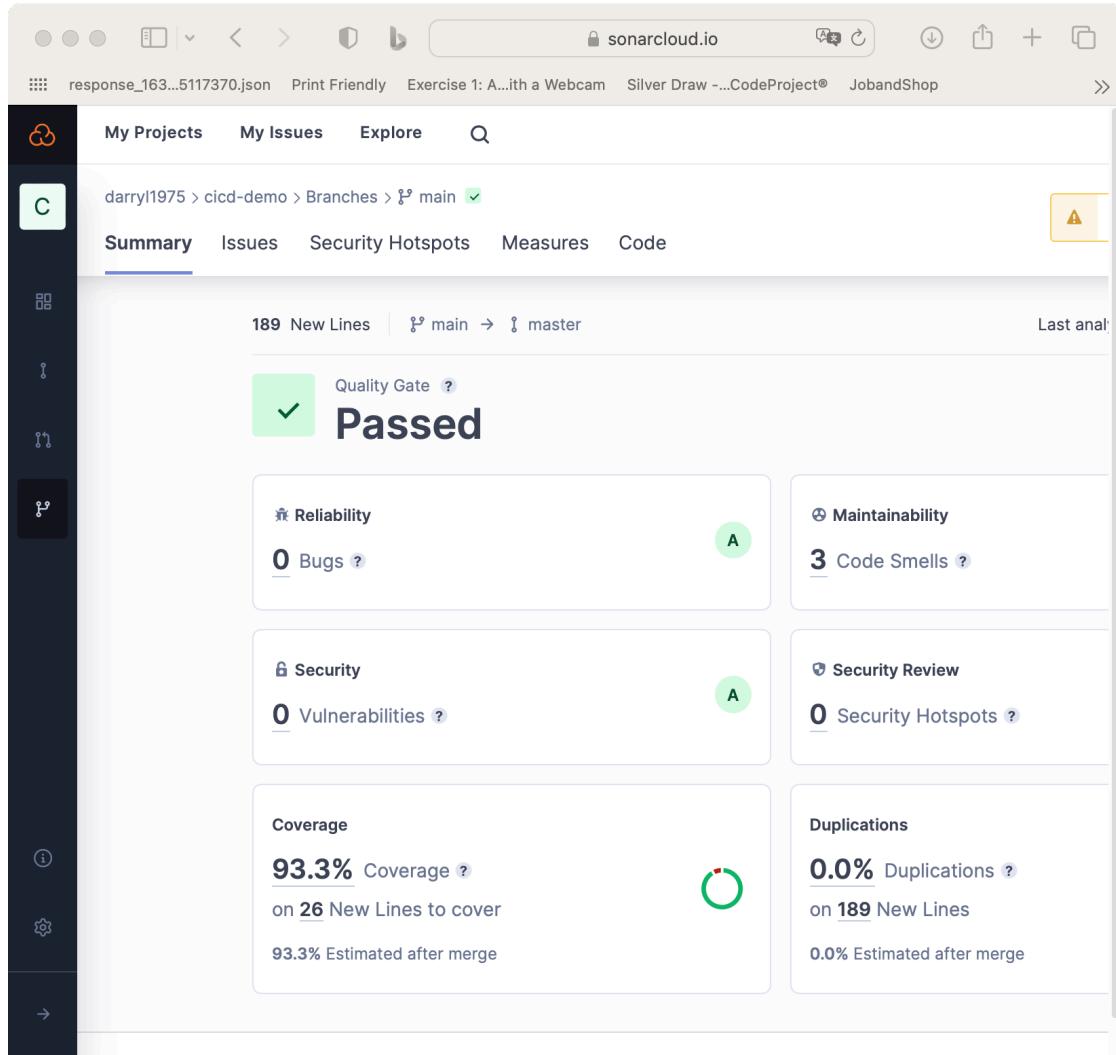
12. Analyse your workflow



The screenshot shows a GitHub Actions workflow log for a job named 'sonar'. The job succeeded last week in 1m 43s. The log details the following steps:

- > ✓ Set up job (2s)
- > ✓ Run actions/checkout@v3 (1s)
- > ✓ Set up JDK 20 (9s)
- > ✓ Cache SonarCloud packages (0s)
- > ✓ Cache Maven packages (1s)
- > ✓ Build and analyze on SonarCloud (1m 9s)
- > ✓ Post Cache Maven packages (4s)
- > ✓ Post Cache SonarCloud packages (12s)
- > ✓ Post Set up JDK 20 (0s)
- > ✓ Post Run actions/checkout@v3 (0s)
- > ✓ Complete job (0s)

13. Analyse your sonar results

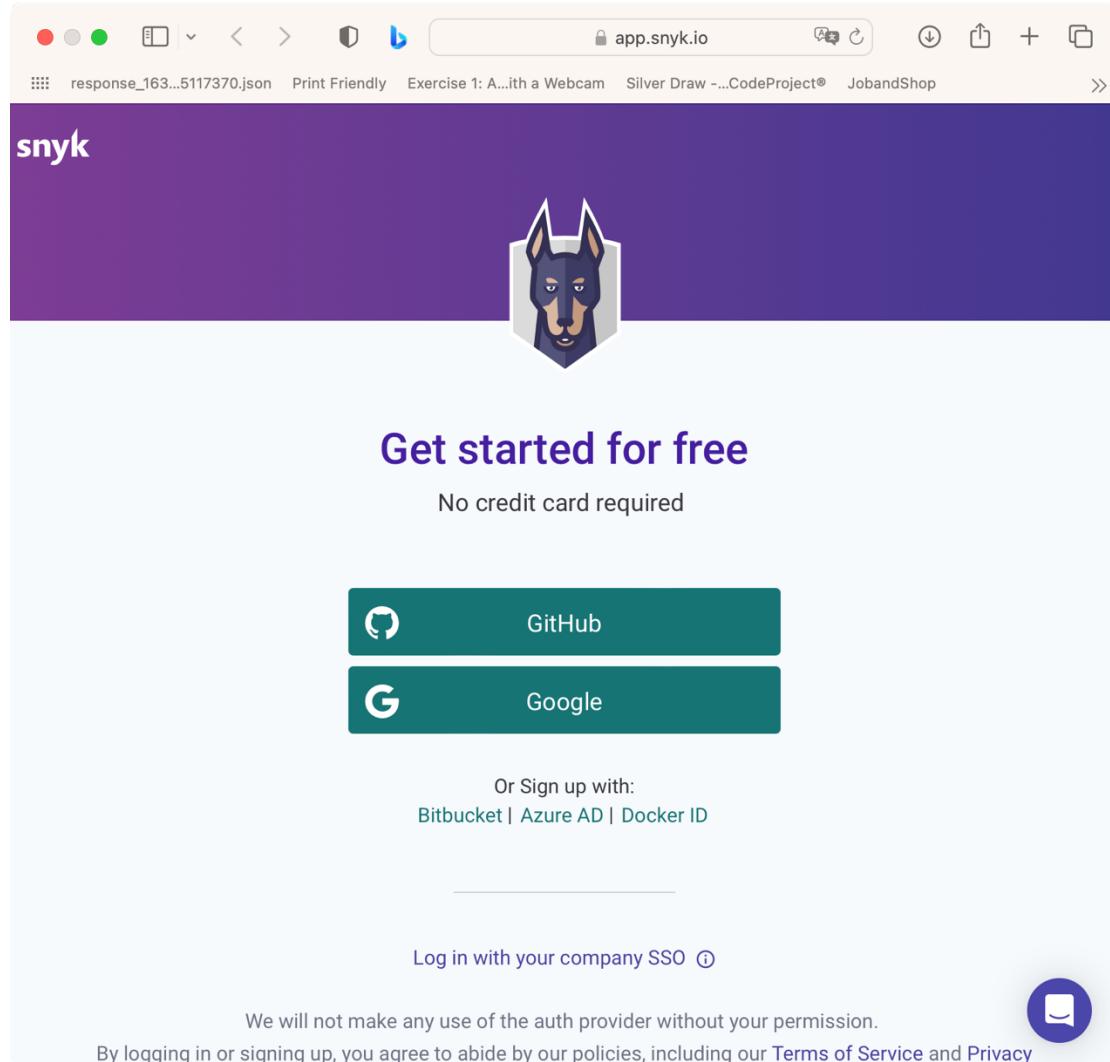


The screenshot shows the SonarCloud.io interface for a project named "darryl1975 > cicd-demo > Branches > main". The "Summary" tab is selected. The analysis summary indicates "189 New Lines" and "main → master". A prominent green "Passed" status is displayed under the "Quality Gate". Below this, six key metrics are shown in cards:

- Reliability:** 0 Bugs (Grade A)
- Maintainability:** 3 Code Smells
- Security:** 0 Vulnerabilities (Grade A)
- Security Review:** 0 Security Hotspots
- Coverage:** 93.3% Coverage (Grade A) - based on 26 New Lines to cover, 93.3% Estimated after merge.
- Duplications:** 0.0% Duplications (Grade A) - based on 189 New Lines, 0.0% Estimated after merge.

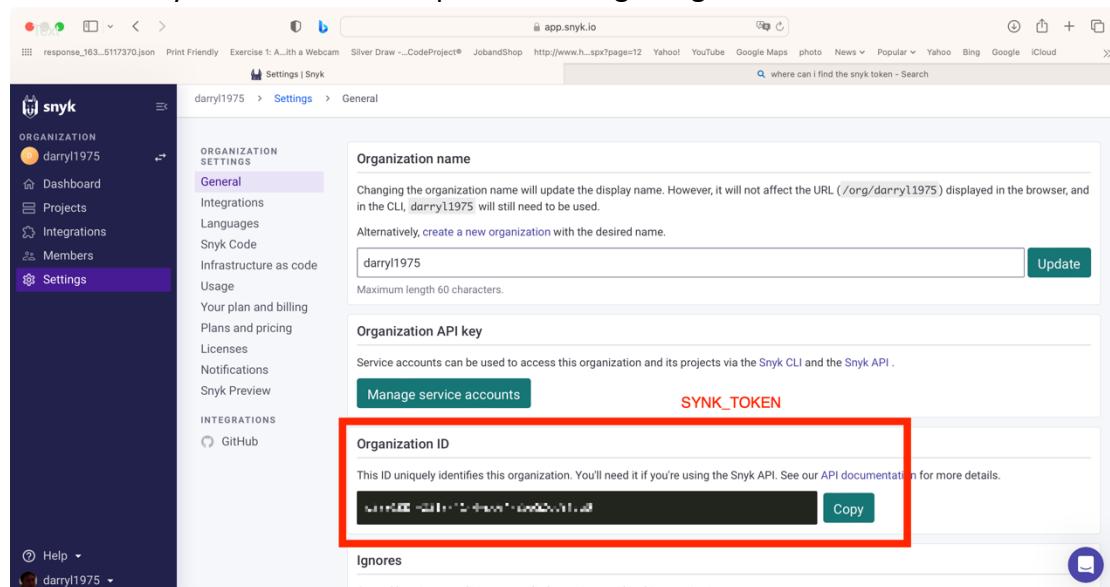
Integrating Snyk into your Github CI/CD (*Estimate: 45 minutes*)

1. Login to your snyk account



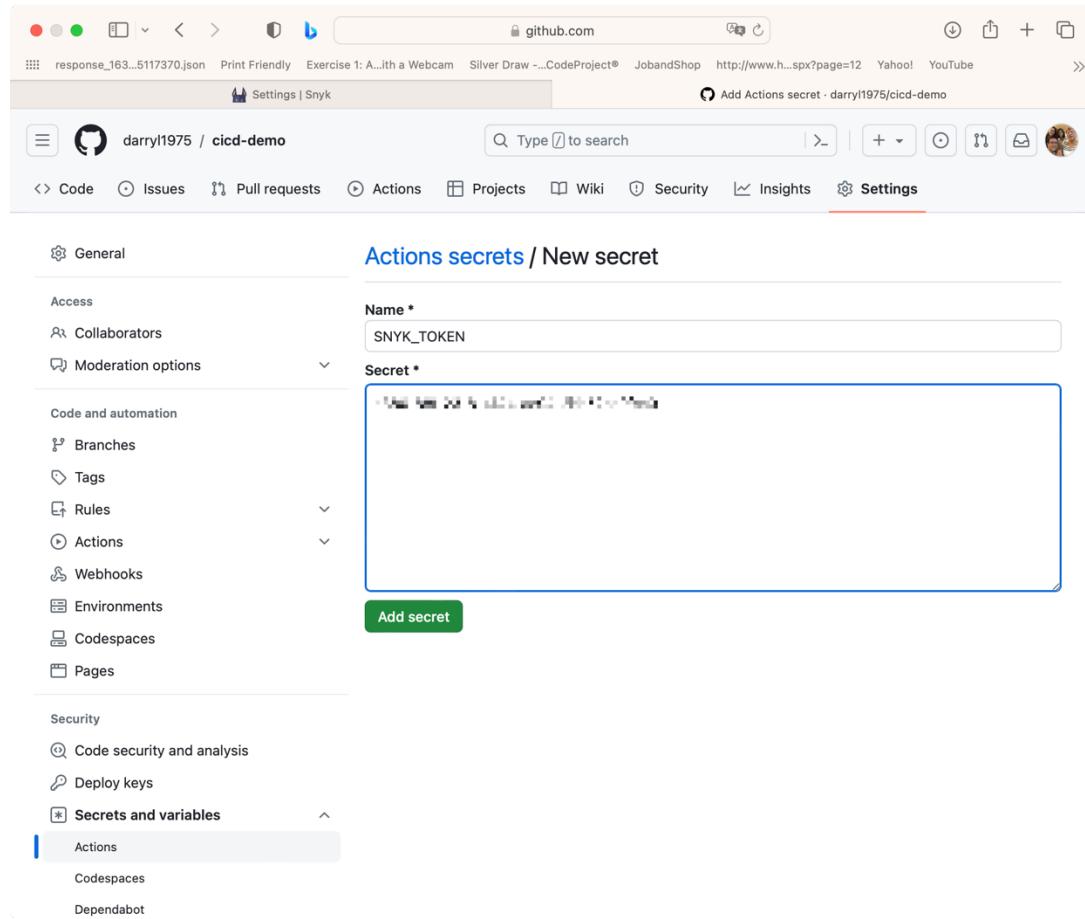
The screenshot shows the Snyk login page with a purple header featuring the Snyk logo (a dog's head inside a box). Below the header, there's a large "Get started for free" button with the subtext "No credit card required". Underneath, there are two prominent sign-in buttons: "GitHub" and "Google". Below these buttons, there's a "Or Sign up with:" section listing "Bitbucket | Azure AD | Docker ID". Further down, there's a "Log in with your company SSO" link with a blue icon. At the bottom of the page, there's a note about permission and links to "Terms of Service" and "Privacy".

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



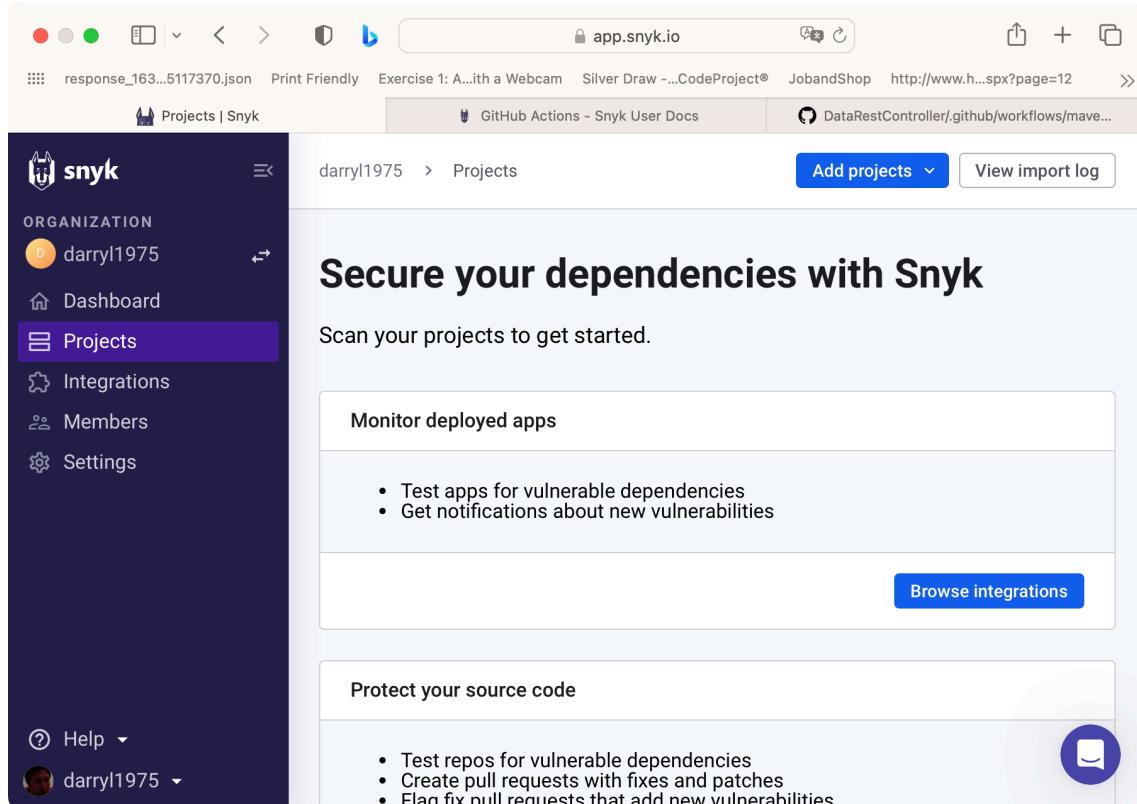
The screenshot shows the Snyk organization settings page for the user "darryl1975". The left sidebar has a "Settings" tab selected. The main area shows "Organization name" set to "darryl1975" with an "Update" button. Below it is the "Organization API key" section with a "Manage service accounts" button and a "SYNK_TOKEN" field. The "Organization ID" field at the bottom is also highlighted with a red box. A "Copy" button is visible next to the Organization ID field.

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



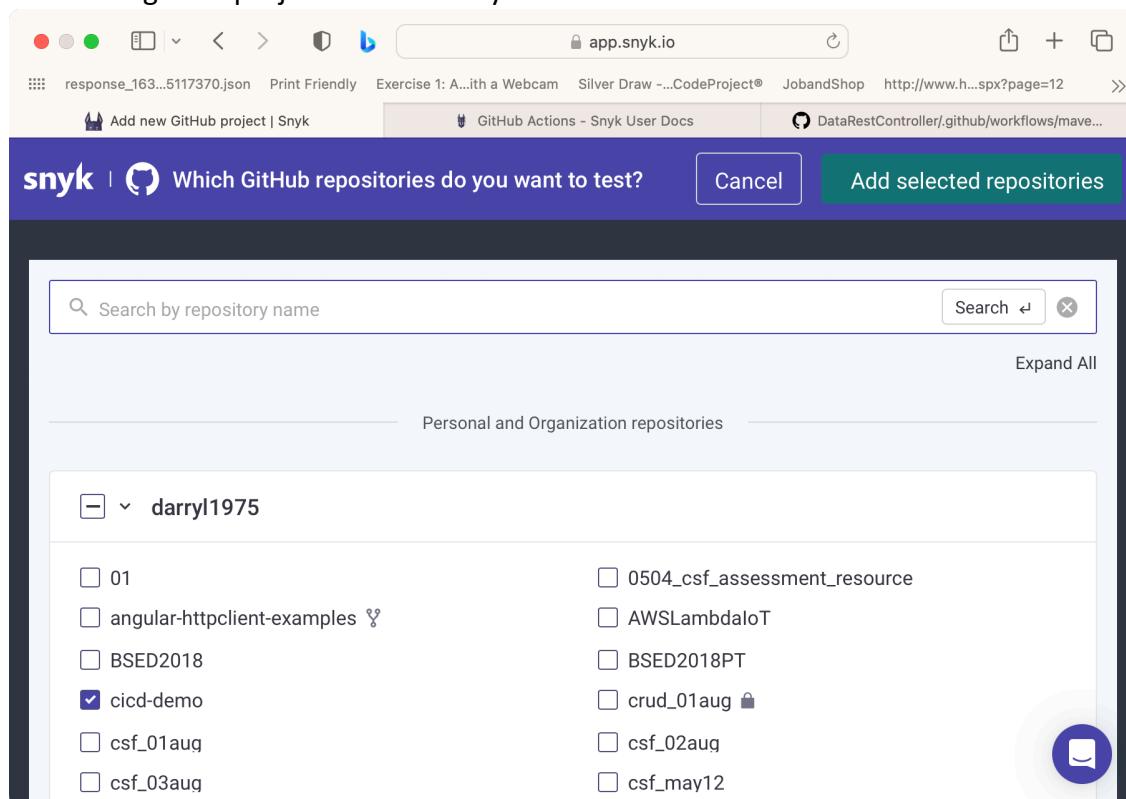
The screenshot shows the GitHub Actions secrets configuration page for a repository named 'cicd-demo'. The 'Actions' tab is selected in the navigation bar. On the left, a sidebar lists various GitHub features like General, Access, Collaborators, and Secrets and variables. Under 'Secrets and variables', the 'Actions' section is expanded, showing a list of secrets: 'Actions', 'Codespaces', and 'Dependabot'. A new secret is being added, with the name 'SNYK_TOKEN' and a long, obscured secret value. A green 'Add secret' button is at the bottom right of the input field.

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



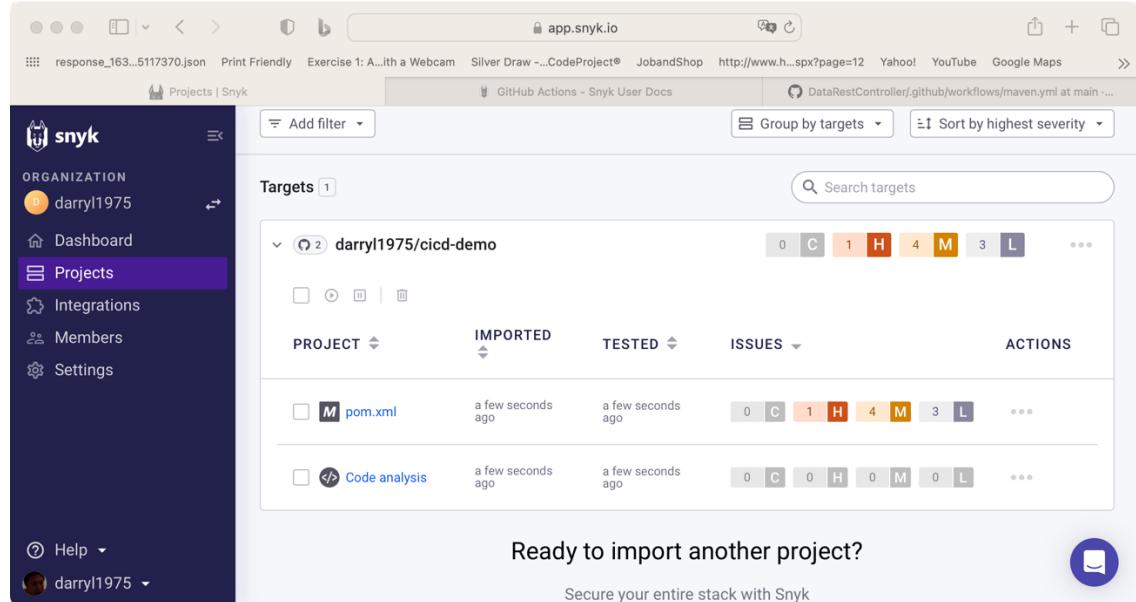
The screenshot shows the Snyk web interface. On the left, there's a sidebar with options like 'ORGANIZATION', 'darryl1975', 'Dashboard', 'Projects' (which is selected and highlighted in purple), 'Integrations', 'Members', and 'Settings'. Below that are 'Help' and a user dropdown for 'darryl1975'. The main content area has a heading 'Secure your dependencies with Snyk' and a sub-section 'Scan your projects to get started.' It includes two boxes: 'Monitor deployed apps' (with bullet points 'Test apps for vulnerable dependencies' and 'Get notifications about new vulnerabilities') and 'Protect your source code' (with bullet points 'Test repos for vulnerable dependencies', 'Create pull requests with fixes and patches', and 'Flag fix pull requests that add new vulnerabilities'). At the top of the main content area, there are tabs for 'GitHub Actions - Snyk User Docs' and 'DataRestController/github/workflows/maven...'. A blue 'Add projects' button is located at the top right of the main content area.

5. Select the github project to add to snyk.



The screenshot shows the 'Add new GitHub project' screen. At the top, there's a search bar with 'Search by repository name' and a 'Cancel' button. Below it, a list of repositories under 'Personal and Organization repositories' is shown. The repository 'darryl1975' is expanded, showing its contents. The repository 'cicd-demo' is checked with a blue checkmark. Other repositories listed include '01', 'angular-httpclient-examples', 'BSED2018', 'csc_01aug', 'csc_03aug', '0504_csf_assessment_resource', 'AWSLambdaT', 'BSED2018PT', 'crud_01aug', 'csc_02aug', and 'csc_may12'. A green 'Add selected repositories' button is located at the top right of the list area. A blue message icon is visible on the right side of the list.

6. Ensure your project has been added and scanned.



The screenshot shows the Snyk web interface. On the left, there's a sidebar with options: Organization (darryl1975), Dashboard, Projects (selected), Integrations, Members, Settings, Help, and a user profile (darryl1975). The main area is titled 'Targets [1]' and shows one target: 'darryl1975/cicd-demo'. This target has two items: 'pom.xml' and 'Code analysis'. The 'pom.xml' item was imported 'a few seconds ago' and tested 'a few seconds ago', showing 1 High (H) severity issue. The 'Code analysis' item was imported and tested 'a few seconds ago', showing 0 issues. There are filters for 'Add filter', 'Group by targets', and 'Sort by highest severity'. A message at the bottom says 'Ready to import another project?' with a 'Secure your entire stack with Snyk' button.

7. Add the Snyk configuration to your git workflow.

```
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

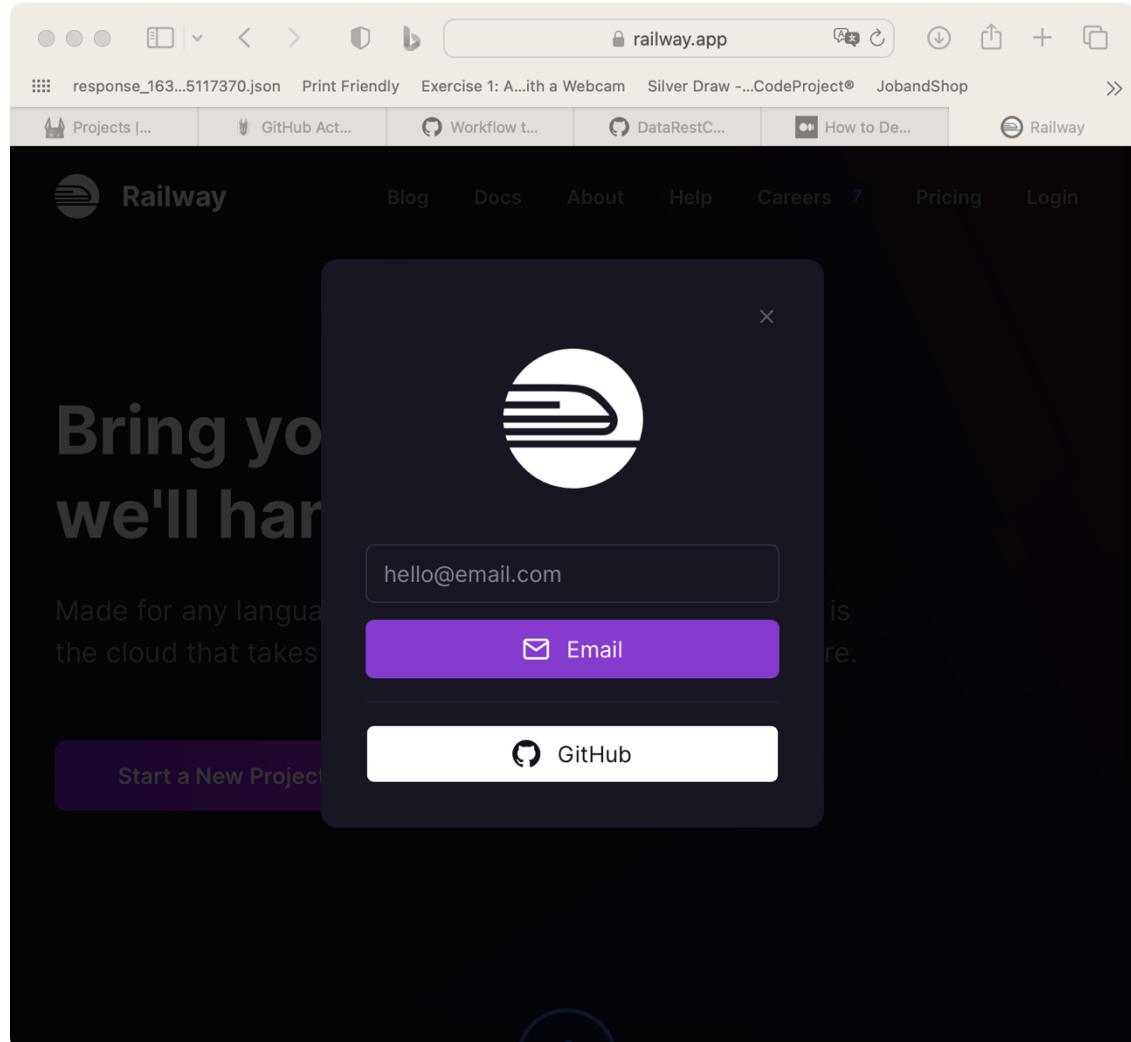
Deploying Spring Boot application to Railway (*Estimate: 30 minutes*)
 (Optional: For information & references only.)

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

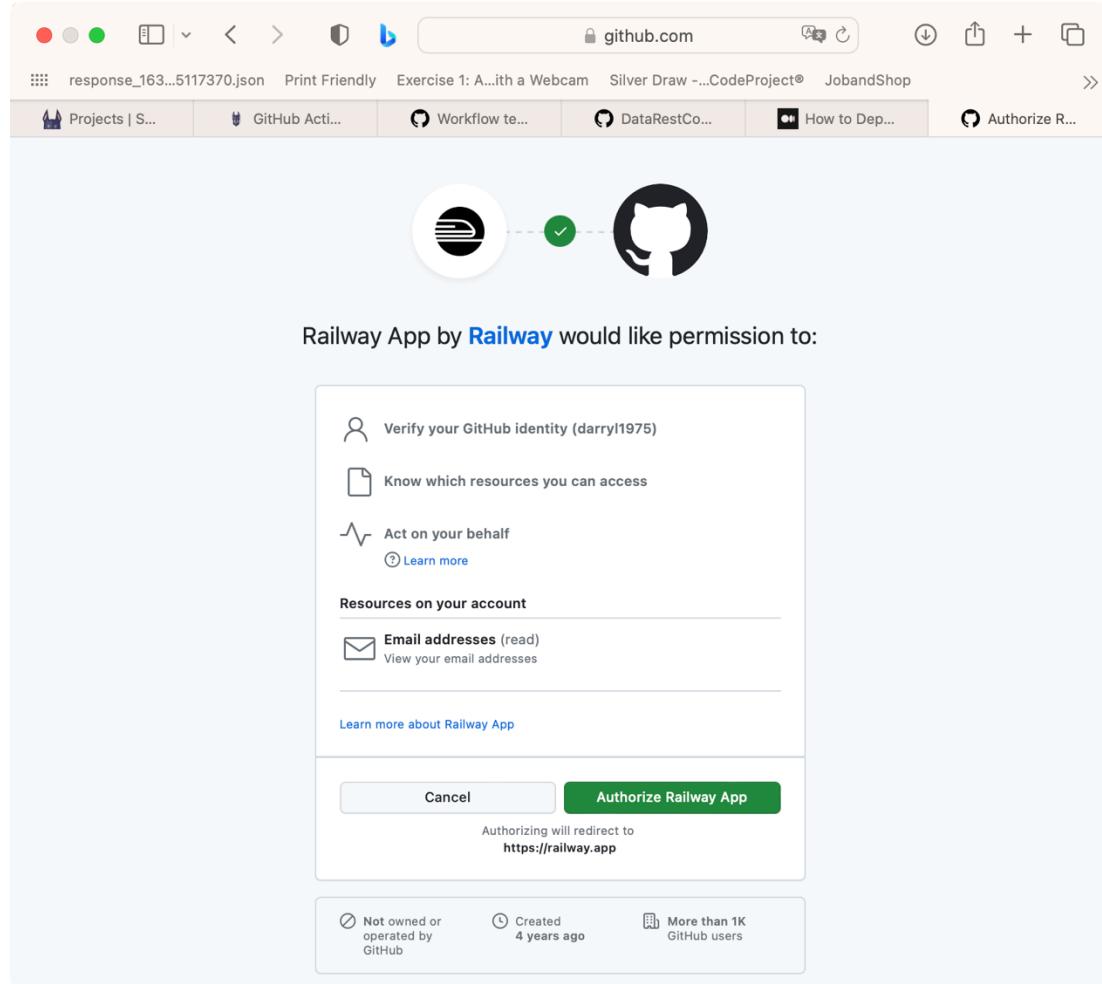
6. Create a Railway.app account
7. Connect Railway.app to your github account app

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.

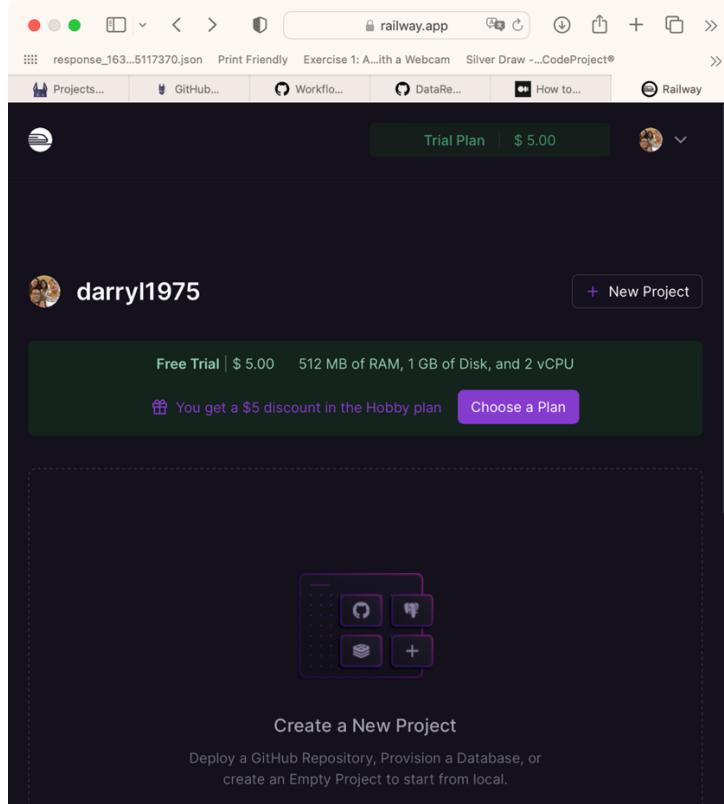
1. Login Railways using Github account.



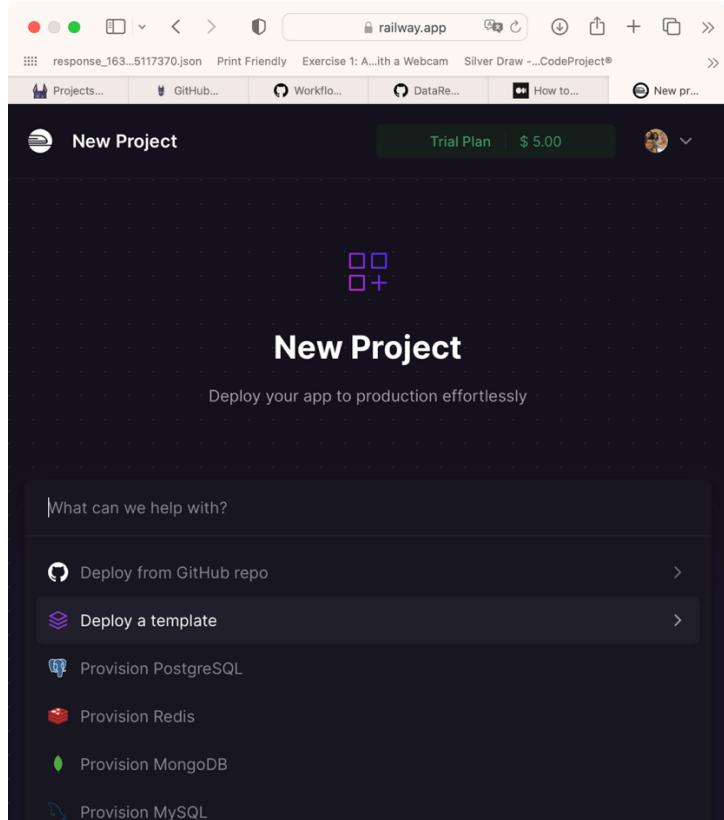
2. If you are logging in via Github (recommended), authorize Railway app.



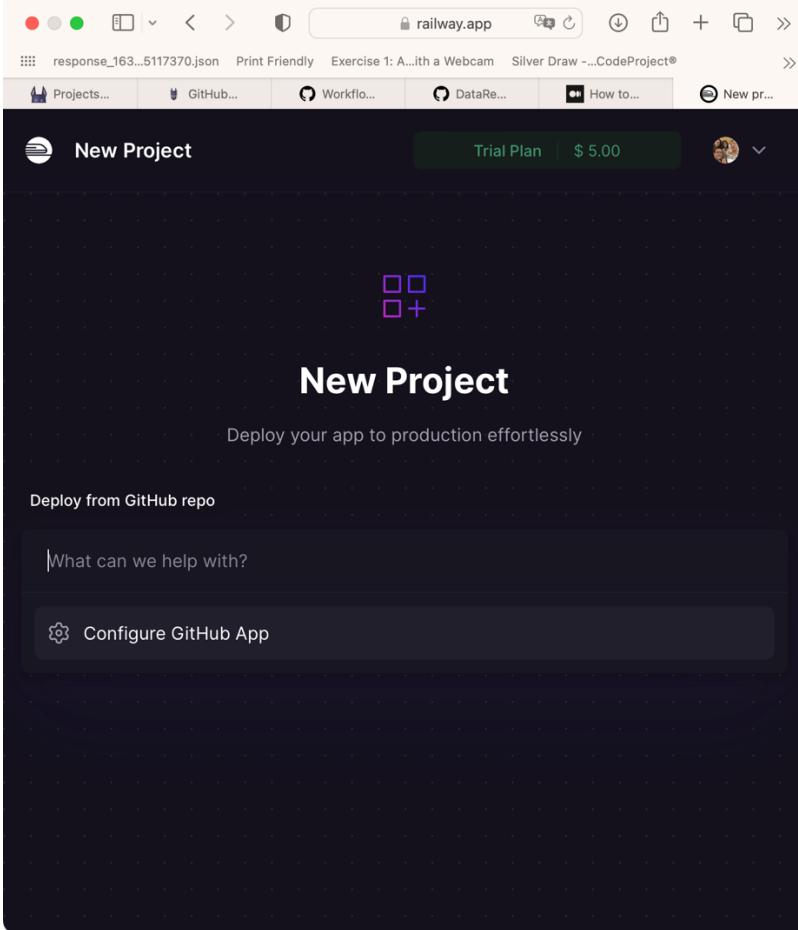
3. Create a **New Project** on Railway.



4. Deploy from Github Repo.

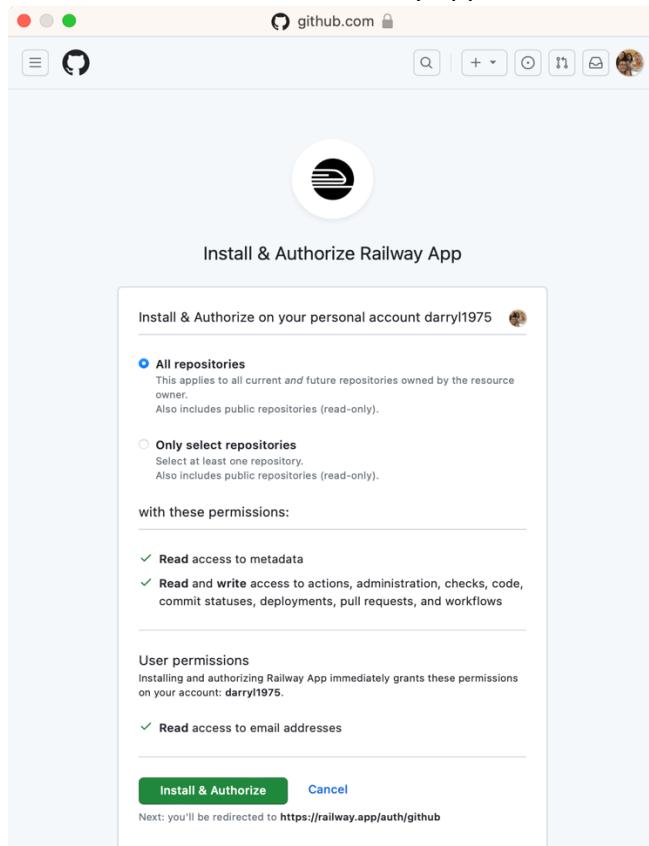


5. **Configure GitHub App** on Railway.



The screenshot shows the Railway app's 'New Project' screen. At the top, there are several tabs: Projects..., GitHub..., Workflo..., DataRe..., How to..., and New pr.... Below the tabs, there are three small icons: a square with two smaller squares inside, a square with a plus sign, and another square with a plus sign. The main title 'New Project' is displayed prominently. Below it, the sub-instruction 'Deploy your app to production effortlessly' is shown. A button labeled 'Deploy from GitHub repo' is visible. A search bar contains the placeholder text 'What can we help with?'. A dropdown menu is open, showing the option 'Configure GitHub App'.

6. Install and authorize Railway app.



Install & Authorize Railway App

Install & Authorize on your personal account darryl1975

All repositories
This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories
Select at least one repository.
Also includes public repositories (read-only).

with these permissions:

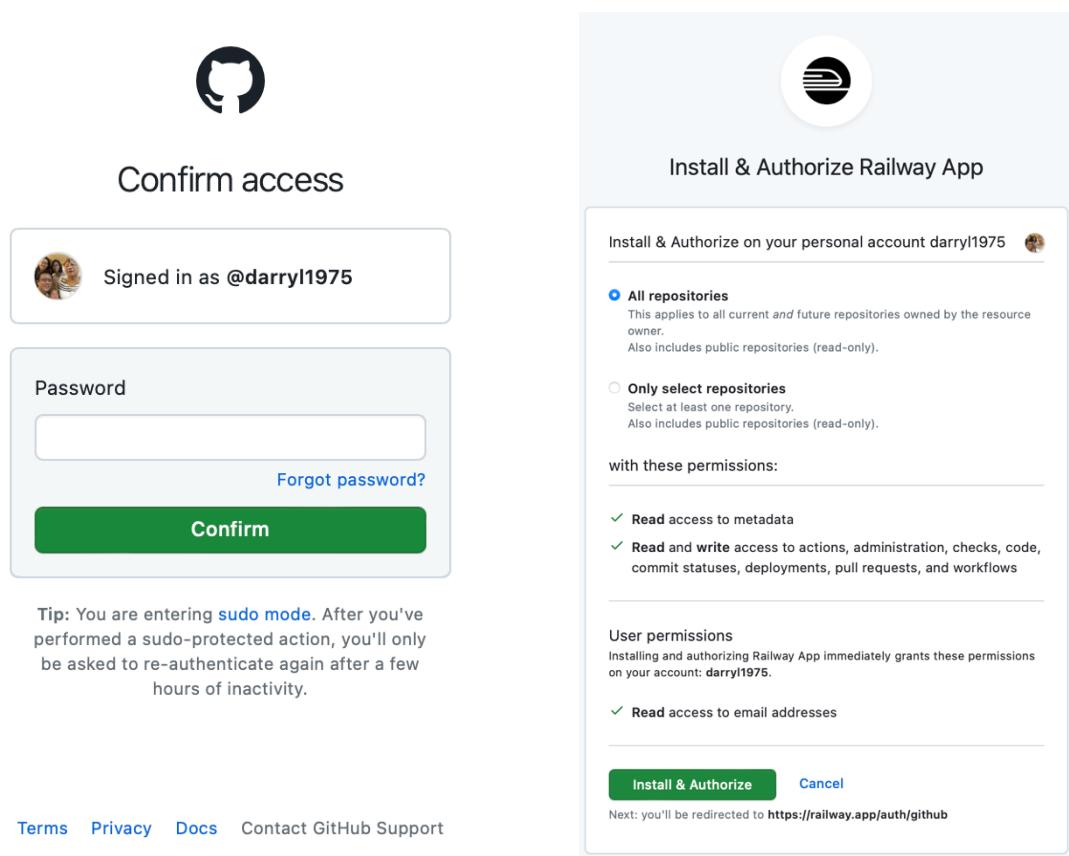
- Read access to metadata
- Read and write access to actions, administration, checks, code, commit statuses, deployments, pull requests, and workflows

User permissions
Installing and authorizing Railway App immediately grants these permissions on your account: darryl1975.

- Read access to email addresses

Install & Authorize **Cancel**

Next: you'll be redirected to <https://railway.app/auth/github>



Confirm access

 Signed in as @darryl1975

Password

[Forgot password?](#)

[Confirm](#)

Tip: You are entering **sudo mode**. After you've performed a sudo-protected action, you'll only be asked to re-authenticate again after a few hours of inactivity.

Install & Authorize Railway App

Install & Authorize on your personal account darryl1975

All repositories
This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories
Select at least one repository.
Also includes public repositories (read-only).

with these permissions:

- Read access to metadata
- Read and write access to actions, administration, checks, code, commit statuses, deployments, pull requests, and workflows

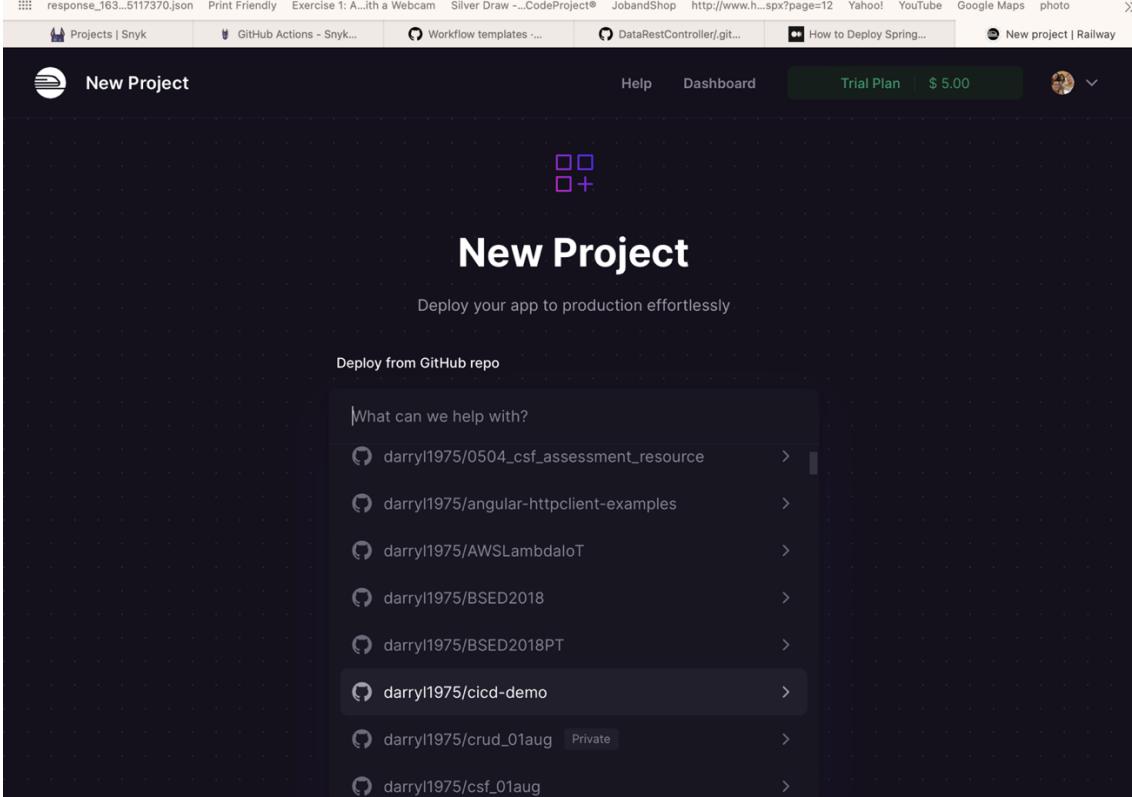
User permissions
Installing and authorizing Railway App immediately grants these permissions on your account: darryl1975.

- Read access to email addresses

Install & Authorize **Cancel**

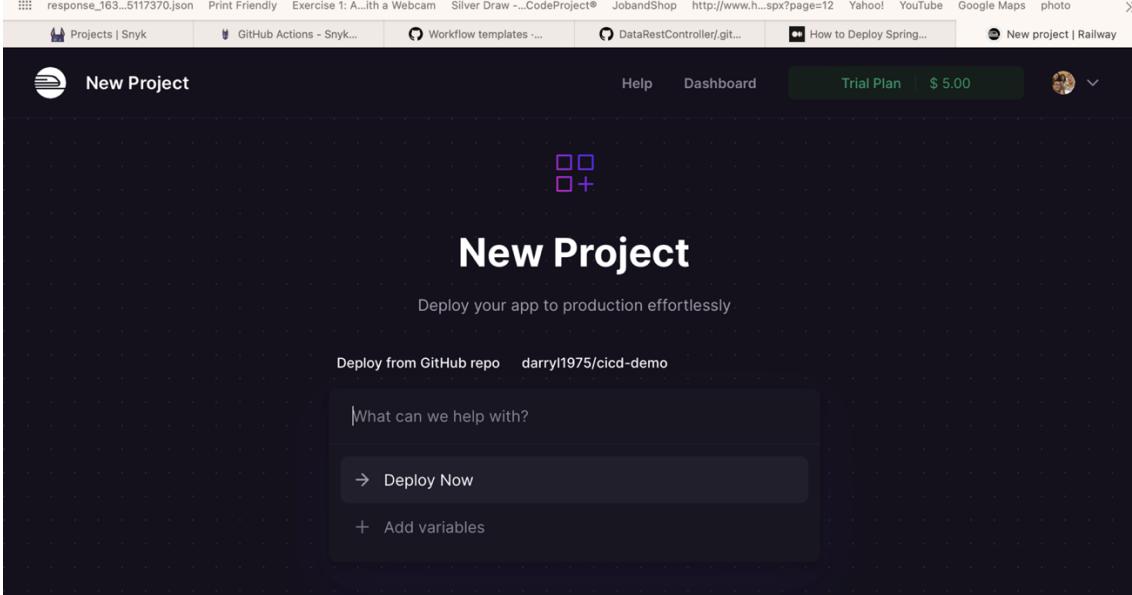
Next: you'll be redirected to <https://railway.app/auth/github>

7. Choose your project on Railway.



The screenshot shows the Railway.app interface for creating a new project. At the top, there's a navigation bar with links like 'Help', 'Dashboard', 'Trial Plan (\$ 5.00)', and a user profile icon. Below the navigation is a large 'New Project' button with a plus sign icon. The main area is titled 'New Project' with the subtitle 'Deploy your app to production effortlessly'. A section titled 'Deploy from GitHub repo' lists several GitHub repositories owned by 'darryl1975': '0504_csf_assessment_resource', 'angular-httpclient-examples', 'AWSLambdaIoT', 'BSED2018', 'BSED2018PT', 'cicd-demo' (which is highlighted), 'crud_01aug' (marked as 'Private'), and 'csf_01aug'. Each repository has a small circular profile picture and a right-pointing arrow.

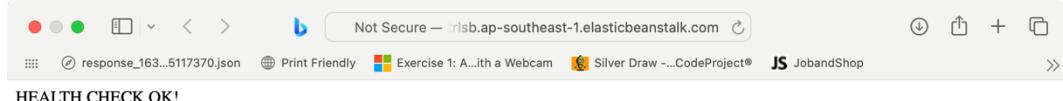
8. Deploy your project onto Railway.



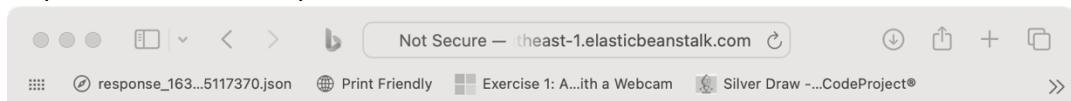
This screenshot shows the same Railway.app interface after selecting the 'cicd-demo' repository. The repository name 'darryl1975/cicd-demo' is now displayed next to the 'Deploy from GitHub repo' label. Below the repository list, there's a search bar with the placeholder 'What can we help with?' and two buttons: 'Deploy Now' (highlighted with a dark background) and '+ Add variables'.

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

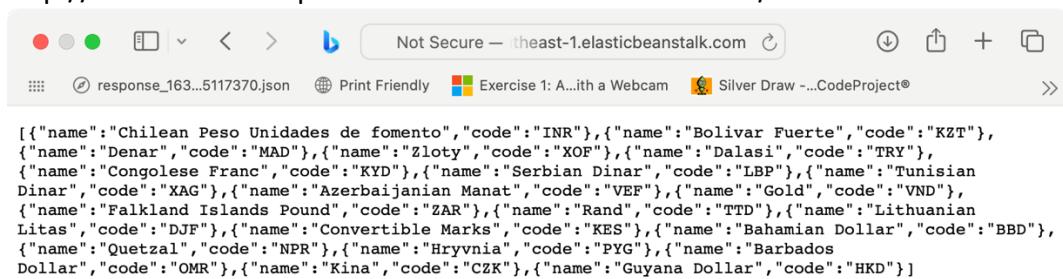
<http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/>



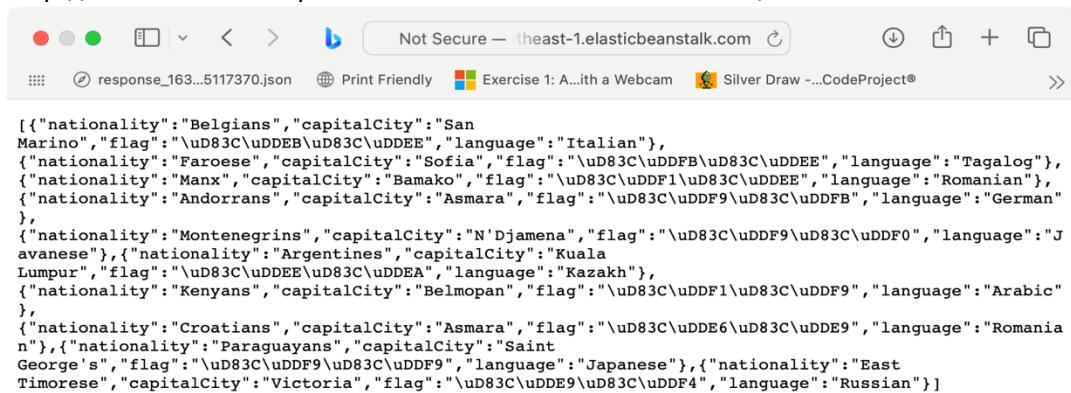
<http://datarestctrlsb.ap-southeast-1.elasticbeanstalk.com/version>



<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-dempp-app

This command will tell Docker to run the container and forward the exposed 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