2023

Automated Security Testing in DevSecOps Pipelines

RESEARCH REPORT

NEDELCHEV,TSANKO T.N.

ABSTRACT

This research report focuses on the integration of automated security testing in DevSecOps pipelines to ensure continuous security throughout the software development lifecycle. Traditional manual security testing methods face challenges in DevOps environments due to the fast-paced nature of software development, necessitating the adoption of automated approaches. The report conducts a comprehensive literature review to examine existing studies and tools for automated security testing in DevSecOps. Furthermore, explores various techniques such as static analysis, dynamic analysis, software composition analysis, and penetration testing, assessing their benefits and limitations. The report continues with an evaluation and selection of suitable tools and frameworks. It proposes an architecture for integrating automated security testing and presents the implementation of a sample DevSecOps pipeline using selected security testing tools. The results demonstrate the effectiveness of automated security testing in identifying vulnerabilities and security weaknesses compared to manual methods. The report discusses the advantages of integrating automated security testing in DevSecOps pipelines, addressing practical challenges and offering potential solutions. It provides best practices and guidelines for tool selection, defining security test cases, and integrating security testing into CI/CD processes. The report concludes by summarizing the findings, emphasizing the significance of continuous security testing in DevSecOps, and outlining future research directions. The outcomes of this research contribute to enhancing the understanding and adoption of automated security testing practices in DevSecOps, and in result improving software security in agile development environments.

Contents

1. INTRODUCTION

In today's digital realm, a secure software application is an absolute requirement. With the ever-increasing cyber threats, organizations now fully acknowledge the significance of robust security measures at every stage of software development. This has given rise to a methodology called DevSecOps, which integrates security practices seamlessly into development and operations processes. DevSecOps signifies a change in software development, highlighting the importance of considering security from the beginning rather than treating it as an afterthought. It acknowledges that traditional security approaches, which relied on manual testing by dedicated security teams, are insufficient in the fast-paced DevOps environment. This means that in the DevSecOps process every developer working on the software is responsible for the security of the software.

As organizations strive to deliver software faster and more frequently, maintaining adequate security measures has become increasingly complex. With the rapid development cycles and continuous deployment practices of DevOps, traditional manual security testing struggles to keep up. Manual testing is time-consuming, prone to errors, and often lacks comprehensive coverage. Consequently, critical vulnerabilities may go undetected until it's too late, leaving applications and users exposed to potential breaches and data leaks. To tackle these challenges, automated security testing has emerged as a crucial element of DevSecOps. By utilizing automation tools and techniques, organizations can improve the speed, accuracy, and coverage of security testing within their development pipelines. To ensure the effectiveness of automated security testing, however, it is vital to identify the key vulnerabilities and threats that require specific attention.

This research investigates the primary security weaknesses and risks that automated security testing should concentrate on in DevSecOps pipelines. The goal is to explore effective methods of integrating automated security testing into DevSecOps pipelines to guarantee consistent security across the software development lifecycle.

The report starts by stating the research objective in the form of a main question and related sub-questions. These questions define the research scope, covering various topics and processes to ensure the research's quality, comprehensiveness, and direction.

The main part of the report will present the findings from the literature review and the results obtained from the tasks and tests performed during the research. It will also detail the setup of the testing environment utilized. Additionally, a series of steps and explanations will be provided to describe the process that led to the findings, including the various technologies involved.

In its conclusion the report will remark on all the results of the research as well as the findings and will explain how that answers the goal/main research question.

2. METHODOLOGY

Main research question:

How can automated security testing be effectively integrated into DevSecOps pipelines to ensure continuous security throughout the software development lifecycle?

Sub-questions with methodology:

- **What are the key security operations that automated security testing can focus on within DevSecOps pipelines?**
    - Library (Best Good and Bad practices)
      Figuring out common practices when it comes to automated testing.
    - Field (Task analysis)
      Gain a proper understanding of the tasks that automated testing seeks to complete.
- **What are the measurable benefits of integrating automated security testing into DevSecOps pipelines, such as improved vulnerability detection and reduced time to remediate?**
    - Library(Literature study)
      Find general information on the benefits of automated security testing.
    - Library(available product analysis)
      Find out what others who implemented automated testing found to be beneficial about it.
    - Lab(Security test)
      Extrapolate from the testing that has been done for this research to the potential benefits
- **What criteria should be considered when selecting automated security testing tools for integration into DevSecOps pipelines?**
    - Showroom(Product review)
      Evaluate the technology in order to determine if it’s valuable for the use case that it is looked up for.

   - Workshop(Multi-criteria decision making)

     Which criteria is more important to consider for each type of automated testing and which tool is the best overall.

- **How to setup and integrate the chosen tools in the DevOps process for consistent and continuous security checks?**
   - Library (Available product analysis)

     Find out what levels of automation do the tools offer and how they function.
   - Field (Document analysis)

     Read different tools' documentation to better understand their function.
   - Workshop (Prototyping)

     Use the available technologies in order to better evaluate them

3. RESULTS AND FINDINGS

Showcasing the findings of the literature study and the results derived from the tasks and tests performed throughout the research.

What are the key security operations that automated security testing can focus on within DevSecOps pipelines?

Code Quality (SAST)

Automation is a fundamental aspect of the DevOps process, and when security is integrated throughout, it becomes the DevSecOps process. In this context, the DevSecOps process can be viewed as augmenting each level of the DevOps process with security features. The initial stage of the DevOps process involves pushing software code to a git repository. Consequently, it is logical to consider static application security testing (SAST) as the first type of security automation.

Static application security testing involves scanning the application code to identify potential issues, including security vulnerabilities. Those issues include vulnerability detection, early bug identification and enforcing secure coding practices among others. The process of static code analysis can begin even in the IDE by installing a real-time developer SAST tool such as Sonar Lint or Code Sight. SAST does not even require a complete application in order to start working. After installing the required tool a developer can just continue the development process as usual but now they will be notified in real time about the mistakes they are making in their code.

Static application security testing is also available as a complete application scanner tool that is also easy to integrate into a CI/CD pipeline. After a simple installation the tools can be configured to perform a scan of the application's code on each push to the git repository. Furthermore some tools can be configured to require a manual check by a developer before letting the pipeline continue. This adds extra layer of security but might also slow down the process as it eliminates some of the advantages of the automation of the process. (9)

During the Building process (Unit Tests)

Unit testing is the process of testing down to the different structures like functions, methods or classes of the code. They are predefined set of tests that serve as a verification whether each unit of code that is being tested works as intended. There are different methodologies that allow unit testing to be integrated seamlessly into the DevSecOps process such as test driven design or TDD. TDD is the practice of writing the automated test before writing the code.

1. **Test**: The TDD process starts by writing test cases that specify the expected behavior of the code. These tests are created using a suitable testing framework for the specific programming language or platform. Initially, the tests are designed to because since the corresponding code has not been implemented yet.

2. **Code**: Once the test cases are in place, the developer focuses on writing the minimum amount of code necessary to make the tests pass. The code is implemented to fulfill the requirements specified in the unit tests. It is crucial to only focus on making the tests pass and avoid adding any extra code beyond what is required.
3. **Refactor and Repeat**: After successfully passing the tests, the code can be refactored. Refactoring involves restructuring the code without changing its external behavior. This step is important for improving the code's design, readability, and maintainability. The goal is to ensure that the code remains clean, efficient, and well-structured. Once the refactoring is done, the cycle repeats by creating additional test cases that define the next desired functionality.

Unit tests can easily be ran through the pipeline as part of the DevSecOps automation. Usually this is the step before building the application in the pipeline. (8)

Software Composition Analysis (SCA)

Open source libraries are a part of most applications today, however, using open source code can easily introduce a lot of vulnerabilities to the software. This is why SCA testing is an important part of the DevSecOps process. SCA is the process of automatically analyzing and managing open source and third-party software that are used in the application. This is performed in order to check if an outdated version of the third party software is used or maybe a version that has a known vulnerability in it. The npm package provides some build in SCA functionality which analyzes the application's dependencies and it comes preinstalled.

The SCA tool first scans the software composition in question and defined the third party software. Consecutively, it provides information about each module including the license under which the software is distributed, whether the software requires any conditions to be met before using it and whether the license is compliant with the licenses that are allowed in the organization or if any such restrictions are present in the first place. Finally the SCA software ties to identify any vulnerabilities inside the third party software that are either the result of an outdated version being used (in which case the SCA software offers to update it to it's latest version) or the software has a known vulnerability in which case the SCA might offer a solution to the problem. (10)

Dynamic application security testing (DAST)

Dynamic Application Security Testing (DAST) is a broad term that encompasses various techniques and tools used to assess the security of applications at different stages of their development. DAST offers flexibility, allowing for the application of diverse approaches and methodologies to achieve a comprehensive security assessment. One facet of DAST involves utilizing an API fuzzing tool. This tool focuses on thoroughly testing all endpoints of an API, aiming to identify vulnerabilities or uncover potential issues that could lead to API failures. By generating a high volume of malformed or unexpected inputs, the fuzzer tests the API's robustness and evaluates its ability to handle unforeseen data correctly. The objective is to expose vulnerabilities or situations where the API may break, enabling developers to implement necessary measures to fortify its security and enhance its ability to withstand potential attacks.

Additionally, DAST can refer to the usage of automatic application scanners, which are integrated into the testing process following the deployment of the application in a dedicated test environment. These scanners simulate attacks and interactions with the application's front-end interface, specifically targeting security flaws such as injection attacks, cross-site scripting, and other vulnerabilities. By actively testing the application's behavior, these scanners provide valuable insights into potential weaknesses that could be exploited by malicious actors. As a result, DAST is categorized as a type of penetration testing - its purpose is to discover vulnerabilities and potential entry points that attackers could exploit in an application. By analyzing the application's behavior and how it responds to different attack scenarios, security experts can identify areas of concern and offer suggestions for fixing them.

DAST tools offer valuable automated testing capabilities, however, they may not uncover all vulnerabilities, particularly those that require manual analysis or a deep understanding of the application's underlying logic. Therefore, it is often best to combine DAST with other testing approaches, such as Static Application Security Testing (SAST) or manual security assessments. This combination ensures a comprehensive evaluation of the application's security, leaving fewer chances for vulnerabilities to go undetected. (4)

Container scanning

Containers are the main structure when it comes to building today’s application architectures. Whether it would be a single container or a multiple container deployment all applications can benefit from container security. Containers have multiple layers that can each introduce vulnerabilities to the system. This is why container scanning is important.

Container scanning is the automated process of analyzing the content of container images to identify potential vulnerabilities or security issues. Containers act like complete runtime environments in their way of storing, executing, and packaging the application and it’s dependencies. Therefore, they can have similar vulnerabilities and furthermore, any library that the container includes adds another potential for vulnerabilities. By introducing container scanning during the development process the developer minimizes the risk of deploying vulnerable containers of their application into production. This is most optimally performed in the CI/CD pipeline as part of the dev ops process automation, before the deployment phase as that way it would add a final check of the containers before they get exposed to the outside world. (7)

What are the measurable benefits of integrating automated security testing into DevSecOps pipelines, such as improved vulnerability detection and reduced time to remediate?

The purpose of CI/CD pipelines is to automate the development process by performing repeatable tasks that would require way too much time to setup and perform manually. The same can be said about automated security testing in the CI/CD pipeline. There are numerous benefits to such automation that are worth the time and effort that it takes to setup. These benefits range from faster and efficient security testing process and improved security to better information gathering, earlier vulnerability awareness, continuous level of security assurance and quicker vulnerability patching.

Faster and efficient security testing

Manual security testing is difficult and time consuming. By integrating automated security testing tools into the CI/CD pipeline, security tests can be executed quickly and efficiently. Static code analysis tools can scan the codebase for potential security vulnerabilities, such as insecure coding practices or known vulnerabilities in third-party libraries. Dynamic application testing tools can simulate attacks and identify security weaknesses in the running application. Container security tools can analyze container images for vulnerabilities and misconfigurations. This largely eliminates the need for manual testing and therefore improves the speed at which security is tested inside the CI/CD pipeline (11)

Early feedback and reduced time to remediate

Automated security testing speeds up the detection and fixing of vulnerabilities. By catching security issues early in development, teams can address them quickly, saving time and preventing complications. This results in shorter time-to-market for software releases. Integrating security testing into the DevSecOps pipeline provides immediate feedback on security risks. Developers receive instant notifications about security issues, allowing them to promptly resolve problems. This early feedback loop enhances teams' understanding of the security impact of their code changes, enabling proactive risk assessment and mitigation. In comparison, manual testing also has it's benefits such as more in-depth testing and smarter penetration tests however it needs to be performed by security experts and requires much more time. (12)

Enhanced compliance adherence

For companies that are especially focused on security such as banks, healthcare companies and government sectors CI/CD integration of security into the development process allows faster and more thorough adherence to security regulations. A DevSecOps pipeline that integrates security testing promotes collaboration between development, security, and operations teams. This collaboration helps foster a security-aware culture inside the organization and ensures that security concerns are addressed as a focused collective rather than an afterthought. By introducing the process of security earlier in the development process these companies can more easily ensure that their software is compliant with all regulations and provides the needed security levels that they require. (13)

What criteria should be considered when selecting automated security testing tools for integration into DevSecOps pipelines?

With the growing importance of cybersecurity and the rising popularity and adoption of the DevSecOps process, there has been an increase in the availability of testing tools on the market. This necessitates the definition of criteria which will be helpful when choosing the correct tool for the job. There are a lot of criteria that can be looked at however the most important for the sake of automation the tools should be easy to use, cheap and ci/cd integratable.

SAST

There is a large number of available static application security testing tools. So in order to evaluate them a set of criteria needs to be taken into account. The first criteria to look at when taking into account static code analysis is the capabilities of the tool – is the tool able to work with the selected language or framework that is being used to create the application in the first place and if so how capable is it in doing so.
Next is the platform agnosticism or how well does it integrate into different platforms whether that is the type of ci/cd pipeline that is being used or the ide.

Finally the price of the tool is also important as it is very easy to ramp up the price with automation tools such as those. Some tools offer free versions and only charge on the integration. Other companies charge on lines of code scanned. With so many things that can be monetized it is important to know in order to determine the worth to cost ratio.

The tool that will be shown in this example is one that covers all these requirements – SonarQube. It is one of if not the most popular SAST tool on the market. It is an open source tool that offers a free version. The only charges can be accrued are related to the hosting of the tool so if that is covered then the free version is good enough to use. For this research I'm going to be scanning an application that is written in TypeScript using NodeJS and the Express framework. Typescript/JavaScript is one of the 15 languages that SonarQube supports, therefore it is covered. SonarQube also has a very easy time integrating into a CI/CD pipeline as it already has some pre-prepared configuration options and menus inside most git repositories such as Azure DevOps and GitHub. (14)

Unit Testing

When creating unit tests for JavaScript and TypeScript there are two major tools that are the most used - Mocha and Jest.

Mocha is an open source, older testing tool and therefore it has a large user base and is well supported. It is more complicated than Jest as it has more options. Therefore, it can be more difficult to learn with the upside of the fact that an experienced user can achieve a lot with it. (15)

Jest is a tool that was developed by the team of Facebook. It is a faster version of Mocha that offers different advantages such as automatic test execution during the development process in order to see if the changes are still congruent with the tests. This is ideal for test-driven-development. In the past it did not support async testing which was a big downside however in the recent years some workarounds have been introduced that help in that regard.

The tool that is going to be shown in this example is Jest as it is an easier tool to learn and offers good automation. Most of the things that disparage it from Mocha have a workaround or have been resolved already. (16)

Software Composition Analysis (SCA) and Container scanning

The key features to looks for in a SCA tool are the ability to scan open sourced software such as modules, frameworks and packages that are a part of the software application that is being scanned. It is important that such a tool can detect known vulnerabilities in open-source software modules and do it thoroughly and fast.

Furthermore an SCA tool might have capabilities to scan a docker container which is a huge advantage.

For this section the tool that is going to be shown in the practice example is Snyk. Snyk is a security company that offers multiple tools for full software coverage. They have a dedicated SCA tool and a dedicated Container scanner tool which should cover all bases. Furthermore, their software is free to use at a 200 tests a month rate which is more than enough for this use case. (19)

DAST

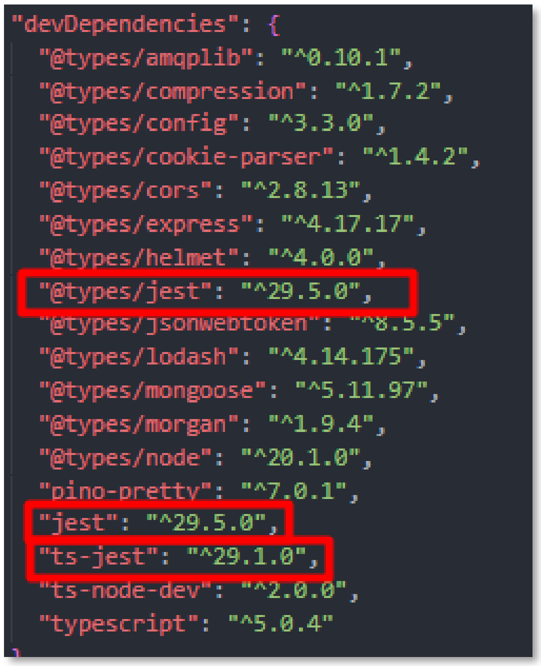
In terms of dynamic application security testing the most important aspects to look at are ease-of-use, scan effectiveness and price.

In this report the tool that is going to be shown is OSWASP ZAP. It was chosen on the basis of it being completely free to use and integrate into a ci/cd pipeline. (17) Other tools such as Burp Suite are more developed and easier to setup however they offer the same scanning potential and require payment for the enterprise version of the software which means that they can’t be integrated into a CI/CD pipeline for free. (18)

What are the existing automated security testing tools and frameworks available for integration into DevSecOps pipelines?

Unit tests with Jest

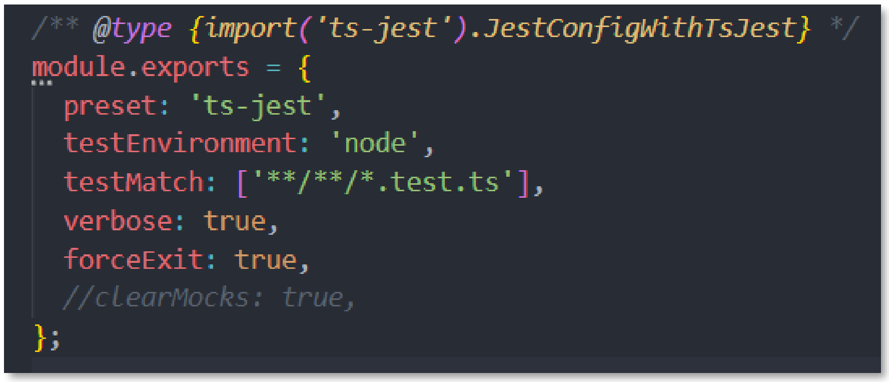
Setting up unit tests in a Typescript application can be done very easily with Jest. First an install is required as Jest is a third party software.



```json
"devDependencies": {
  "@types/amqplib": "^0.10.1",
  "@types/compression": "^1.7.2",
  "@types/config": "^3.3.0",
  "@types/cookie-parser": "^1.4.2",
  "@types/cors": "^2.8.13",
  "@types/express": "^4.17.17",
  "@types/helmet": "^4.0.0",
  "@types/jest": "^29.5.0",
  "@types/jsonwebtoken": "^8.5.5",
  "@types/lodash": "^4.14.175",
  "@types/mongoose": "^5.11.97",
  "@types/morgan": "^1.9.4",
  "@types/node": "^20.1.0",
  "pino-pretty": "^7.0.1",
  "jest": "^29.5.0",
  "ts-jest": "^29.1.0",
  "ts-node-dev": "^2.0.0",
  "typescript": "^5.0.4"
```

Figure 1 - Package.json

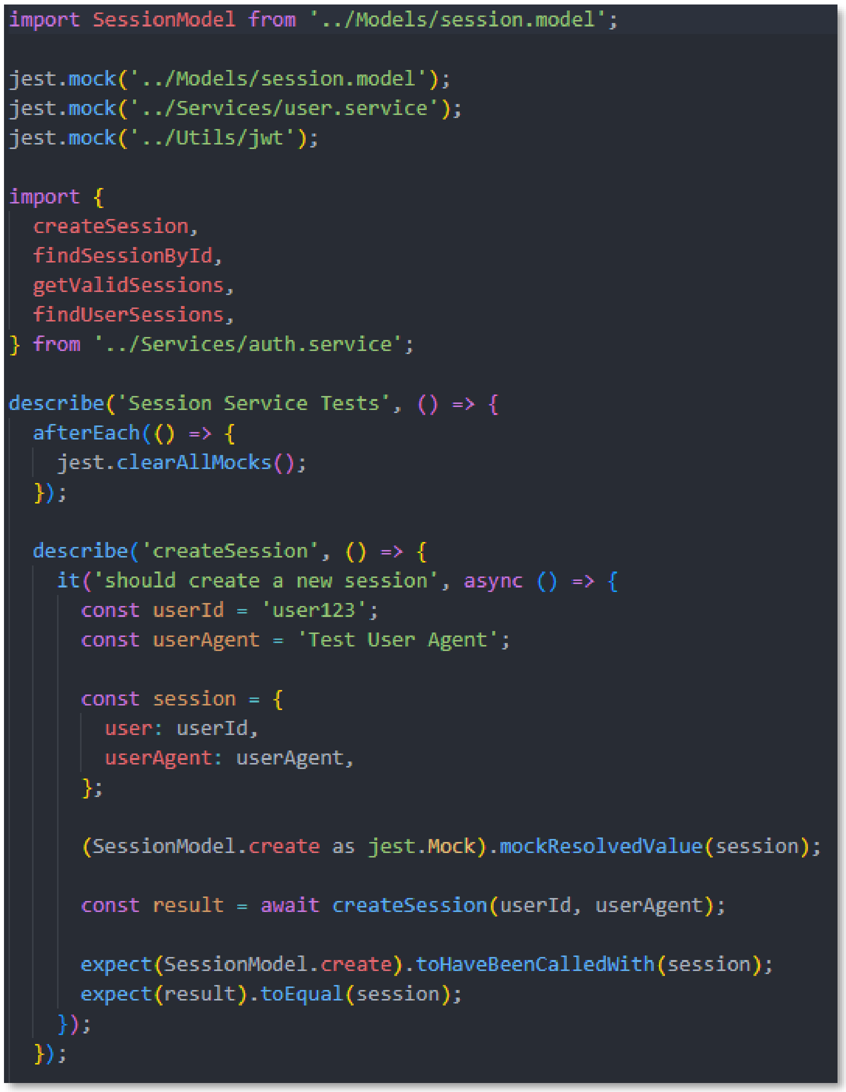
The package.json of the project containing information about the jest setup. Because the application uses Typescript we need to install the types declaration as well with **@types/jest.**



```javascript
/** @type {import('ts-jest').JestConfigWithTsJest} */
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  testMatch: ['**/**/*.test.ts'],
  verbose: true,
  forceExit: true,
  //clearMocks: true,
};
```

Figure 2 - jest.config.js

The next step is to configure the **jest.config.js** configuration file so that it knows that it should use the **ts-jest** preset in order to run typescript tests as well as define the tests folder from where they can be started automatically using the "jest" command.



```
import SessionModel from '../Models/session.model';

jest.mock('../Models/session.model');
jest.mock('../Services/user.service');
jest.mock('../Utils/jwt');

import {
  createSession,
  findSessionById,
  getValidSessions,
  findUserSessions,
} from '../Services/auth.service';

describe('Session Service Tests', () => {
  afterEach(() => {
    jest.clearAllMocks();
  });

  describe('createSession', () => {
    it('should create a new session', async () => {
      const userId = 'user123';
      const userAgent = 'Test User Agent';

      const session = {
        user: userId,
        userAgent: userAgent,
      };

      (SessionModel.create as jest.Mock).mockResolvedValue(session);

      const result = await createSession(userId, userAgent);

      expect(SessionModel.create).toHaveBeenCalledWith(session);
      expect(result).toEqual(session);
    });
  });
```

Figure 3 - Unit Test example

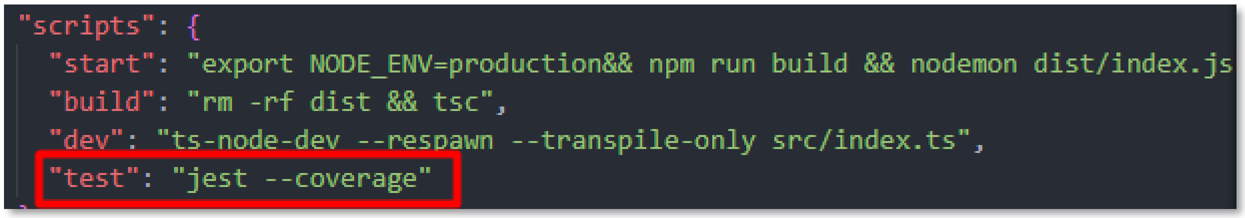
An example of one of the unit tests in the application. The test begins with an import of the service that is being tested after which it is mocked along with all of its dependencies.

Then all the functions that are going to be tested are imported as well.

A description of the test file is defined and after that so is a description of the first unit test in this document.

The “createSession” unit test begins with a definition of a user and a userAgent. These are random test values that are needed for the creation of a session.

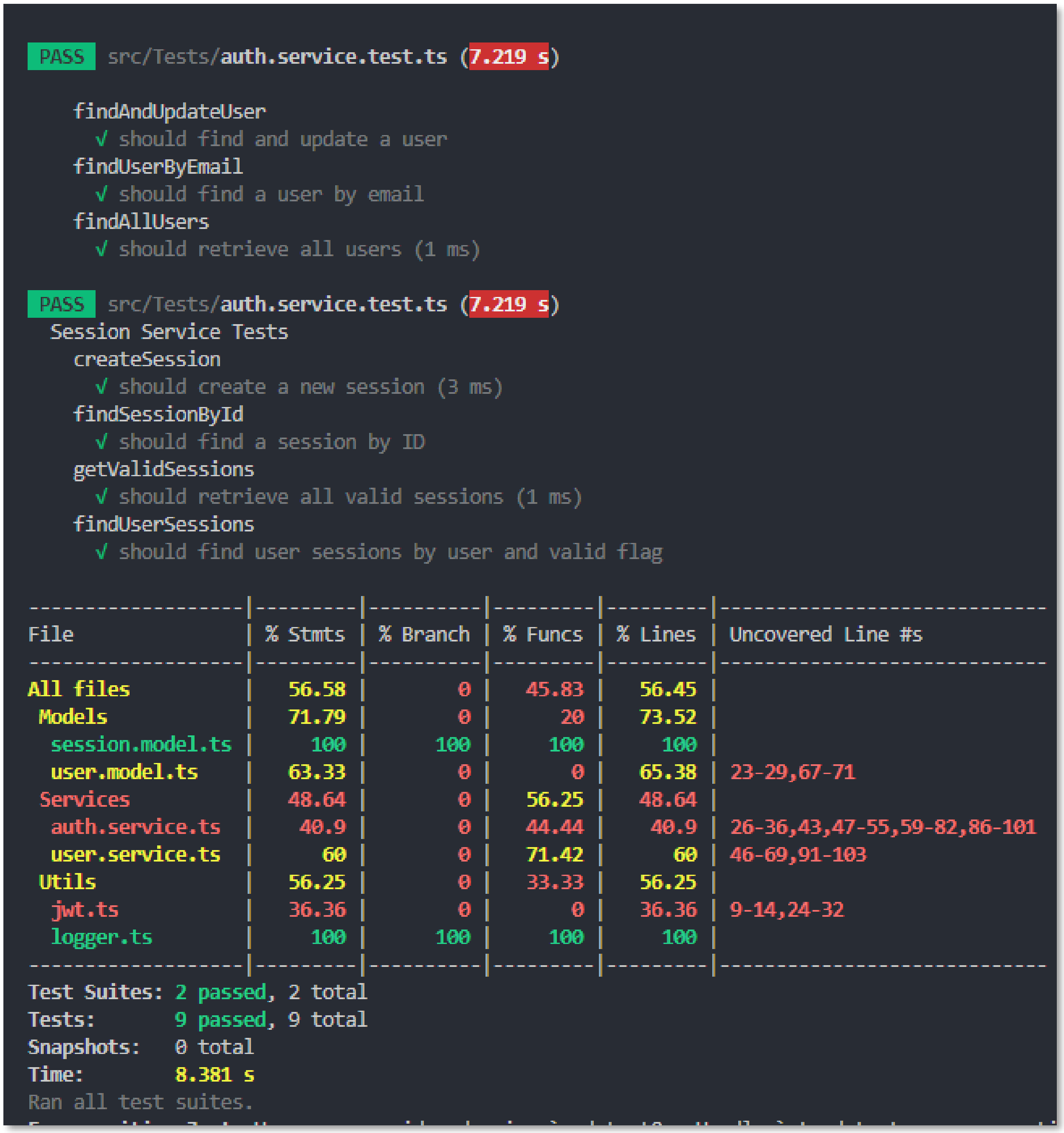
The function is called with the hardcoded values. The test checks whether the function was called correctly and whether the result is equal to the hardcoded value.



```
"scripts": {
  "start": "export NODE_ENV=production&& npm run build && nodemon dist/index.js
  "build": "rm -rf dist && tsc",
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
  "test": "jest --coverage"
```

Figure 4 - Test trigger script with --coverage option

The test trigger script can be called with the command “**npm run test**”. The –coverage option describes an automatic generation of a “coverage” directory that is automatically created inside the base directory for the project. The purpose of this directory is to communicate to SonarQube of the percentage of unit test coverage for this project.

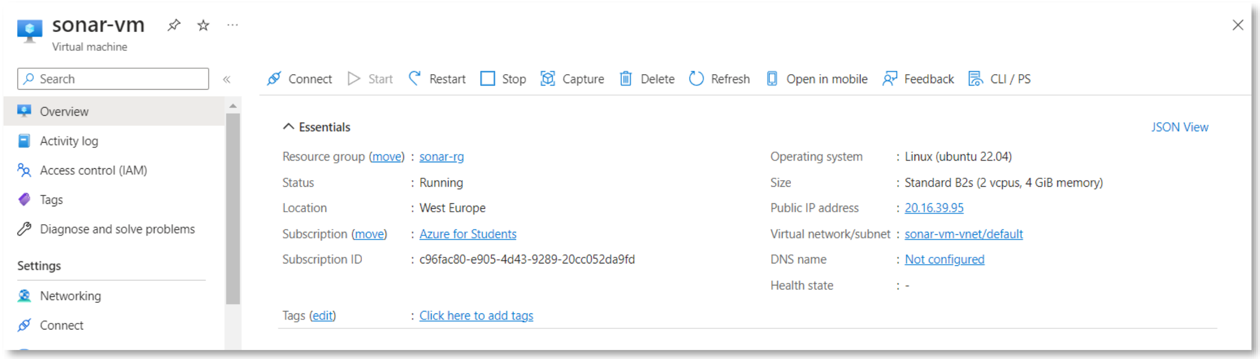


```
 PASS  src/Tests/auth.service.test.ts (7.219 s)

    findAndUpdateUser
      √ should find and update a user
    findUserByEmail
      √ should find a user by email
    findAllUsers
      √ should retrieve all users (1 ms)

 PASS  src/Tests/auth.service.test.ts (7.219 s)
  Session Service Tests
    createSession
      √ should create a new session (3 ms)
    findSessionById
      √ should find a session by ID
    getValidSessions
      √ should retrieve all valid sessions (1 ms)
    findUserSessions
      √ should find user sessions by user and valid flag

-------------------|---------|----------|---------|---------|-----------------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-----------------------------
All files          |   56.58 |        0 |   45.83 |   56.45 |
 Models            |   71.79 |        0 |      20 |   73.52 |
  session.model.ts |     100 |      100 |     100 |     100 |
  user.model.ts    |   63.33 |        0 |       0 |   65.38 | 23-29,67-71
 Services          |   48.64 |        0 |   56.25 |   48.64 |
  auth.service.ts  |    40.9 |        0 |   44.44 |    40.9 | 26-36,43,47-55,59-82,86-101
  user.service.ts  |      60 |        0 |   71.42 |      60 | 46-69,91-103
 Utils             |   56.25 |        0 |   33.33 |   56.25 |
  jwt.ts           |   36.36 |        0 |       0 |   36.36 | 9-14,24-32
  logger.ts        |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-----------------------------
Test Suites: 2 passed, 2 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        8.381 s
Ran all test suites.
```

Figure 5 - Unit test report and coverage report

After the test suites are ran a report is generate with the names of the tests and whether they passed or not, along with the coverage report.

SonarQube (SAST)

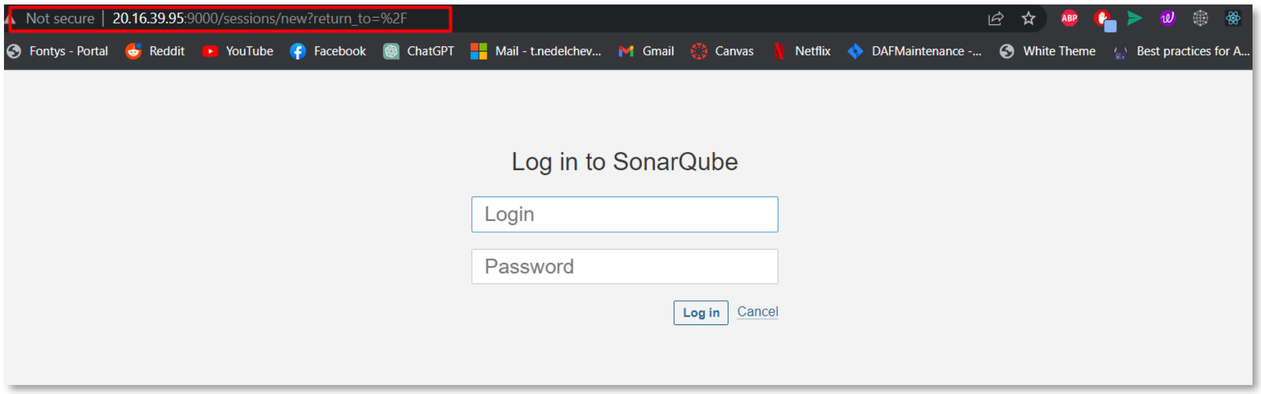

sonar-vm
Virtual machine

Essentials

Resource group (move)	: sonar-rg
Status	: Running
Location	: West Europe
Subscription (move)	: Azure for Students
Subscription ID	: c96fac80-e905-4d43-9289-20cc052da9fd
Operating system	: Linux (ubuntu 22.04)
Size	: Standard B2s (2 vcpus, 4 GiB memory)
Public IP address	: 20.16.39.95
Virtual network/subnet	: sonar-vm-vnet/default
DNS name	: Not configured
Health state	: -
Tags (edit)	: Click here to add tags

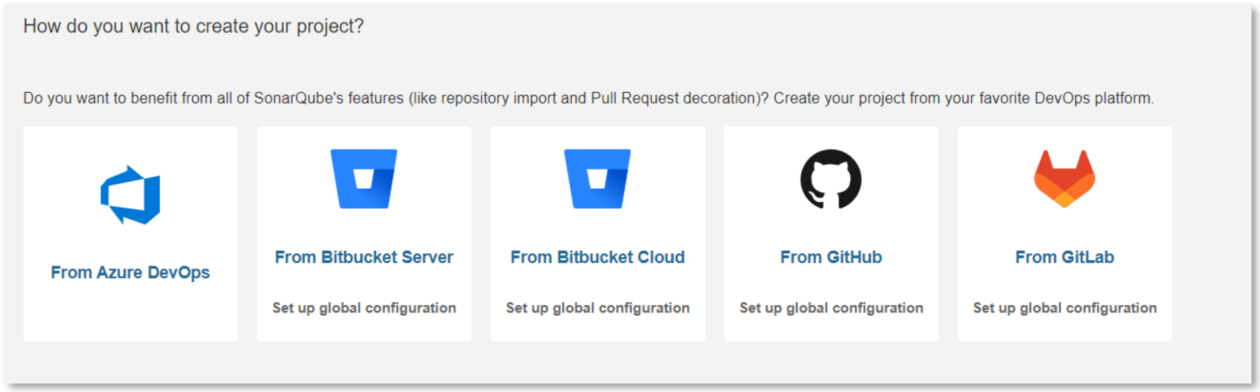
Figure 6 - SonarQube deployed on a VM on Azure Cloud

The SonarQube setup has been successfully deployed and configured on an Azure virtual machine (VM). Due to limitations in the subscription, the automatic SonarQube setup from the Azure Marketplace could not be utilized. This is why, a manual deployment had to be carried out on an Ubuntu VM. This approach takes a lot longer to setup but is just as viable.


Not secure | 20.16.39.95:9000/sessions/new?return_to=%2F

Log in to SonarQube

Figure 7 - SonarQube is accessible from it's IP on port 9000

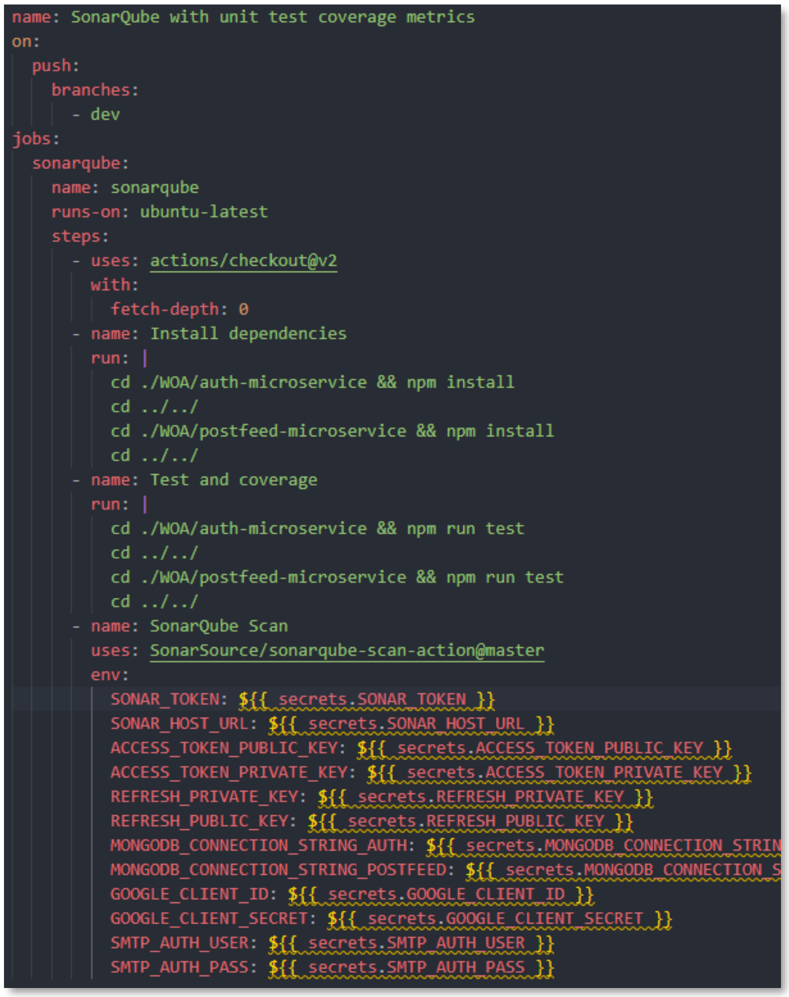

How do you want to create your project?

Do you want to benefit from all of SonarQube's features (like repository import and Pull Request decoration)? Create your project from your favorite DevOps platform.

From Azure DevOps
From Bitbucket Server
Set up global configuration
From Bitbucket Cloud
Set up global configuration
From GitHub
Set up global configuration
From GitLab
Set up global configuration

Figure 8 - SonarQube Dashboard after login.

I will be using GitHub actions to connect and configure automatic SonarQube scans.



```yaml
name: SonarQube with unit test coverage metrics
on:
  push:
    branches:
      - dev
jobs:
  sonarqube:
    name: sonarqube
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      - name: Install dependencies
        run: |
          cd ./WOA/auth-microservice && npm install
          cd ../../
          cd ./WOA/postfeed-microservice && npm install
          cd ../../
      - name: Test and coverage
        run: |
          cd ./WOA/auth-microservice && npm run test
          cd ../../
          cd ./WOA/postfeed-microservice && npm run test
          cd ../../
      - name: SonarQube Scan
        uses: SonarSource/sonarqube-scan-action@master
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
          SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
          ACCESS_TOKEN_PUBLIC_KEY: ${{ secrets.ACCESS_TOKEN_PUBLIC_KEY }}
          ACCESS_TOKEN_PRIVATE_KEY: ${{ secrets.ACCESS_TOKEN_PRIVATE_KEY }}
          REFRESH_PRIVATE_KEY: ${{ secrets.REFRESH_PRIVATE_KEY }}
          REFRESH_PUBLIC_KEY: ${{ secrets.REFRESH_PUBLIC_KEY }}
          MONGODB_CONNECTION_STRING_AUTH: ${{ secrets.MONGODB_CONNECTION_STRIN
          MONGODB_CONNECTION_STRING_POSTFEED: ${{ secrets.MONGODB_CONNECTION_S
          GOOGLE_CLIENT_ID: ${{ secrets.GOOGLE_CLIENT_ID }}
          GOOGLE_CLIENT_SECRET: ${{ secrets.GOOGLE_CLIENT_SECRET }}
          SMTP_AUTH_USER: ${{ secrets.SMTP_AUTH_USER }}
          SMTP_AUTH_PASS: ${{ secrets.SMTP_AUTH_PASS }}
```

Figure 9 - SonarQube with unit tests coverage metrics

The .yaml file that is going to trigger unit testing and SonarQube scan.

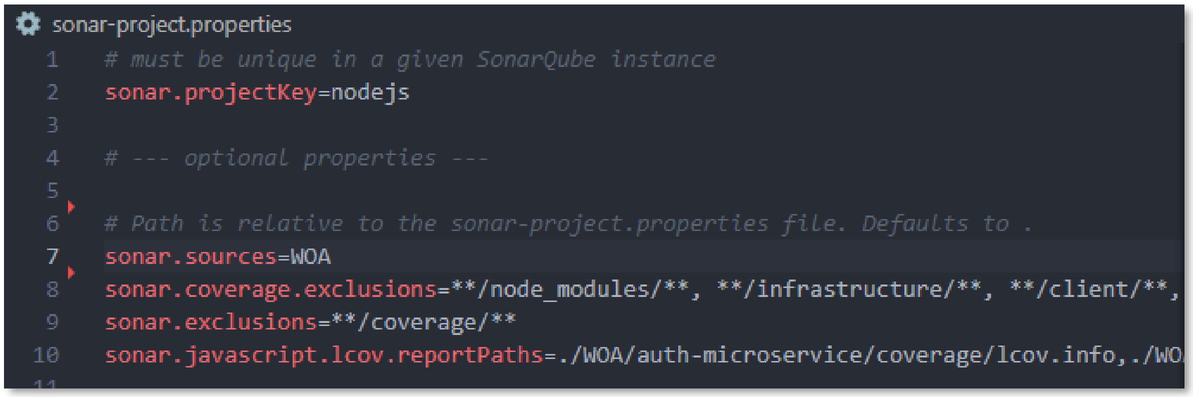
The ENV variables come from GitHub repository secrets where they have been setup beforehand.

The job runs on a latest version of Ubuntu. The first step is to clone the repository tot his environment.

After that it's time to install the dependencies with npm install inside the microservices' directories where the unit tests reside so that the tests can be ran.

Consequently, the tests are ran from all directories where tests reside.

Finally SonarQube is ran. It uses a preconfigured token and a URL in order to connect to the deployed version of SonarQube.

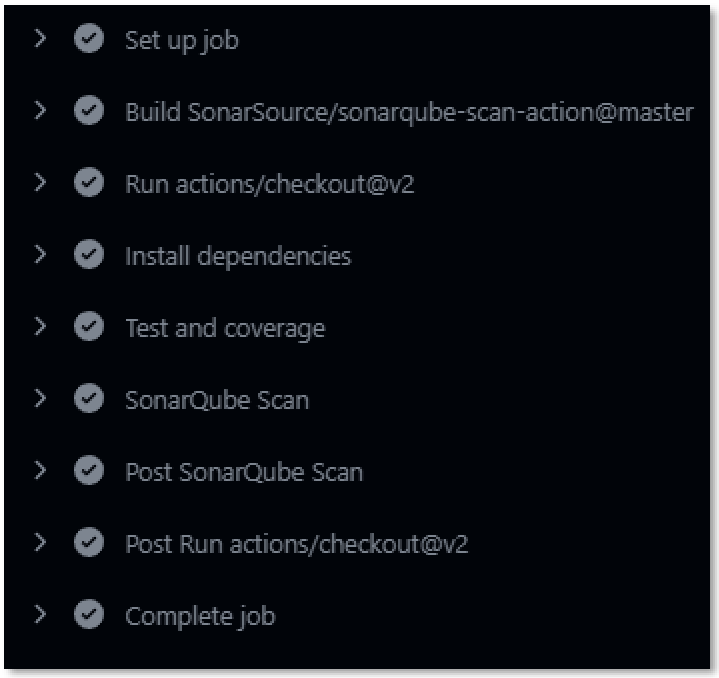


sonar-project.properties

```properties
# must be unique in a given SonarQube instance
sonar.projectKey=nodejs

# --- optional properties ---

# Path is relative to the sonar-project.properties file. Defaults to .
sonar.sources=WOA
sonar.coverage.exclusions=**/node_modules/**, **/infrastructure/**, **/client/**,
sonar.exclusions=**/coverage/**
sonar.javascript.lcov.reportPaths=./WOA/auth-microservice/coverage/lcov.info,./WO
```

Figure 10 - sonar-project.properties Sonar Scan configuration file

SonarQube requires a configuration file in the project base folder that contains explanations that help SonarQube to navigate the project, include, and exclude different files and directories from being scanned.

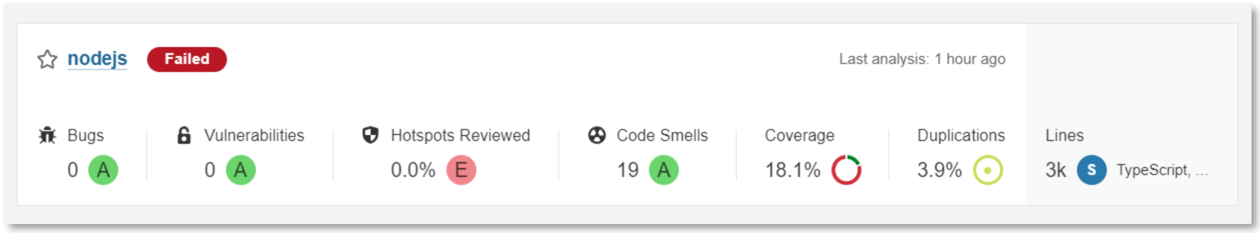


```
> Set up job
> Build SonarSource/sonarqube-scan-action@master
> Run actions/checkout@v2
> Install dependencies
> Test and coverage
> SonarQube Scan
> Post SonarQube Scan
> Post Run actions/checkout@v2
> Complete job
```

The tasks all completed in GitHub actions.

Figure 11 - All test actions completed

Finally after the static code analysis is finished



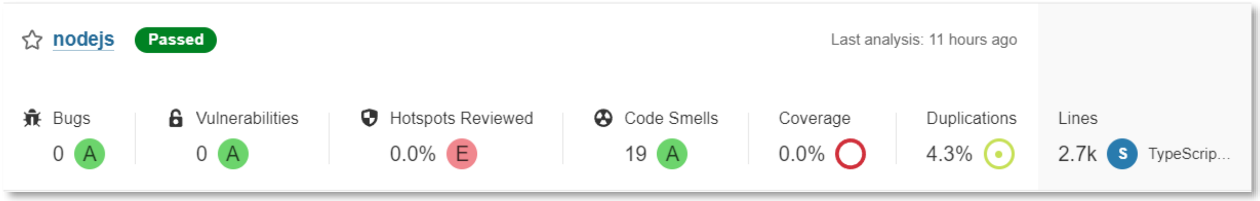
nodejs Failed — Last analysis: 1 hour ago

Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines
0 A	0 A	0.0% E	19 A	18.1%	3.9%	3k S TypeScript, ...

Figure 12 - Completed SonarQube scan with 0 bugs and 19 smells inside 3000 lines code (The fail is because of only 18% unit test coverage)

The project appears in SonarQube on the cloud's projects menu. The failed status is because the api failed an arbitrary metric for unit test coverage. In this case it is only 18%

All other metrics are alright as the scan passes with no problem without the unit test coverage metric.



nodejs Passed — Last analysis: 11 hours ago

Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines
0 A	0 A	0.0% E	19 A	0.0%	4.3%	2.7k S TypeScrip...

Figure 13 - Sonar Scan without unit test coverage metric - passes

Software Composition Analysis (SCA) and Container Security

Snyk offers bot SCA and docker container scanning. For the sake of SCA it needs to be connected to a git repository. After logging in with a GitHub account and being granted access Snyk needs a few minutes to load all repositories from the account. After that it is connected and begins a constant vigilance over the repository.

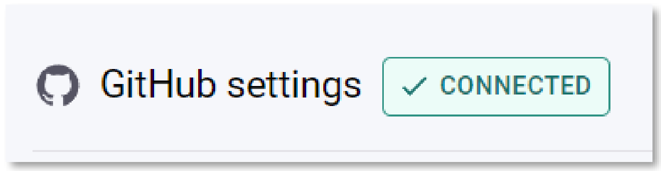
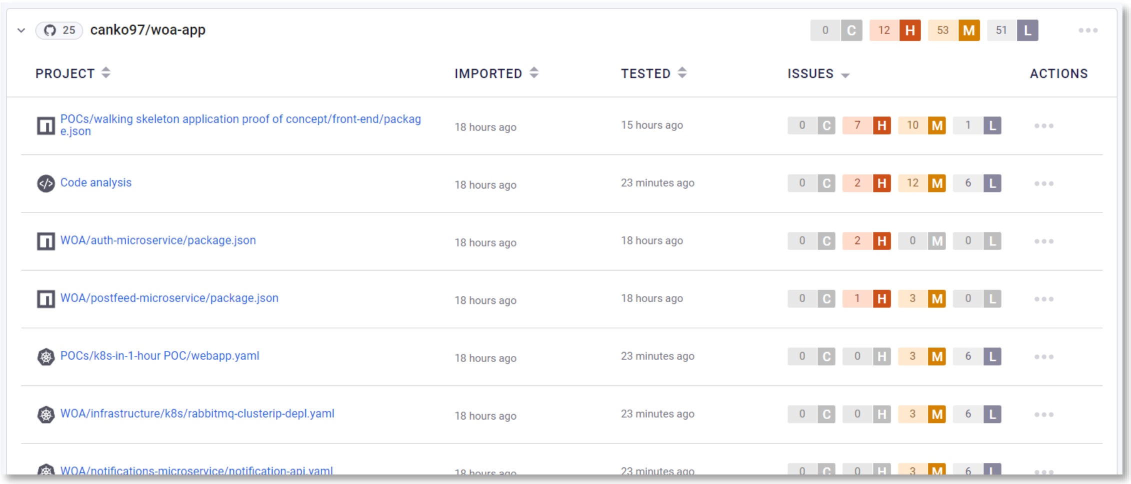


Figure 14 - Snyk integrated with GitHub

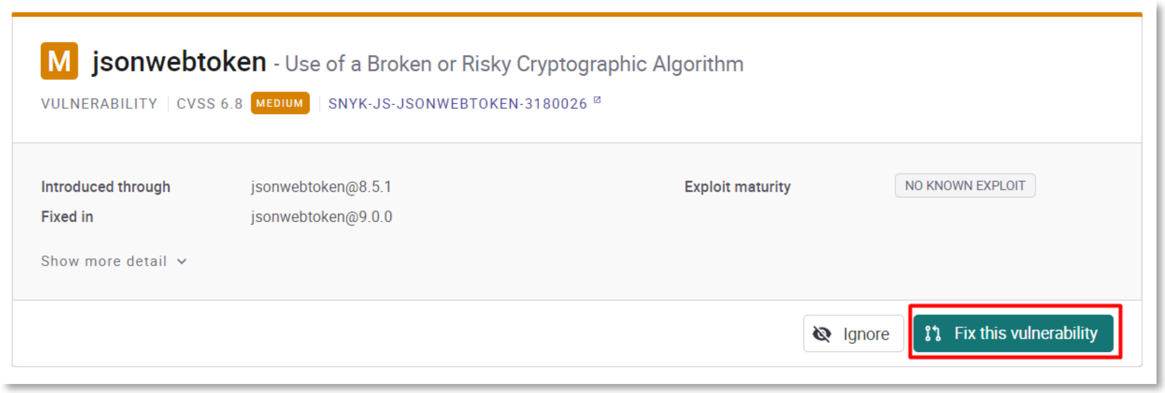


25 canko97/woa-app — 0 C 12 H 53 M 51 L

PROJECT	IMPORTED	TESTED	ISSUES	ACTIONS
POCs/walking skeleton application proof of concept/front-end/package.json	18 hours ago	15 hours ago	0 C 7 H 10 M 1 L	•••
Code analysis	18 hours ago	23 minutes ago	0 C 2 H 12 M 6 L	•••
WOA/auth-microservice/package.json	18 hours ago	18 hours ago	0 C 2 H 0 M 0 L	•••
WOA/postfeed-microservice/package.json	18 hours ago	18 hours ago	0 C 1 H 3 M 0 L	•••
POCs/k8s-in-1-hour POC/webapp.yaml	18 hours ago	23 minutes ago	0 C 0 H 3 M 6 L	•••
WOA/infrastructure/k8s/rabbitmq-clusterip-depl.yaml	18 hours ago	23 minutes ago	0 C 0 H 3 M 6 L	•••
WOA/notifications-microservice/notification-api.yaml	18 hours ago	23 minutes ago	0 C 0 H 3 M 6 L	•••

Figure 15 - After it is connected it needs a few seconds to scan the repository and find potential vulnerabilities.

Snyk will keep a constant status of the repository's problems and vulnerabilities. It will scan all directories and look through code, Dockerfile configurations and Kubernetes manifests in order to determine if there are any problems. Snyk will also sometimes offer (if the right permissions are set) to automatically fix a vulnerability that is very simple to fix or has a universal solution.



M **jsonwebtoken** - Use of a Broken or Risky Cryptographic Algorithm

VULNERABILITY | CVSS 6.8 MEDIUM | SNYK-JS-JSONWEBTOKEN-3180026

Introduced through	jsonwebtoken@8.5.1	Exploit maturity	NO KNOWN EXPLOIT
Fixed in	jsonwebtoken@9.0.0		

Figure 16 - Snyk offers to fix an outdated dependency

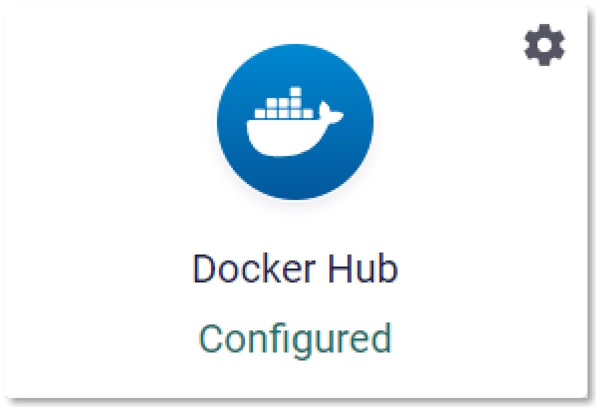
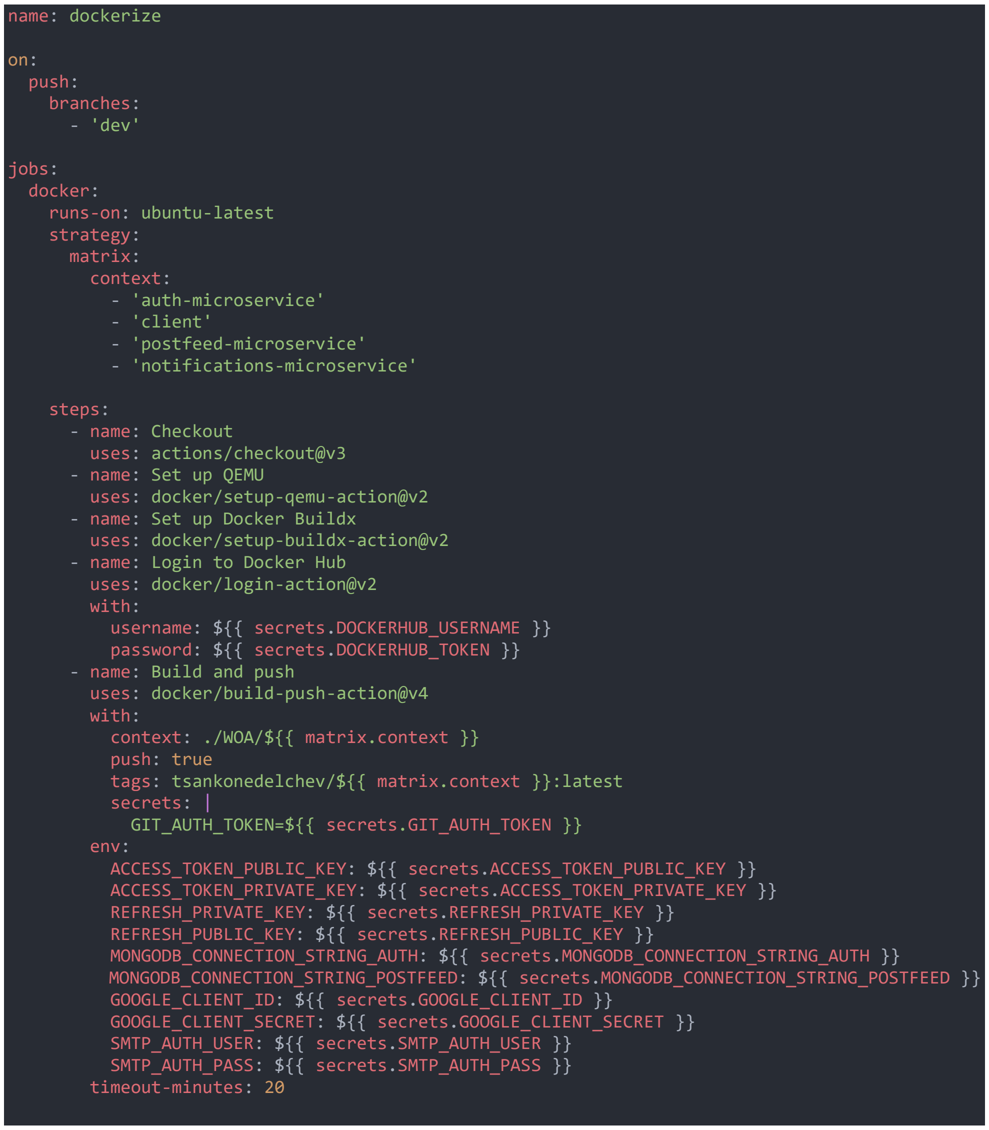


Figure 17 - Snyk/Docker Hub integration completed via a pre-generated Docker Hub personal access token

After connecting Docker Hub to Snyk using a git hub generated token along with the Docker Hub repository username Snyk will offer to download all the images that are in the repository.

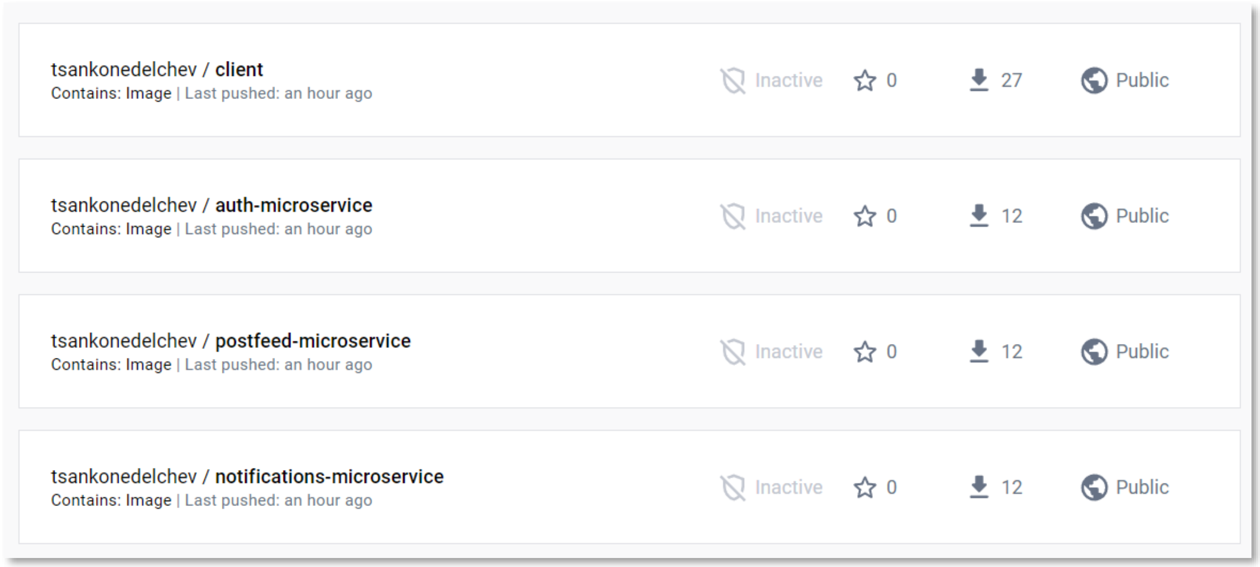


```yaml
name: dockerize

on:
  push:
    branches:
      - 'dev'

jobs:
  docker:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        context:
          - 'auth-microservice'
          - 'client'
          - 'postfeed-microservice'
          - 'notifications-microservice'

    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: ./WOA/${{ matrix.context }}
          push: true
          tags: tsankonedelchev/${{ matrix.context }}:latest
          secrets: |
            GIT_AUTH_TOKEN=${{ secrets.GIT_AUTH_TOKEN }}
        env:
          ACCESS_TOKEN_PUBLIC_KEY: ${{ secrets.ACCESS_TOKEN_PUBLIC_KEY }}
          ACCESS_TOKEN_PRIVATE_KEY: ${{ secrets.ACCESS_TOKEN_PRIVATE_KEY }}
          REFRESH_PRIVATE_KEY: ${{ secrets.REFRESH_PRIVATE_KEY }}
          REFRESH_PUBLIC_KEY: ${{ secrets.REFRESH_PUBLIC_KEY }}
          MONGODB_CONNECTION_STRING_AUTH: ${{ secrets.MONGODB_CONNECTION_STRING_AUTH }}
          MONGODB_CONNECTION_STRING_POSTFEED: ${{ secrets.MONGODB_CONNECTION_STRING_POSTFEED }}
          GOOGLE_CLIENT_ID: ${{ secrets.GOOGLE_CLIENT_ID }}
          GOOGLE_CLIENT_SECRET: ${{ secrets.GOOGLE_CLIENT_SECRET }}
          SMTP_AUTH_USER: ${{ secrets.SMTP_AUTH_USER }}
          SMTP_AUTH_PASS: ${{ secrets.SMTP_AUTH_PASS }}
        timeout-minutes: 20
```

However, before that, there need to be images in the repository. This can be done by manually pushing the images to the repository, however in that case it's not automated.

This is where GitHub actions come in handy again.
This .yaml file serves to build and push the Docker images to Docker Hub

After the Docker images have been uploaded on Docker Hub they can be downloaded on Snyk and scanned.

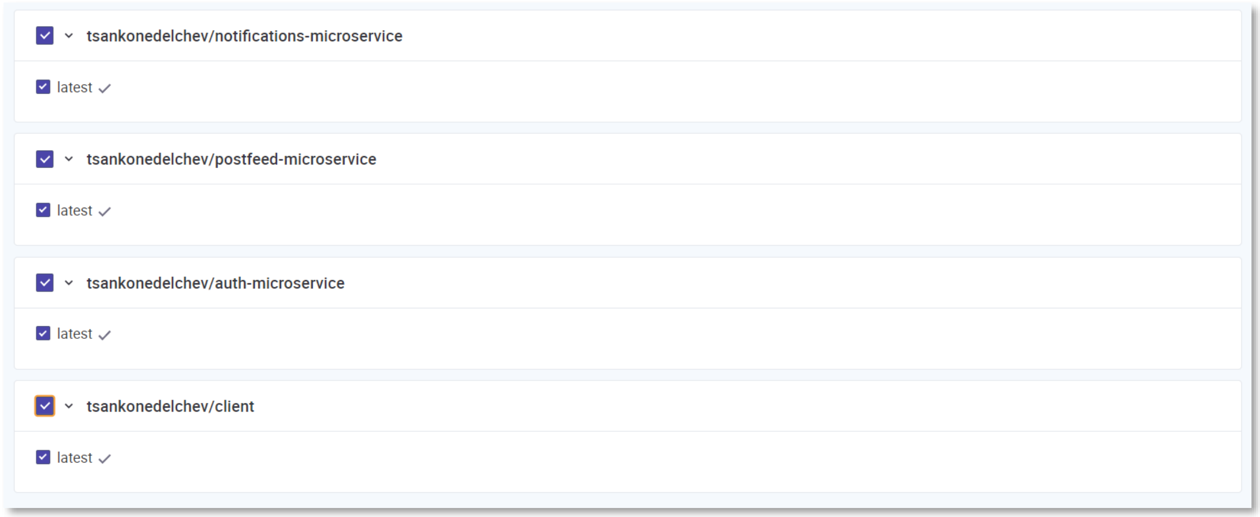

tsankonedelchev / **client**
Contains: Image | Last pushed: an hour ago
Inactive 0 27 Public

tsankonedelchev / **auth-microservice**
Contains: Image | Last pushed: an hour ago
Inactive 0 12 Public

tsankonedelchev / **postfeed-microservice**
Contains: Image | Last pushed: an hour ago
Inactive 0 12 Public

tsankonedelchev / **notifications-microservice**
Contains: Image | Last pushed: an hour ago
Inactive 0 12 Public

Figure 18 - Docker Images uploaded to Docker Hub


☑ tsankonedelchev/notifications-microservice
☑ latest ✓

☑ tsankonedelchev/postfeed-microservice
☑ latest ✓

☑ tsankonedelchev/auth-microservice
☑ latest ✓

☑ tsankonedelchev/client
☑ latest ✓

Figure 19- Docker images appear ready to be downloaded on Snyk to be scanned for vulnerabilities

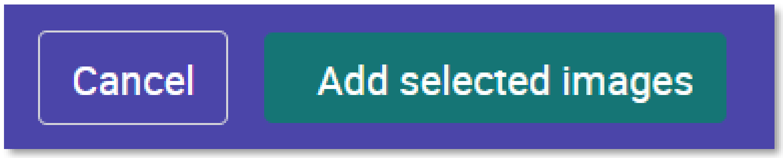
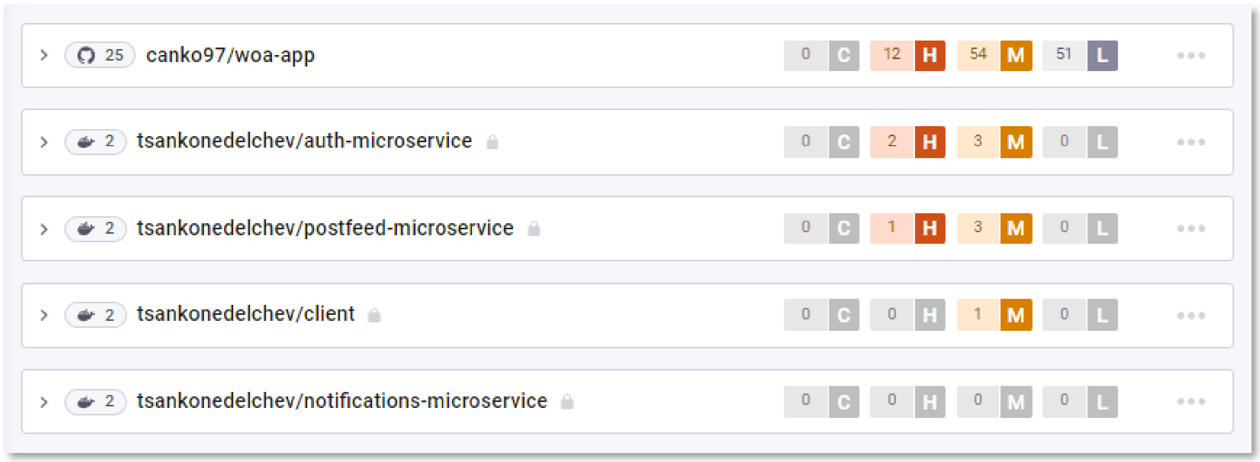


Figure 20 - Snyk "add selected images" button adds the images to Snyk's environment.



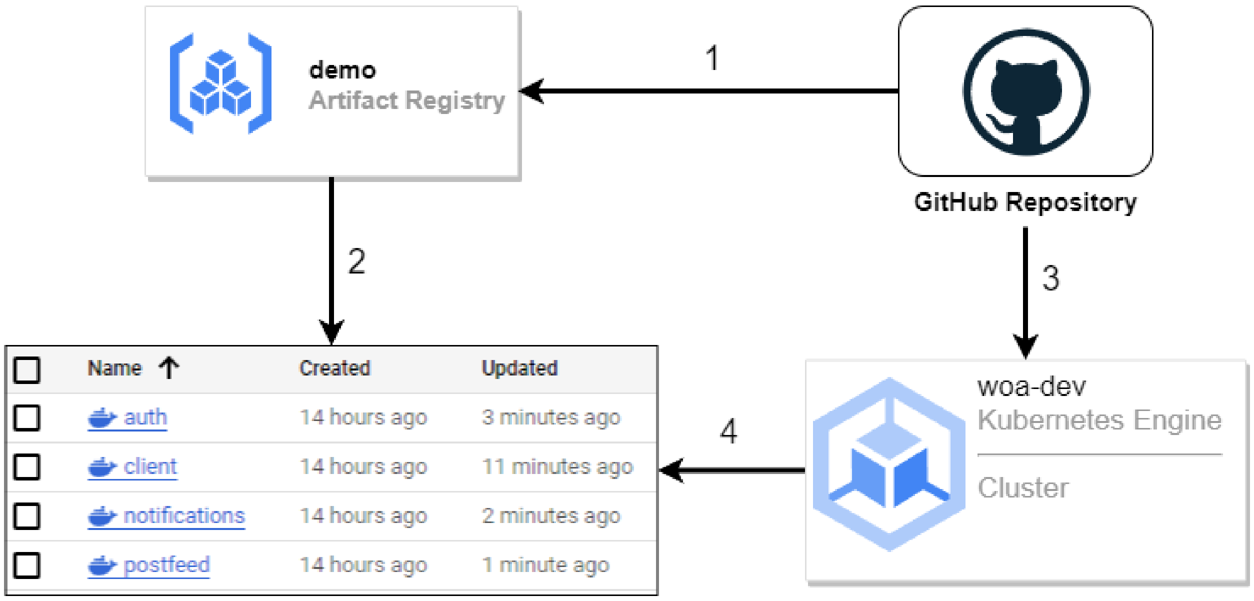
		C	H	M	L
25	canko97/woa-app	0	12	54	51
2	tsankonedelchev/auth-microservice	0	2	3	0
2	tsankonedelchev/postfeed-microservice	0	1	3	0
2	tsankonedelchev/client	0	0	1	0
2	tsankonedelchev/notifications-microservice	0	0	0	0

Figure 21 - Finally Snyk contains several projects - the repository from GitHub and the Images from Docker Hub and can scan them every time they get updated.

Dynamic Application Security Testing (DAST)

So far the pipeline has mainly focused on the **dev** branch as it executed tests and triggered SonarQube scans as well as built images in order for them to be tested by Snyk on each push. DAST however is best performed on a deployed application. So the next steps will be performed on the **main** branch which is the production branch of the repository.

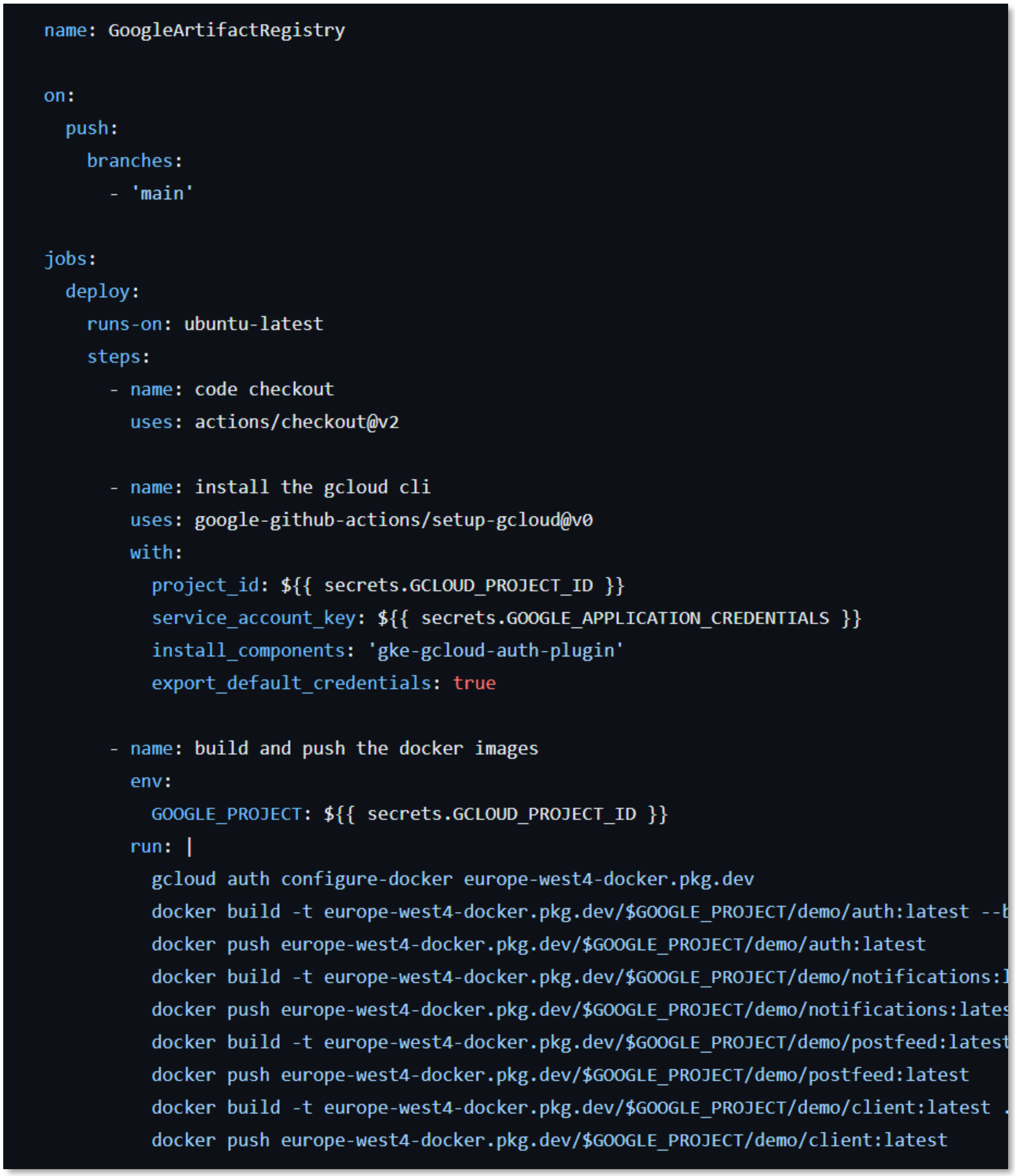
First it is best to deploy the application. In order to do that the following workflow will be setup:



Name	Created	Updated
auth	14 hours ago	3 minutes ago
client	14 hours ago	11 minutes ago
notifications	14 hours ago	2 minutes ago
postfeed	14 hours ago	1 minute ago

Figure 22 – Deployment GitHub workflow action

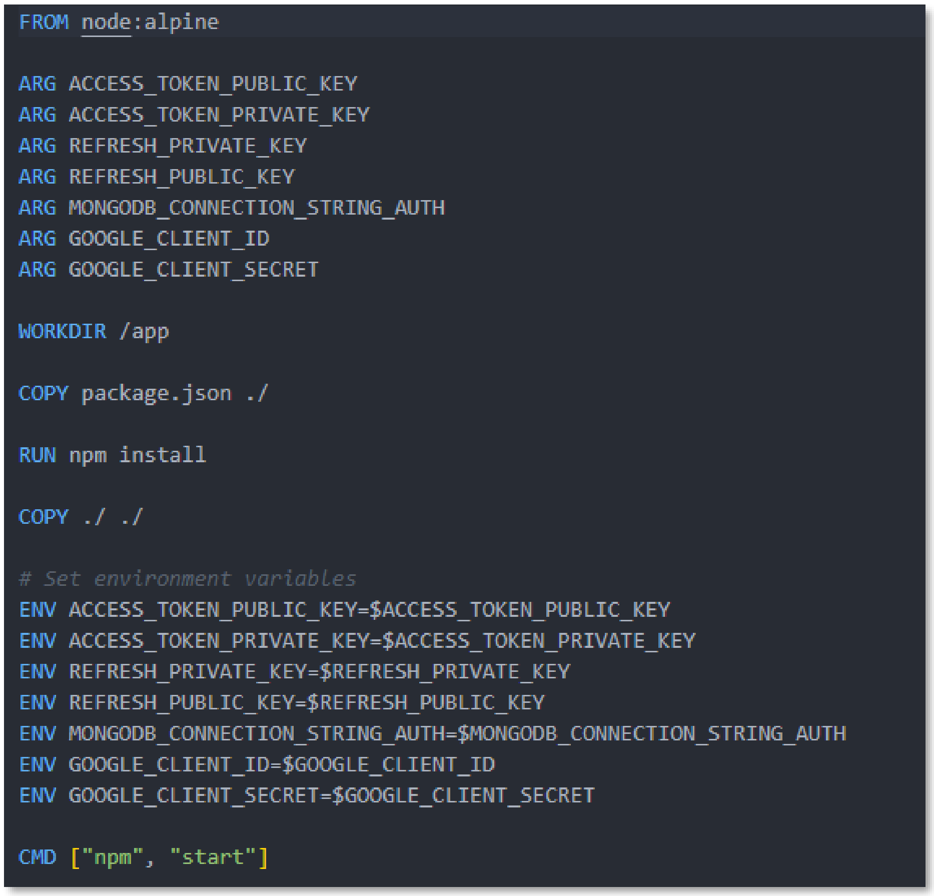
1. GitHub builds the docker containers and pushes them to Google Artifact registry. There they stay stored until GitHub tells Google Kubernetes Engine to download them and use them to build pods.



```yaml
name: GoogleArtifactRegistry

on:
  push:
    branches:
      - 'main'

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: code checkout
        uses: actions/checkout@v2

      - name: install the gcloud cli
        uses: google-github-actions/setup-gcloud@v0
        with:
          project_id: ${{ secrets.GCLOUD_PROJECT_ID }}
          service_account_key: ${{ secrets.GOOGLE_APPLICATION_CREDENTIALS }}
          install_components: 'gke-gcloud-auth-plugin'
          export_default_credentials: true

      - name: build and push the docker images
        env:
          GOOGLE_PROJECT: ${{ secrets.GCLOUD_PROJECT_ID }}
        run: |
          gcloud auth configure-docker europe-west4-docker.pkg.dev
          docker build -t europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/auth:latest --b
          docker push europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/auth:latest
          docker build -t europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/notifications:l
          docker push europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/notifications:lates
          docker build -t europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/postfeed:latest
          docker push europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/postfeed:latest
          docker build -t europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/client:latest .
          docker push europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/client:latest
```

Figure 23 - First part of the deployment process -pushing the docker images to Artifact Registry

In the first part of the deployment process the docker images are pushed to Artifact Registry. The connection is made via a google service account with artifact registry access permissions.



```
FROM node:alpine

ARG ACCESS_TOKEN_PUBLIC_KEY
ARG ACCESS_TOKEN_PRIVATE_KEY
ARG REFRESH_PRIVATE_KEY
ARG REFRESH_PUBLIC_KEY
ARG MONGODB_CONNECTION_STRING_AUTH
ARG GOOGLE_CLIENT_ID
ARG GOOGLE_CLIENT_SECRET

WORKDIR /app

COPY package.json ./

RUN npm install

COPY ./ ./

# Set environment variables
ENV ACCESS_TOKEN_PUBLIC_KEY=$ACCESS_TOKEN_PUBLIC_KEY
ENV ACCESS_TOKEN_PRIVATE_KEY=$ACCESS_TOKEN_PRIVATE_KEY
ENV REFRESH_PRIVATE_KEY=$REFRESH_PRIVATE_KEY
ENV REFRESH_PUBLIC_KEY=$REFRESH_PUBLIC_KEY
ENV MONGODB_CONNECTION_STRING_AUTH=$MONGODB_CONNECTION_STRING_AUTH
ENV GOOGLE_CLIENT_ID=$GOOGLE_CLIENT_ID
ENV GOOGLE_CLIENT_SECRET=$GOOGLE_CLIENT_SECRET

CMD ["npm", "start"]
```

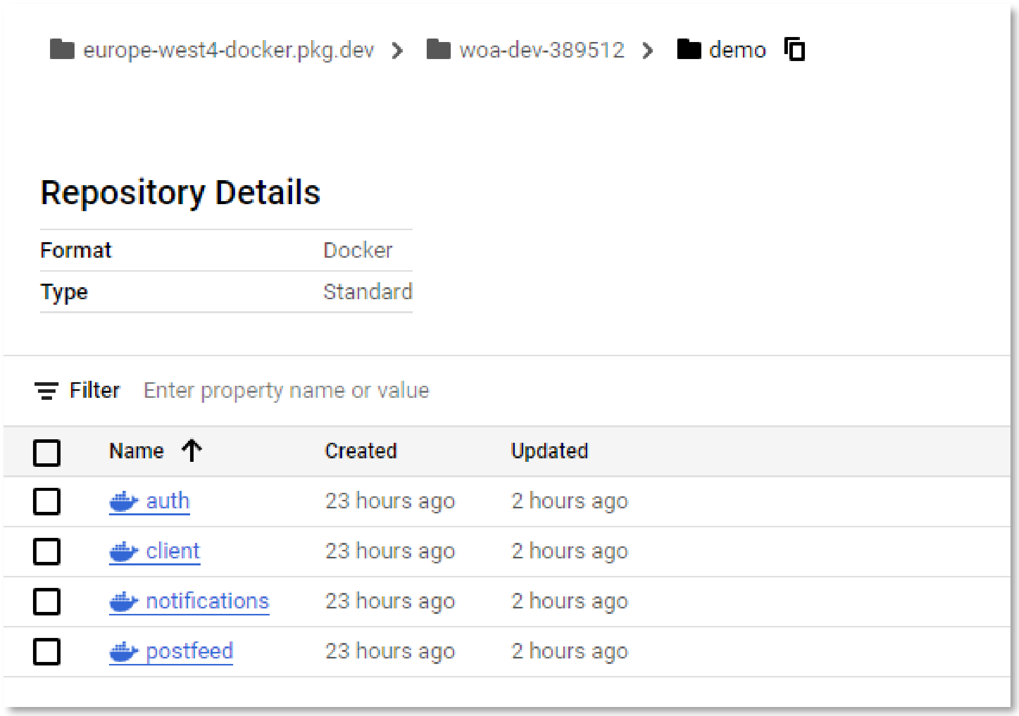
Figure 24 - auth-microservice's Dockerfile with secret generated dynamic EVN variables

The images are build by passing the GitHub repo secrets as ARG tags and then applying them as ENV variables inside the Dockerfile. Performing the translation this way ensures that the secrets are completely safe from being found out.

The following command takes the secrets from GitHub's repo secrets and translates it to the ARG variable in the Dockerfile:

```
docker build -t europe-west4-docker.pkg.dev/$GOOGLE_PROJECT/demo/auth:latest --build-arg ACCESS_TOKEN_PUBLIC_KEY="${{ secrets.ACCESS_TOKEN_PUBLIC_KEY }}" --build-arg ACCESS_TOKEN_PRIVATE_KEY="${{ secrets.ACCESS_TOKEN_PRIVATE_KEY }}" --build-arg REFRESH_PRIVATE_KEY="${{ secrets.REFRESH_PRIVATE_KEY }}" --build-arg REFRESH_PUBLIC_KEY="${{ secrets.REFRESH_PUBLIC_KEY }}" --build-arg MONGODB_CONNECTION_STRING_AUTH="${{ secrets.MONGODB_CONNECTION_STRING_AUTH }}" --build-arg GOOGLE_CLIENT_ID="${{ secrets.GOOGLE_CLIENT_ID }}" --build-arg GOOGLE_CLIENT_SECRET="${{ secrets.GOOGLE_CLIENT_SECRET }}" ./WOA/auth-microservice
```

From there the ENV variable is generated from the ARG build-time variable


europe-west4-docker.pkg.dev > woa-dev-389512 > demo

Repository Details

Format	Docker
Type	Standard

Filter Enter property name or value

Name	Created	Updated
auth	23 hours ago	2 hours ago
client	23 hours ago	2 hours ago
notifications	23 hours ago	2 hours ago
postfeed	23 hours ago	2 hours ago

Figure 25 - Google Artifact Registry with pushed Docker images

After GitHub actions pushes the Docker image to the Artifact Registry the images are created. If they already exist they are updated with the **latest** tag. It's time for the final step of the deployment process:

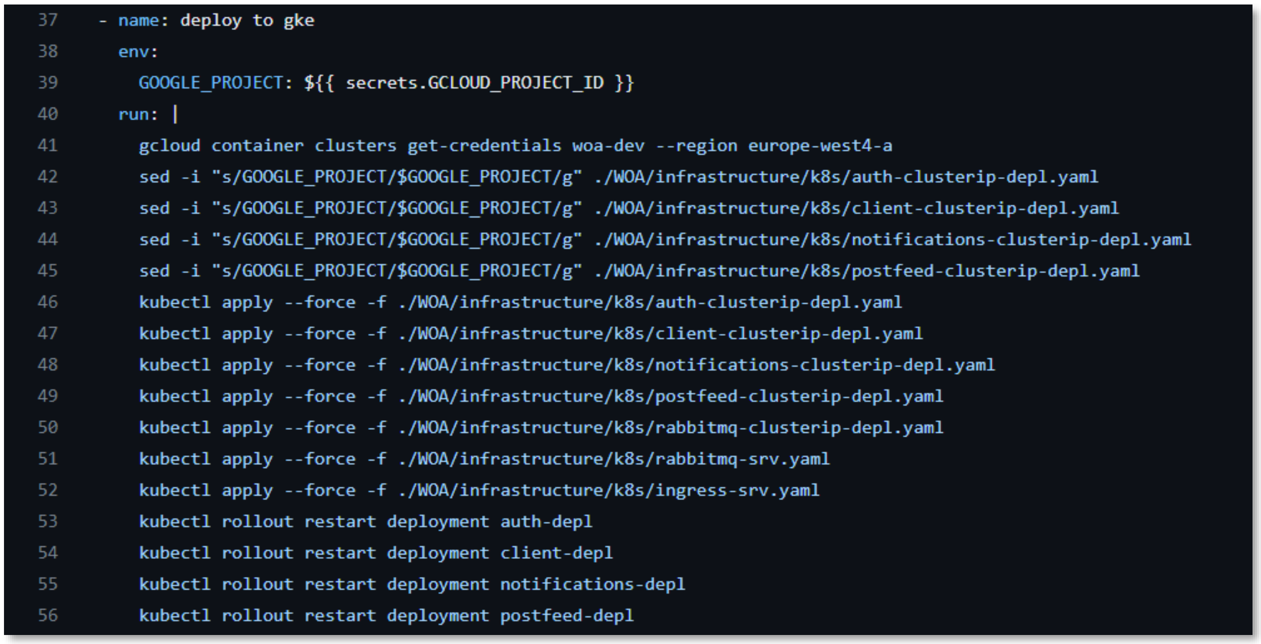

```
- name: deploy to gke
  env:
    GOOGLE_PROJECT: ${{ secrets.GCLOUD_PROJECT_ID }}
  run: |
    gcloud container clusters get-credentials woa-dev --region europe-west4-a
    sed -i "s/GOOGLE_PROJECT/$GOOGLE_PROJECT/g" ./WOA/infrastructure/k8s/auth-clusterip-depl.yaml
    sed -i "s/GOOGLE_PROJECT/$GOOGLE_PROJECT/g" ./WOA/infrastructure/k8s/client-clusterip-depl.yaml
    sed -i "s/GOOGLE_PROJECT/$GOOGLE_PROJECT/g" ./WOA/infrastructure/k8s/notifications-clusterip-depl.yaml
    sed -i "s/GOOGLE_PROJECT/$GOOGLE_PROJECT/g" ./WOA/infrastructure/k8s/postfeed-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/auth-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/client-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/notifications-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/postfeed-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/rabbitmq-clusterip-depl.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/rabbitmq-srv.yaml
    kubectl apply --force -f ./WOA/infrastructure/k8s/ingress-srv.yaml
    kubectl rollout restart deployment auth-depl
    kubectl rollout restart deployment client-depl
    kubectl rollout restart deployment notifications-depl
    kubectl rollout restart deployment postfeed-depl
```

Figure 26 - Deploying the images to Google Kubernetes Engine using .yaml k8s manifests.

Let’s look at a .yaml manifest file:

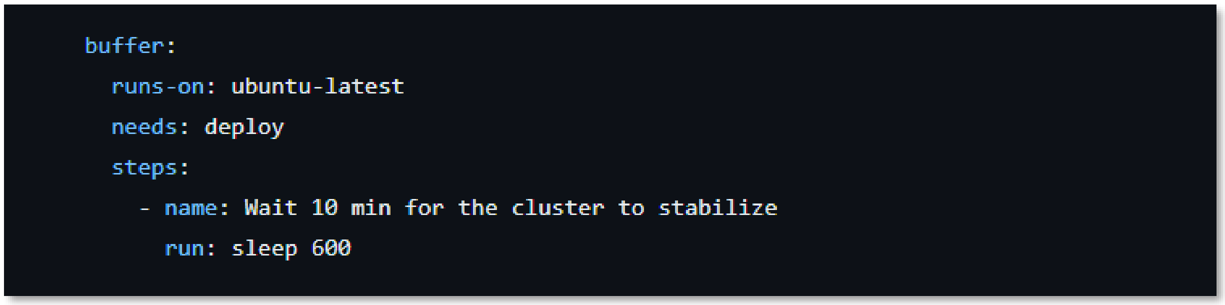
```yaml
#prettier-ignore
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: europe-west4-docker.pkg.dev/GOOGLE_PROJECT/demo/auth
          imagePullPolicy: Always
          resources:
            requests:
              cpu: 200m
              memory: 200Mi
            limits:
              cpu: 300m
              memory: 300Mi
---
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: auth-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: auth-depl
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
---
apiVersion: v1
kind: Service
metadata:
  name: auth-clusterip-srv
spec:
  selector:
    app: auth
  type: ClusterIP
  ports:
    - name: auth
      protocol: TCP
      port: 5000
      targetPort: 5000
```

The **.yaml** file contains the deployment using the artifact registry image. The **GOOGLE_PROJECT** variable is generated from the command **sed -I *** which performs an in-place editing of the **.yaml** manifest and replaces the value with the name of the project inside Google Cloud Platform.

The file also contains an autoscaling configuration so that when deployed, the pod can scale horizontally from 1 to up to 10 pods at the same time.

Finally the service is configured which exposes the container’s port 5000 and make sit accessible to other pods in the cluster.

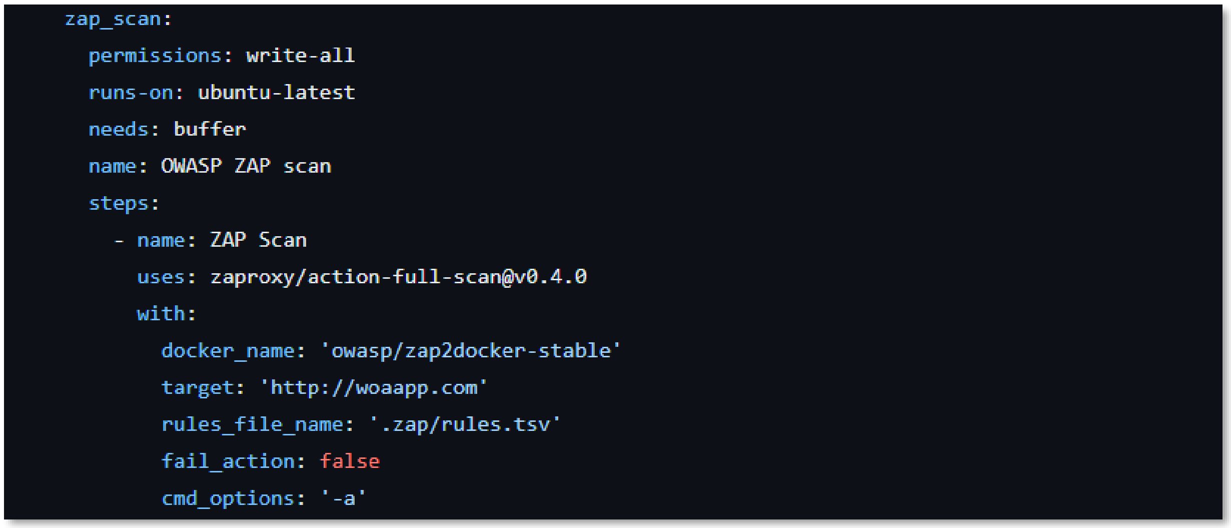
After this the action deploys the .yaml file to the cluster where it downloads the image from Artifact Registry and the deployment is complete.



```yaml
  buffer:
    runs-on: ubuntu-latest
    needs: deploy
    steps:
      - name: Wait 10 min for the cluster to stabilize
        run: sleep 600
```

Figure 27 - The next job in the GitHub action - buffer.

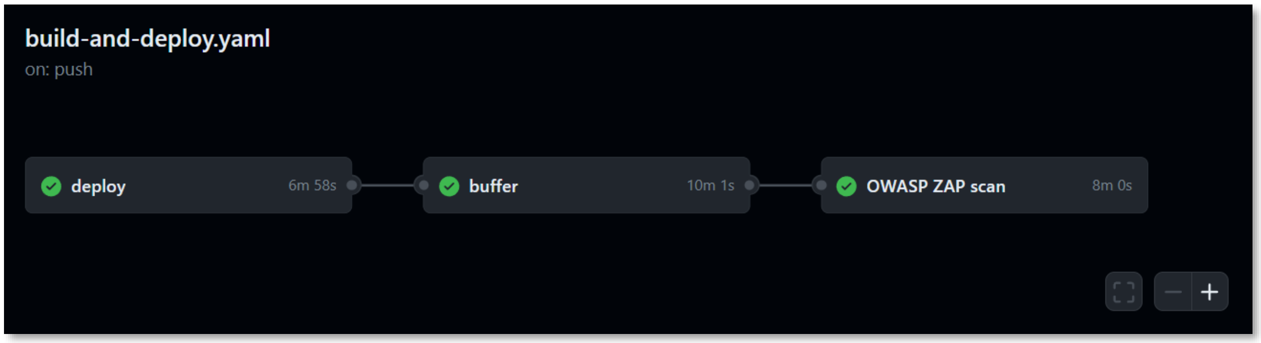
The cluster takes about 10 minutes to completely stabilize after a deployment. This is why the next job waits for the deployment process to finish and immediately triggers a 10 min buffer that waits for the cluster to finish setting up and scaling down.



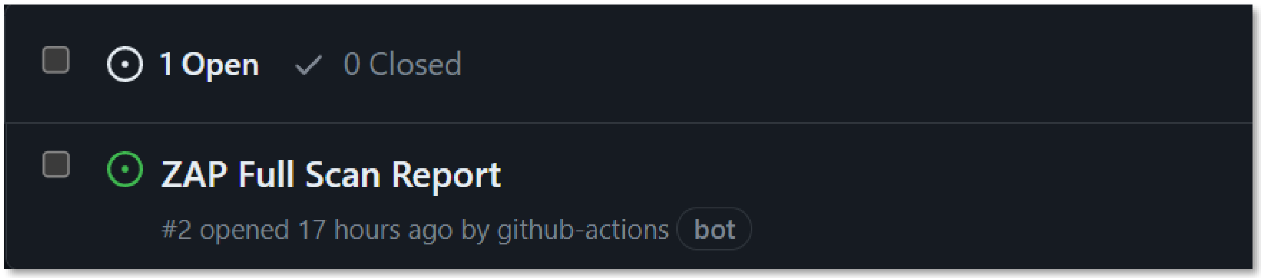
```yaml
  zap_scan:
    permissions: write-all
    runs-on: ubuntu-latest
    needs: buffer
    name: OWASP ZAP scan
    steps:
      - name: ZAP Scan
        uses: zaproxy/action-full-scan@v0.4.0
        with:
          docker_name: 'owasp/zap2docker-stable'
          target: 'http://woaapp.com'
          rules_file_name: '.zap/rules.tsv'
          fail_action: false
          cmd_options: '-a'
```

Figure 28 - The last job in the pipeline - zap_scan

The last job in the pipeline waits for the buffer to finish the 10 min countdown and assuming the cluster has calmed down triggers an OWASP ZAP dynamic application security testing scan. As soon as the scan is over any new found vulnerability or issue is automatically added to the issue board to be looked at by the developers.



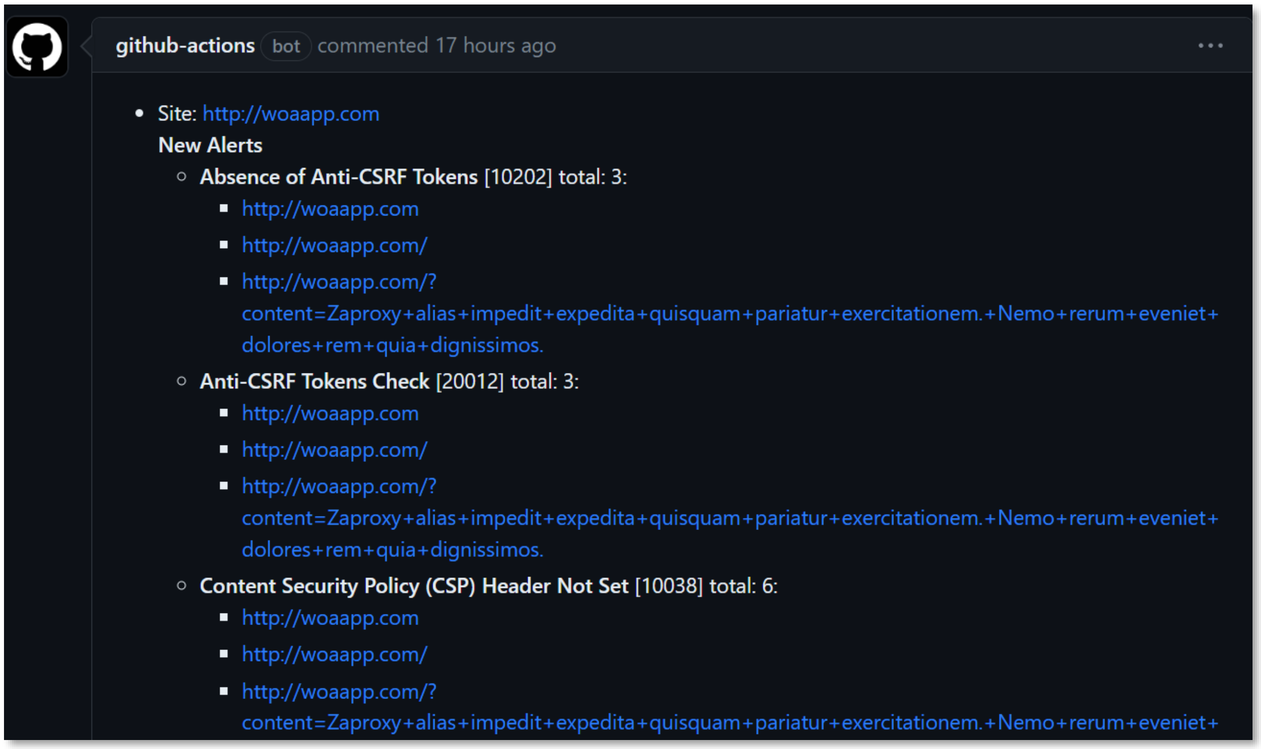
In the issues tab an OWASP ZAP Full Scan Report is generated:



1 Open ✓ 0 Closed

ZAP Full Scan Report

#2 opened 17 hours ago by github-actions bot



github-actions bot commented 17 hours ago

- Site: http://woaapp.com
  New Alerts
  - **Absence of Anti-CSRF Tokens** [10202] total: 3:
    - http://woaapp.com
    - http://woaapp.com/
    - http://woaapp.com/?content=Zaproxy+alias+impedit+expedita+quisquam+pariatur+exercitationem.+Nemo+rerum+eveniet+dolores+rem+quia+dignissimos.
  - **Anti-CSRF Tokens Check** [20012] total: 3:
    - http://woaapp.com
    - http://woaapp.com/
    - http://woaapp.com/?content=Zaproxy+alias+impedit+expedita+quisquam+pariatur+exercitationem.+Nemo+rerum+eveniet+dolores+rem+quia+dignissimos.
  - **Content Security Policy (CSP) Header Not Set** [10038] total: 6:
    - http://woaapp.com
    - http://woaapp.com/
    - http://woaapp.com/?content=Zaproxy+alias+impedit+expedita+quisquam+pariatur+exercitationem.+Nemo+rerum+eveniet+

Figure 29 - Part of the OWAS ZAP report

The OWASP ZAP report shows the type of vulnerability on the URL that it was found on.

4. Conclusion

In conclusion, this report has emphasized the importance of integrating automated security testing into DevSecOps pipelines. By leveraging automation tools and techniques, organizations can overcome the limitations of manual testing and enhance their ability to identify critical vulnerabilities and threats. This approach ensures consistent and robust security throughout the software development lifecycle, even in the fast-paced DevOps environment. Adopting automated security testing is crucial for maintaining adequate security measures while delivering software at an accelerated pace. It minimizes human error, improves testing speed and accuracy, and enables organizations to proactively address security concerns from the beginning. Overall, automated security testing is a key element in achieving secure and reliable software applications in today's digital realm.

REFERENCES

1. Froehlich, A. (2022). 8 benefits of DevSecOps automation. *Security*. https://www.techtarget.com/searchsecurity/tip/8-benefits-of-DevSecOps-automation
2. Moyle, E. (2021). 5 ways to automate security testing in DevSecOps. *Security*. https://www.techtarget.com/searchsecurity/tip/5-ways-to-automate-security-testing-in-DevSecOps
3. Aqua Security. (2023, April 3). *DevSecOps Tools: 9 Ways to Integrate Security Into the SDLC*. Aqua. https://www.aquasec.com/cloud-native-academy/devsecops/devsecops-tools/
4. Snyk. (2021). Dynamic Application Security Testing (DAST). *Snyk*. https://snyk.io/learn/application-security/dast-dynamic-application-security-testing/
5. Snyk. (2022). Guide to Software Composition Analysis (SCA). *Snyk*. https://snyk.io/series/open-source-security/software-composition-analysis-sca/
6. Snyk. (2021b). Everything You Need to Know About Container Scanning. *Snyk*. https://snyk.io/learn/container-security/container-scanning/
7. Odogwu, C. (2022). What Is Container Security and Why Do You Need It? *MUO*. https://www.makeuseof.com/what-is-container-security/
8. Hamilton, T. (2023). What is Test Driven Development (TDD)? Example. *Guru99*. https://www.guru99.com/test-driven-development.html
9. *What Is SAST and How Does Static Code Analysis Work? | Synopsys*. (n.d.-b). https://www.synopsys.com/glossary/what-is-sast.html
10. Mend.io. (2023, May 11). *Software Composition Analysis Explained | Mend*. Mend. https://www.mend.io/resources/blog/software-composition-analysis/

11. Shrikanth, B. (2022). Role of Automation Testing in CI/CD | BrowserStack. *BrowserStack*. https://www.browserstack.com/guide/role-of-automation-testing-in-ci-cd
12. Kumar, M. (2020). Automated Vs Manual Web Application Security Testing. *Aeologic Blog*. https://www.aeologic.com/blog/automated-vs-manual-web-application-security-testing/
13. Jayati. (2019, September 5). Fusing Security and Compliance in CI/CD Pipelines. *OpenSense Labs*. https://opensenselabs.com/blog/articles/fusing-security-compliance-cicd-pipelines
14. *Source Code Analysis Tools | OWASP Foundation*. (n.d.). https://owasp.org/www-community/Source_Code_Analysis_Tools
15. Mocha vs Jest Comparison of Testing Tools in 2022. (2022, March 10). *Mocha vs Jest Comparison of Testing Tools in 2022*. https://www.blog.duomly.com/mocha-vs-jest/#3-compare-mocha-vs-jest
16. *Testing Asynchronous Code · Jest*. (2023, March 6). https://jestjs.io/docs/asynchronous
17. *OWASP ZAP – Documentation*. (n.d.). https://www.zaproxy.org/docs/
18. *Pricing - Burp Suite Enterprise Edition*. (n.d.). PortSwigger. https://portswigger.net/burp/enterprise/pricing
19. Snyk. (n.d.). *Open Source Security Management | SCA Tool | Snyk*. https://snyk.io/product/open-source-security-management/