# Автоматизация

## GitHub Actions

GitHub Actions е платформа за автоматизация на работни потоци, интегрирана в GitHub. Тя позволява на разработчиците да автоматизират процесите по изграждане, тестване и разгръщане на техния код директно в хранилищата им. Това се постига чрез използване на "действия" (actions), които представляват индивидуални задачи, дефинирани в YAML файлове, наречени "работни потоци" (workflows).

#### Основни характеристики на GitHub Actions

1. **Автоматизация на CI/CD**: GitHub Actions поддържа Continuous Integration (CI) и Continuous Deployment (CD), което позволява на разработчиците автоматично да тестват и разгръщат своя код при всяка промяна.
2. **Множество среди за изпълнение**: Платформата предоставя възможност за изпълнение на действия в различни среди, включително Linux, macOS и Windows.
3. **Гъвкавост и мащабируемост**: GitHub Actions поддържа паралелно изпълнение на задачи и позволява създаването на сложни работни потоци с условни логики, които могат да се мащабират в зависимост от нуждите на проекта.
4. **Интеграция с GitHub**: Платформата е дълбоко интегрирана с GitHub, което улеснява управлението на работните потоци директно от хранилището на проекта и използването на събития като комити, pull requests и издания като тригери за изпълнение на действия.

### GitHub Actions: Примерен работен поток

Работният поток, дефиниран по-долу, автоматизира процеса на изграждане на проект с Gradle и качване на резултатните артефакти в хранилището. Този работен поток е настроен да се изпълнява при всеки push или pull request към основния клон на хранилището.

Дефиниция на работния поток



Фиг. IXI. Скриптов код използван за github action в репозитория volts-server

### Обяснение на работния поток

**Име на работния поток**:

name: Gradle-build

Работният поток е именуван "Gradle-build", което указва, че основната му цел е изграждане на проект с Gradle.

**Тригери за изпълнение**:

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

Работният поток се изпълнява при всеки push или създаване на pull request към основния клон ("main") на хранилището.

**Дефиниране на работа**:

jobs:

build-without-cache:

runs-on: ubuntu-latest

Определя се една работа с име "build-without-cache", която ще се изпълнява на най-новата версия на Ubuntu.

**Стъпки на работа**: Работният поток съдържа няколко стъпки, които се изпълняват последователно:

**Изтегляне на кода**:

yaml

Copy code

- uses: actions/checkout@v2

Тази стъпка използва предварително дефинирано действие за изтегляне на кода от хранилището.

**Настройка на JDK 17**:

- name: Set up JDK 17

uses: actions/setup-java@v2

with:

java-version: 17

distribution: 'adopt'

Тази стъпка настройва JDK версия 17, използвайки AdoptOpenJDK дистрибуцията. Това е същото JDK което се използва в конфигурационния файл на главния проект. Спазва се еднаквост при всичко скриптове въпреки че са различни скриптове, които се сами по себе си се изпълняват на различни машини или виртуални машини.

**Даване на права за изпълнение на gradlew**:

- name: Grant execute permission for gradlew

working-directory: volts-server

run: chmod +x gradlew

Тази стъпка дава права за изпълнение на скрипта gradlew.

**Изграждане с Gradle**:

- name: Build with Gradle

working-directory: volts-server

run: ./gradlew build

Тази стъпка изпълнява командата за изграждане на проекта с Gradle.

**Изброяване на файловете в директорията за изграждане**:

- name: List build directory

working-directory: volts-server

run: ls -la build/libs

Тази стъпка изброява съдържанието на директорията build/libs, за да покаже изградените артефакти.

**Качване на JAR файла**:

- name: Upload JAR file

uses: actions/upload-artifact@v3

with:

working-directory: volts-server

name: my-app-build

path: build/libs/\*.jar

Тази стъпка качва изградените JAR файлове като артефакти на работния поток.

**Комитване на JAR файла в хранилището**:

- name: Commit JAR file to repository

working-directory: volts-server

run: |

git add build/libs/\*.jar

git commit -m "Add build artifacts"

git push origin main

env:

GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

Тази стъпка комитва изградените JAR файлове в хранилището, използвайки конфигурирано GitHub потребителско име и имейл. Тук се използва вътрешната променлива за пазене на таен текст secrets.GITHUB\_TOKEN. Проекта представлява public repository тоест всеки може да го погледанe, да го изтегли и да предложи промени по кода. Затова когато имаме секретни ключове за достъп до личните ресурси на проекта те трябва да се пазят в тайма. Затова платформата github предоставя възможността за скриване на ценна информация под формата на secret. В този случай е създадена тайна която пази ключ за достъп до ресурсите на проекта. Github само по себе си когато срещне подобна тайна при изпълнение ще бъде подменена с тайният текст.

Този примерен работен поток илюстрира как GitHub Actions може да бъде използван за автоматизация на процесите по изграждане и публикуване на артефакти в софтуерен проект. Автоматизирането на тези процеси не само спестява време и усилия, но и гарантира последователност и надеждност в цикъла на разработка. В проекта е използван за всяка репозитория github actions поради лекотата от използването им и екосистемата на github.

# Комуникационен портал на приложението

За да се достъпи приложението от различните интерфейси то трябва всяка заявка да премине през комуникативния портал. Тапи методология е избрана поради множествено положителни показатели. Тези положителни показатели включват: единен вход което води до единни проверки за коректност, превод от rest метода на комуникиране по-бързия разширен протокол за опашка за съобщения( Advanced Message Queuing Protocol- AMQP)

# Защитен сервиз

## Автентикация

Автентикирането на потребители се случва на ниво защитен сервиз и включва потвърждаване на потребителя чрез емайл/потребителски име и парола. След което се създава JWT( Json Web Token) токен. За да се случи това потвърждаване на потребителската самоличност първо трябва да се приемат неговите данни и това е възможно чрез следния бизнес обект:

public record UserLogIn(String credentials, String password) {}

Който съдържа и поле за идентификация и поле за парола. Което поле за идентификация представлява или емайл или потребителско име и поред вина на полета се приемат два различни начина на идентифициране. При приемате на данните те минават през верифициране на типа на данните, това се отнася за всяка заявка към сървъра. При

## Авторизация

Авторизацията в уеб приложения се използва за установяване на идентичността на потребителите и определянето на техните права и разрешения за достъп до определени ресурси или функционалности в приложението. Авторизацията осигурява защита на чувствителните данни и ресурси, като позволява само на упълномощени потребители да ги достъпват. Системата за авторизация определя кои потребители имат право да използват определени функционалности или да виждат определени данни в приложението. Авторизацията позволява на приложението да запази информацията за влизането на потребителя в рамките на сесията, което дава възможност за персонализирани функционалности и оптимизация на потребителския опит. Системата за авторизация поддържа аудитни логове, които записват дейността на потребителите в приложението, като това помага при проследяване на проблеми или съдейства при изследване на сигурностни нарушения. Авторизацията играе ключова роля в управлението на идентичността на потребителите, включително аутентикацията, управлението на пароли и обновяването на правата за достъп. Общо казано, авторизацията в уеб приложения е важен механизъм за сигурност и управление на достъпа, който осигурява защита на данните и контролира потребителския достъп до ресурсите на приложението.

## OAuth

OAuth 2.0 е индустриалният стандартен протокол за оторизация. OAuth 2.0 се фокусира върху опростеността на разработчиците на клиенти, като същевременно предоставя специфични потоци за оторизация за уеб приложения, настолни приложения, мобилни телефони и устройства за всекидневна. Тази спецификация и нейните разширения се разработват в рамките на IETF OAuth Working Group. Протоколът за уеб авторизация (OAuth) позволява на потребителя да предостави a достъпът на уеб сайт или приложение на трета страна до защитения потребител ресурси, без непременно да разкриват своите дългосрочни пълномощия, или дори самоличността им. Например сайт за споделяне на снимки, който поддържа OAuth, може да позволи на своите потребители да използват мрежа за печат на трета страна сайт за отпечатване на личните им снимки, без да позволява отпечатването сайт, за да получите пълен контрол върху акаунта на потребителя и без да имате потребител споделя дългосрочните идентификационни данни на своите сайтове за споделяне на снимки със сайтът за печат. Тоест употребата му за приложението ще е приложимо при вписване от трета страна като на пример Google акаунт или Microsoft акаунт. Това улеснява първоначално навлиза в приложението. Така клиента има едно по малко препятствие за регистриране и употребяване на приложението. В днешните дни където съществуват множествено приложения, които вършат множествено услуги, се конкурира за вниманието на клиента си. Повече внимание към едно приложение се транслира директно към повече приходи за това приложение. И когато има нисък праг за приемане на ново приложение то има по-голям шанс да останат и да употребяват приложението. Така се подхожда по-тактично към приветстването на нов потребител. Пакетът протоколи OAuth 2.0 вече включва

* процедура за позволяване на клиент да се регистрира с разрешение сървър,
* протокол за получаване на токени за оторизация от оторизация сървър със съгласието на собственика на ресурса и
* протоколи за представяне на тези токени за оторизация на protected ресурси за достъп до ресурс.

Използвайки един протокол предоставя множествено функционалности. Така се установява единен работен протокол за безопасност на данните. Когато има единен протокол за безопасност предоставя и че помежду екипи и програмисти има един стандарт, който трябва да следва, осланявайки работния процес. Протокола предоставя и създаването на множествено токени за персистиране на сесии с клиента. За приложението е употребен стандарта JWT(Json Web Token) . Извора за използване на JWT токени лежи в факта че е универсален и лесен начин за менажиране на сесии помежду сървъра и потребителския интерфейс ( Фиг. №: LXL). Както е разгледано в следващата под точка, се знае че съхраняването на тези токени е реализирано в база от данни тип библиотека, която сама по себе си е изключителна бърза да вземане и писане на данни. Това означава че менажирането на сесии е изключително бързо поради избраните технологии за съхраняване и автентикация и авторизация.



Фиг. №: LXL- Изображение на как е реализиран модела за създаване на потребителски токен.

Криптиране на пароли

## База от данни отговорна за потребителски сесии

За запазване сесиите на активните потребители в момента се използва база от данни тип библиотека( dictionary) или още позната като ключ със стойност( key value pair). И конкретно тази база от данни тип библиотека е пряко и единствено сервиза със сервиза отговорен за защита(ink-security). Този сервиз както е разгледано в предишните подточки е отговорен за защитата на потребителските данни и общата безопасност на системата от нападения. Това не значи че другите сервизи не спазват традиционните практики за защита от атаки. Напротив те трябва да следват многослойни процедури и техники за избягване на масивни или централизирани атаки, като например DDOS( Denial-of-service attack). Но самите сервизи не трябва да отговарят дали дадената потребителска сесия е активна, това е работа на сервиза за защита който автентификация и авторизация потребителя да има достъп до дадените сервизи. Затова е нужна база която да съхранява времената информация за потребителя, неговата авторизация и до колко е валидна сесията.

Относно базата използвана за реализация на каширане на сесиите на потребителите е използвана дистанционно разгъната библиотечна база от данни Redis. Трябва да се уточни че се използва библиотечната функция на Redis, тъй като Redis поддържа и множествени други специализирани режими на съхраняване на данни. Други функционалности на Redis включват и: база от данни подходяща за търсачи (Search data base), съхраняване в стандарт JSON, граф бази от данни, таблична база от данни, клъстър база от данни и много други които могат да се комбинират и споделят информация помежду си. Но за целта на реализация на каширане на потребителски сесии сме използвали библиотечните функции на платформата. Redis е лидер относно този вид съхраняване на данни и е използван от най-големите софтуерни фирми да забързат свалянето на често използвани данни от традиционни релационни бази от данни. Като пример е социалната мрежа Twitter които използват Redis за да забързан предаване на най-популярните публикации на платформата.



Фиг. XX: Графика показваща цялостна бързина спрямо релационни и не релационни бази от данни.

В java се поддържа свързаност с базата от данни чрез библиотеката наречена ‚ jedis‘ и е създадена от същите създатели на базата от данни. Задава се нова инфомация към базата като след като първо се реализира връзка към базата от данни чрез:

*JedisPool pool = new JedisPool("localhost", 6379);*

След усъществена връзка се преминава към вземане, добавяне, променяне или изтриване на хеш от базата. Пример за вписване на данни в базата:

*jedis.hset("user-session:123", hash);*

# Микро сервизи

## Сервиз за пренасочване на входни заявки ( Volts-gate)

## Сервиз за сигурност ( Ink-security)

## Сервиз за анализ ( Volts-analytics)

## Сервиз за изкуствен интелект (Volts-ai)

# Източници

Работа с база от данни Redis и java: <https://redis.io/docs/connect/clients/java/>

# Приложение

## Входни точки (End points)

### Сервиз за сигурност (Ink-security)

Вписване като потребител:

Post

/auth/..

Body:

{

‘username’:’user1’,

‘password’:’!pasword12345678’

}

Регистриране като потребител

Вземане на потребителски данни

Пропоръчване