ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

Катедра „Софтуерни и интернет технологии“

A blue and white logo

Description automatically generated with medium confidence

**ДИПЛОМЕН ПРОЕКТ**

за придобиване на ОКС „Магистър”

**Тема:**

**ПРОЕКТИРАНЕ И РАЗРАБОТВАНЕ НА СИСТЕМА ЗА КОНТРОЛ НА КЛИЕНТИТЕ\*\*\*\*\*\*\*ОПРАВИ**

**Дипломант:**  **Научен ръководител:**

Ивайло Пламенов Руменов  Доц. Христо Ненов

**Специалност** Софтуерно инжинерство

**Фак. № 23651227**

# Увод

# Изложение

## Обзор на използваните програмни средства и технологии.

### Организация на работния процес

Организацията на работния процес при разработването на микросервизен софтуер в Spring Boot екосистемата е ключова част от реализацията на проекта като цяло. Той представлява структуриран подход за управление на разработката, тестването, внедряването и поддръжката на отделни, автономни услуги, които комуникират помежду си. Този процес включва разделяне на проекта на малки, независими микросервизи, всеки от които отговаря за конкретна бизнес функционалност и може да бъде разработван, мащабиран и обновяван независимо от останалите.

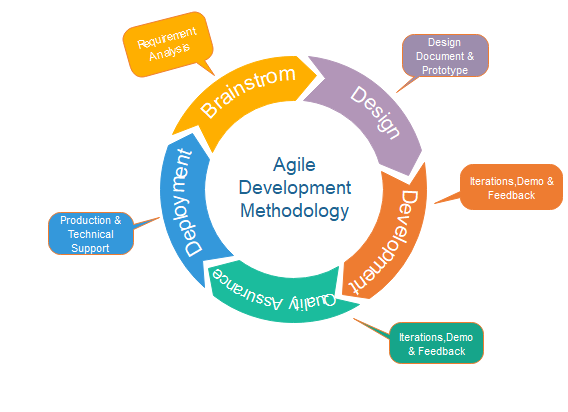
При организацията на работния процес, екипът прилага принципи като CI/CD (непрекъсната интеграция и доставка), автоматизирано тестване и контейнеризация, за да осигури гъвкавост и последователност в процесите. Работата често се разделя на малки, кросфункционални екипи, които отговарят за различни микросервизи. Spring Boot предоставя богата екосистема от инструменти и модули, като Spring Cloud за конфигурация и управление на микросервизи и Spring Security за защита, което улеснява изграждането на мащабируеми и сигурни приложения.

Организацията на работния процес в микросервизна архитектура позволява бърза адаптация към нови изисквания и лесно поддържане на различни версии на услугите, като същевременно подобрява производителността и намалява времето за разработка. Затова е избрана Agile методологията за разработване на всеки елемент на цялостната система.

Agile представлява методология за управление на проекти, създадена през 2001 г. от група софтуерни разработчици, които формулират своя подход в Agile Manifesto[[1]](#footnote-1). Тази методология се фокусира върху гъвкаво и адаптивно управление на задачи, с цел да се реагира бързо на променящите се изисквания и нужди на клиента. Agile се основава на кратки, итеративни цикли на разработка, наречени „спринтове“, които обикновено продължават от една до четири седмици. В края на всеки спринт се представя работеща част от продукта, което позволява постоянна обратна връзка от клиента и навременни корекции.

Agile се базира на четири основни принципа, формулирани в манифеста:

1. Приоритет на хората и взаимодействията пред процеси и инструменти.
2. Работещ софтуер пред изчерпателна документация.
3. Сътрудничество с клиента пред договаряне на изисквания.
4. Готовност за промяна пред следване на предварително начертан план.



Фиг ЙЙМ: Диаграма обобщаваща прилагането на Agile в разработката на софтуер. Източник на диаграмата javapoin.com

Предимства на Agile:

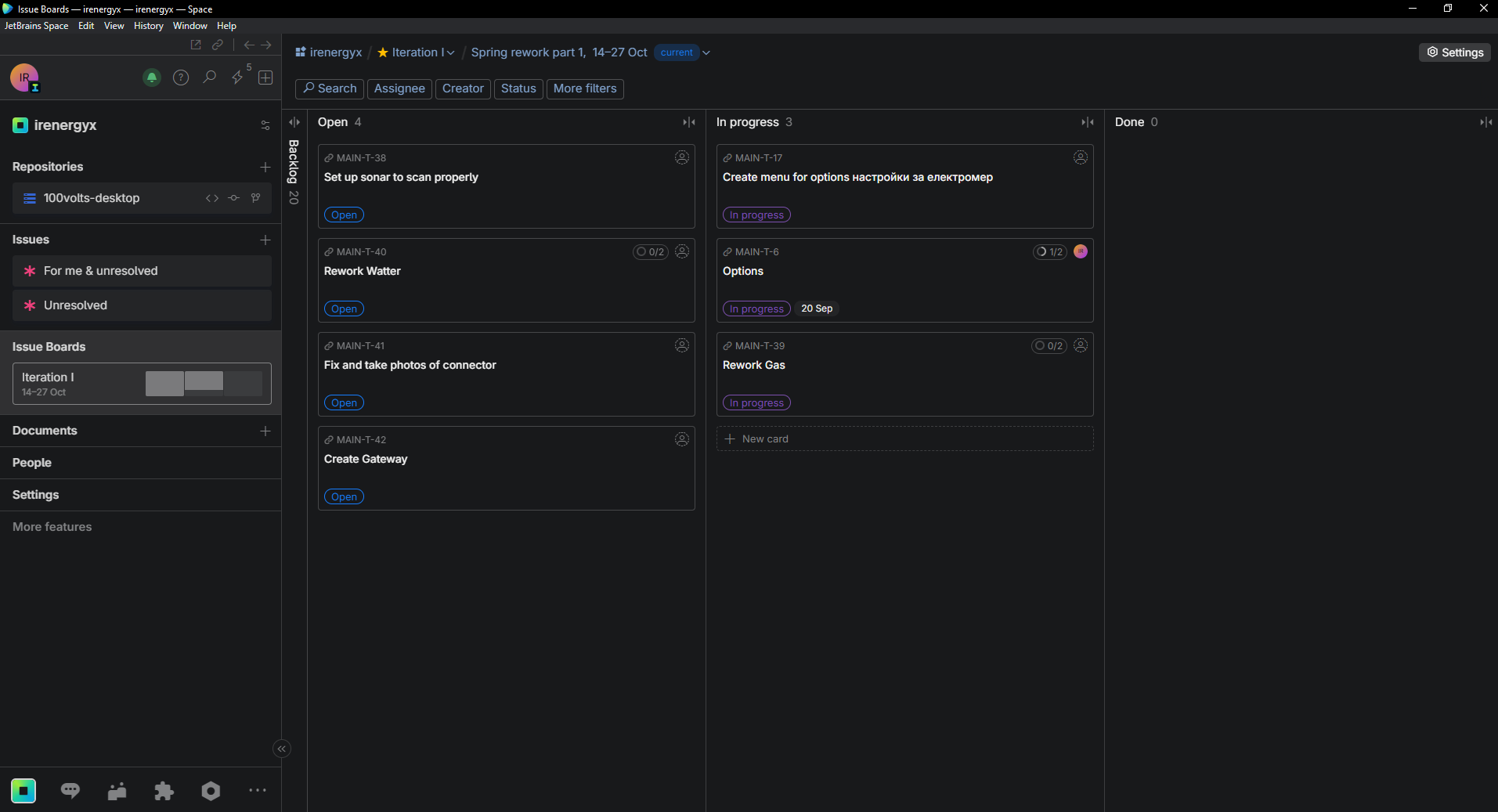
* Адаптивност: Agile позволява на екипа да се адаптира бързо към промени в изискванията, което е особено полезно в динамични и бързо развиващи се проекти.
* Постоянна обратна връзка: Благодарение на итеративните цикли, клиентът редовно вижда прогреса на продукта и може да дава обратна връзка, което помага за бързо идентифициране на нуждите.
* Увеличена ангажираност на екипа: Agile насърчава кросфункционално сътрудничество и участие на всички членове на екипа, като подобрява комуникацията и споделената отговорност. В рамките на проекта се е работело от едно лице но в бъдеще този проекта ще има свободата да се разшири.
* По-нисък риск: Кратките цикли на работа и постоянната обратна връзка намаляват риска от непредвидени проблеми, които могат да забавят или усложнят проекта.

Недостатъци на Agile:

* Неясна крайна цел: Поради адаптивността към промените, често липсва ясно дефинирана крайна визия за проекта, което може да създаде затруднения в планирането и управлението.
* Трудности при мащабни проекти: Agile е подходяща за малки и средни екипи, но при големи проекти и екипи координацията може да стане по-сложна.
* Изисква значителен ангажимент от клиента: Успешното прилагане на Agile зависи от постоянната ангажираност на клиента, който трябва редовно да предоставя обратна връзка и да участва в планирането на следващите стъпки.
* Повече административна работа: Agile изисква редовни срещи, като дневни „stand-up“ срещи и планиращи спринтове, което може да увеличи административното натоварване за екипа.

При реализиране на множество задачи по проект, Agile методологията предоставя ясна структура и редовни прегледи на напредъка, като позволява на екипа да раздели работата на по-малки задачи и да се фокусира върху краткосрочни цели. Това осигурява гъвкав подход, който дава възможност за бързи промени в зависимост от нуждите на клиента и условията на пазара.

Но за прилагане на Agile в реален проект има нужда от начин за следене на всички правила че се имплантират и да се следи качеството и времето което се отделя за разработване на проекта. За това е избран софтуер Spaces разработена от JetBrains, популярни с продукти като IntelliJ за разработка на Java проекти и PyCharm за разработка на Python[[2]](#footnote-2) проекти. JetBrains Spaces е платформа за интегрирано управление на проекти, която предоставя инструменти за сътрудничество и координация на екипи в една среда. Spaces се използва за разработка на софтуер, като обединява функции като управление на задачи, хранилища за код, автоматизация на CI/CD, документация, и комуникационен чат. Spaces позволява на екипите да планират, проследяват, и изпълняват задачи на едно място, улеснявайки организацията на работния процес и ускорявайки разработката.



Фиг. НКК: Екранна снимка на работни задачи от двуседмичен спринт.

Предимства на JetBrains Spaces:

* Цялостна платформа: Spaces събира инструменти за планиране, писане на код и комуникация в една среда, което намалява нуждата от множество външни инструменти.
* Подобрена комуникация: Spaces включва интегриран чат и коментари в задачи и кодови прегледи, което улеснява комуникацията между членовете на екипа.
* Интеграция с CI/CD: Вградената автоматизация на CI/CD позволява на екипите да внедряват и тестват промени по-бързо, поддържайки гъвкав процес на разработка.
* Поддръжка на Git хранилища: Spaces предлага интегрирано хранилище за код, което поддържа Git, позволявайки на екипа лесен достъп до кода и версия контрол.

Недостатъци на JetBrains Spaces:

* Изисква адаптация: Spaces изисква екипите да се адаптират към нова среда, което може да бъде трудно за организации, свикнали с различни инструменти.
* Ресурсоемкост: Работата с Spaces може да изисква повече ресурси за ефективно използване, особено при по-големи екипи и проекти.
* Зависимост от екосистемата на JetBrains: Spaces се интегрира най-добре с други продукти на JetBrains, което може да ограничи избора на инструменти за екипи, които използват различен софтуер.

Spaces подкрепя Agile методологията, като предлага възможности за управление на задачи, спринтове и последователно проследяване на напредъка в единна платформа. С функционалностите за комуникация и интегрираното планиране, Spaces улеснява екипната работа, бързата адаптация към промени и регулярната обратна връзка — ключови аспекти на Agile.

Цялостно spaces служи за менажиране на всичките спринтове в проекта. Документирайки цялостния процес на работа по проекта. Софтуера предоставя нужните инструменти за прилагане на Agile митологията и служи за всичките изисквания по проекта.

### Документиране на проекта

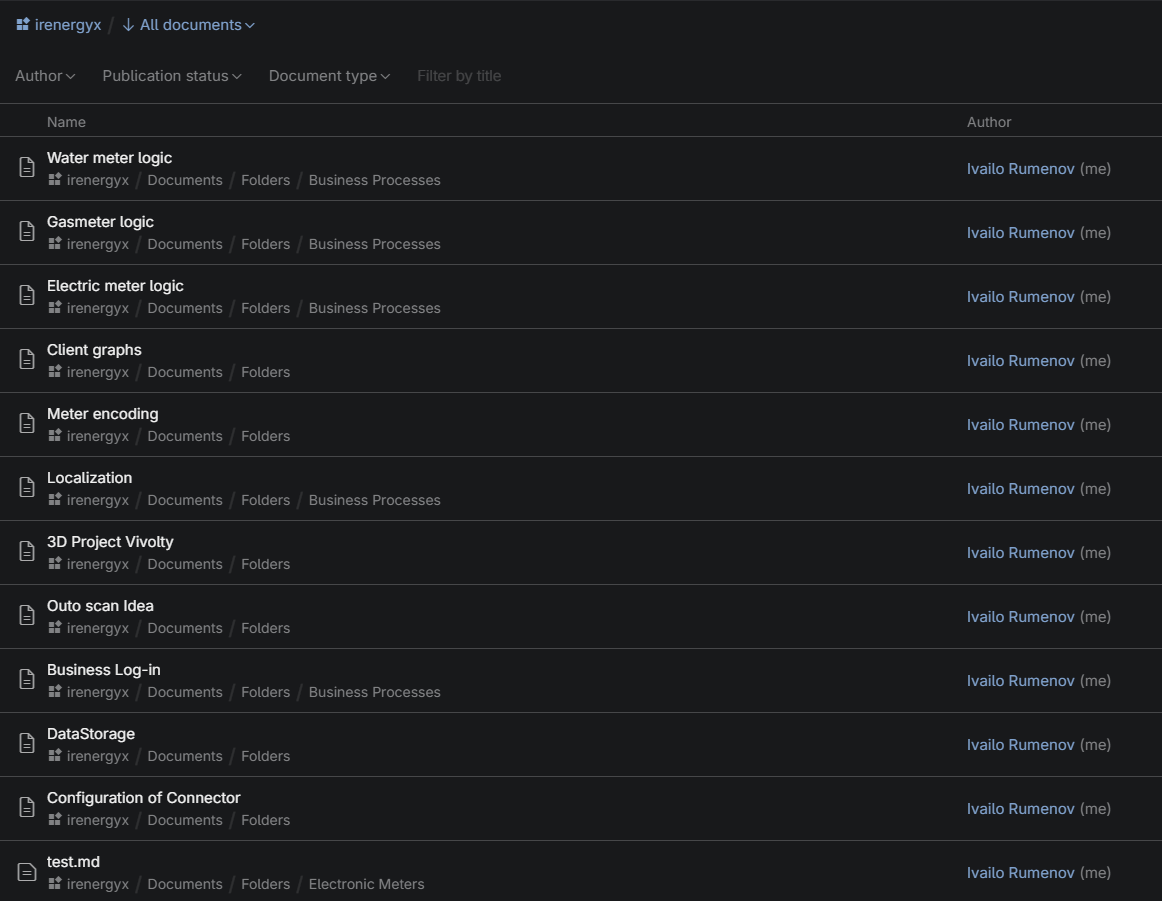
Уеб софтуерната документация представлява съвкупност от записи, които описват структурата, функционалностите, технологиите и начините за използване на дадено уеб приложение. Тя служи като ръководство за всички, които участват в разработката и поддръжката на проекта, включително програмисти, тестери, проектни мениджъри и крайни потребители. Тази документация помага на екипа да разбере как функционира приложението, как трябва да се използва и как се очаква да се развива.

Необходимостта от поддържане на документацията е ключова, защото с развитието на проекта тя отразява всички промени и актуализации, което гарантира точност и яснота за текущото състояние на софтуера. Агресивното поддържане от целия екип е важно, защото своевременното обновяване на документацията предотвратява разминавания между реалния код и неговото описание. Това позволява на нови членове на екипа да се включат по-бързо и улеснява комуникацията между различни роли в проекта.

Липсата на поддръжка на документацията може да доведе до сериозни последици. Пропуски или неточности в документацията създават объркване и затруднения при разширяване или поддръжка на проекта, увеличават риска от грешки и водят до загуба на време при опити за разбиране на неактуална информация.

За целите на проекта документация е разделена на описване на функционалността в кода, описване на бизнес логиката в Agile софтуера и описване на всички крайни точки в Astro софтуера. Това са три различни вида документация и са нужни за успешната реализация на който и да е проект. Повечето случай ще има разминаване в тази документация. Това е така защото бизнес логиката никога няма да отговаря на програмната логика и ще се нуждае от сходна но не пълно едно към едно документиране на функционалността. Много често ще има разлика защото програмната логика е строго обвързана с възможностите на програмния език. Така някой действия които изглеждат логически за бизнеса трябва да се пречупят през призмата на програмния език.

Документирането в Agile софтуера представлява твърдо описание на бизнес логиката така като е заложено за реализация на проекта. Бизнес логиката е това което изгражда едно приложение. Затова документиране на строги граници на бизнес логиката е съществено за реализацията на проекта. Ясното разписване в словом думи как ще работи една функционалност е първата стъпка към реализацията на тази функционалност. Тук се очертава как приложението ще се справя с проблеми и нестандартни сценарии.



Фиг. КЙХ: Екранна снимка показваща цялата документация под формата на бизнес логика в проекта.

Документирането в кода представлява процес на добавяне на коментари и обяснения, които описват функционалността, предназначението и поведението на различни части от кода. Това е ключов аспект от разработката, тъй като яснота в кода улеснява разбирането, поддръжката и разширяването му, особено при работа в екип или дългосрочни проекти.

Javadoc е специален инструмент за документиране в Java, който генерира автоматично HTML документация от коментарите в кода. Използвайки специфичен синтаксис, Javadoc позволява на разработчиците да предоставят подробно описание на класове, методи и полета. Подробната документация чрез Javadoc улеснява ориентирането в кода, като прави функционалностите лесни за разбиране и използване от други разработчици. Това също така помага при идентифицирането на зависимости и предотвратяването на грешки при работа с вече съществуващ код.

Design by Contract (Проектиране чрез договор) е методология, която дефинира ясни правила и условия за използването на класовете и методите в кода. Тя изисква от разработчика да зададе конкретни предварителни условия, следствени условия и инварианти, които трябва да бъдат спазени. Например, метод може да има изисквания за валидност на параметрите (предварителни условия) и очакван резултат (следствени условия). Този подход помага за предотвратяване на неочаквани грешки, като насърчава ясна структура и спазване на отговорностите на всеки компонент.

Съвместно използвани, доброто документиране и подходът Design by Contract повишават качеството на софтуера и подобряват работата в екип, като създават яснота, предотвратяват грешки и улесняват бъдещата поддръжка на кода.

### Среда за разработка

Подборът на правилната работна среда за разработка на Spring Boot микросервизи играе ключова роля за ефективността и качеството на проекта. Подходящата среда осигурява необходимите инструменти за разработка, тестване, разгръщане и мониторинг на микросервизите, като подпомага целия жизнен цикъл на софтуера. Тя включва избор на интегрирана среда за разработка (IDE), CI/CD инструменти, платформа за контейнеризация и оркестрация като Docker и Kubernetes, и средства за наблюдение и логване.

Правилно подбраната среда съкращава времето за разработка, подобрява комуникацията между отделните услуги и осигурява стабилност и мащабируемост на приложението. За Spring Boot микросервизи това е особено важно, тъй като тази архитектура разчита на независимостта и взаимодействието между компонентите. Среда с правилните инструменти помага за минимизиране на грешките, по-бързо откриване на проблеми и осигуряване на надеждност.

Освен това, подходящата работна среда улеснява прилагането на Agile методологията и DevOps практики, като поддържа бързи итерации, автоматизирани тестове и гладка интеграция на промените.

Затова е подходено с средата за разработка IntelliJ, който е продукт разработен от JetBrains. IntelliJ IDEA е подходящ избор за разработване на Spring Boot микросервизен проект, защото предлага мощни инструменти и интеграции, специално създадени за Java разработка и Spring екосистемата. Той осигурява интуитивна поддръжка за Spring Boot чрез автоматизирана конфигурация, препоръки и лесен достъп до зависимости, което значително ускорява работния процес. IntelliJ предлага вградени функции като автоматично допълване на кода, инспекции за грешки в реално време и навигация в проекти, които улесняват разработчиците при работа с многобройни микросервизи.

Предимства на IntelliJ IDEA:

- Отлична поддръжка за Spring Boot: IntelliJ разпознава Spring Boot конфигурациите и осигурява лесно управление на зависимости, бързи стартирания на приложението и автоматизирани съвети за оптимизация.

- Мощни инструменти за кодиране и рефакторинг: IntelliJ IDEA предлага интелигентно допълване на кода, автоматични корекции и инспекции, което намалява риска от грешки и улеснява навигацията в кода.

- Интеграция с Git и CI/CD инструменти: Поддръжката на Git е интегрирана, а също така платформата позволява лесно настройване на CI/CD автоматизация и работен поток с контейнеризация и оркестрация.

- Поддръжка на Docker и Kubernetes: IntelliJ включва вградена поддръжка за Docker и Kubernetes, което улеснява работата с контейнери и мащабирането на микросервизи.

Недостатъци на IntelliJ IDEA:

- Ресурсоемкост: IntelliJ IDEA е по-тежък инструмент, който изисква повече системни ресурси, което може да забави работата на по-стари или по-слаби компютри.

- По-висока цена: Пълната версия на IntelliJ е платена, което може да е значителен разход за малки екипи или индивидуални разработчици.

- Крива на учене: Заради множеството функции и инструменти, IntelliJ IDEA може да изисква повече време за адаптация, особено за нови потребители.

С използването на IntelliJ за разработка на Spring Boot микросервизи, проектът се възползва от бърза разработка, лесно управление на зависимости и стабилна интеграция с инструменти за DevOps, което прави средата ефективна и продуктивна. В сравнение с конкурентите си, като Eclipse и Visual Studio Code (VS Code), IntelliJ IDEA се отличава със специфичните си функции за Java и Spring Boot. Изборът между тези инструменти зависи от нуждите на проекта: IntelliJ е идеален за задълбочена Java/Spring Boot разработка, докато VS Code и Eclipse предлагат добри алтернативи за по-леки и разнообразни среди.

### Контролер

Контролера представлява връзката помежду вътрешната modbus мрежа на клиента и централизираната сървърна система.

#### Хардуер

Хардуера предвиден за прототип е микро контролер способен да подържа Linux базирана операционна система, да има сериен порт от тип usb, да има Ethernet порт и по възможност да има html out порт. Тези характеристики биват покрити от множествено микро контролери. Проучени микро контролери подходя за целта са:

Orange Pi Zero2:

Orange Pi Zero2 е малък едноплатков компютър, създаден от компанията Shenzhen Xunlong Software CO., Limited. Този компактен и достъпен компютър е предназначен за различни DIY проекти, разработка на вградени системи и IoT (Интернет на нещата) приложения.

**Основни характеристики на Orange Pi Zero2 включват:**

**Процесор и производителност**: Orange Pi Zero2 е оборудван с Allwinner H616 SoC, който включва четириядрен ARM Cortex-A53 процесор. Този процесор предлага добра производителност за широк кръг от приложения.

**Графика**: Вградената графика е Mali G31 MP2 GPU, която поддържа 3D графика и видео ускорение, позволявайки възпроизвеждане на висококачествено видео и графични приложения.

**Памет**: Устройството разполага с 512MB или 1GB DDR3 RAM, в зависимост от конкретния модел, което осигурява достатъчно памет за повечето леки до средно тежки задачи.

**Съхранение и разширение**: Orange Pi Zero2 разполага с microSD слот за разширяемо съхранение, което позволява инсталирането на операционни системи и приложения.

**Свързаност**:

**Wi-Fi и Bluetooth**: Вградена поддръжка за 2.4GHz Wi-Fi и Bluetooth 4.2, което улеснява безжичната свързаност и комуникация. **Ethernet**: 100Mbps Ethernet порт за стабилна кабелна мрежова връзка.

**Портове и интерфейси**:

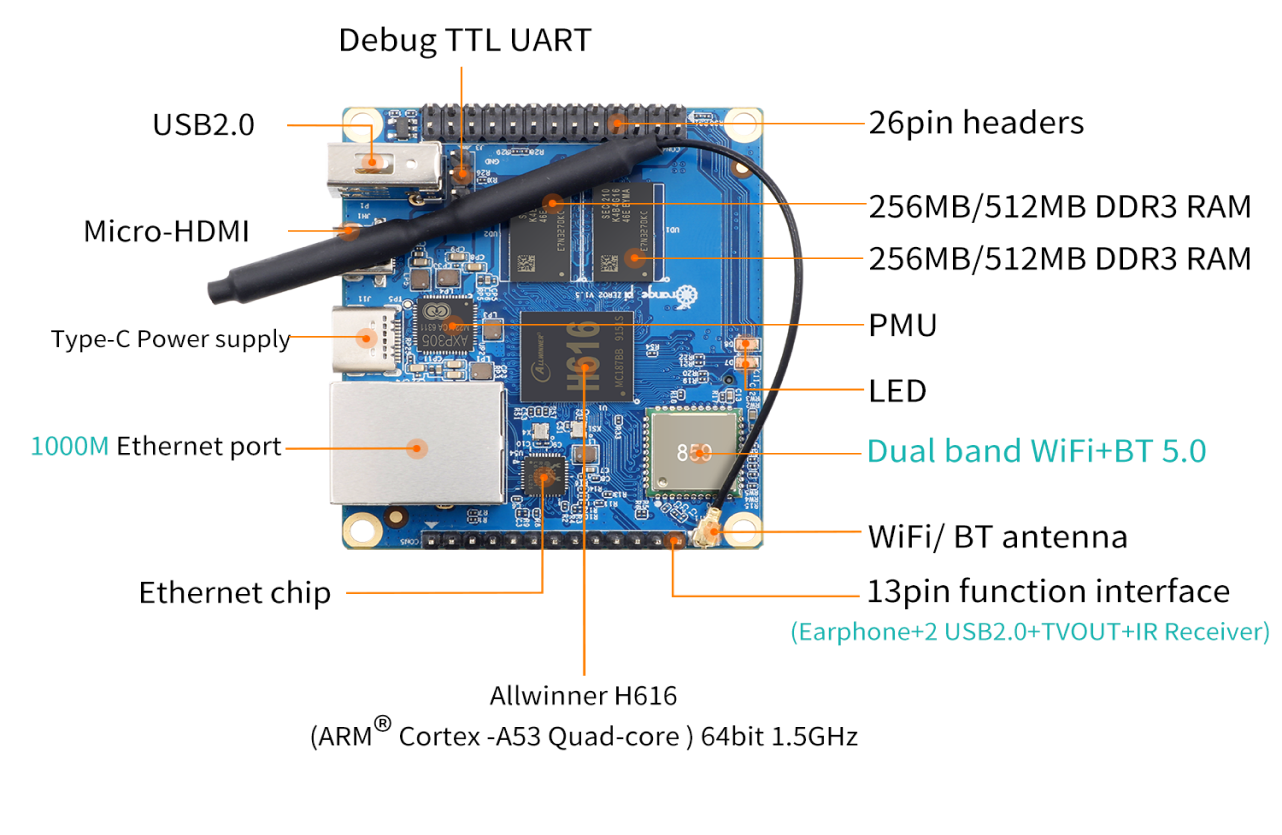
**HDMI**: Пълноразмерен HDMI порт за свързване към дисплеи и телевизори.

**USB**: Различни USB портове за свързване на периферни устройства като клавиатури, мишки и други.

**GPIO**: General Purpose Input/Output пинове за свързване на сензори и други компоненти, което го прави идеален за хоби и професионални електронни проекти.

**Операционни системи**: Поддържа различни операционни системи като Android, Ubuntu и Debian, предоставяйки гъвкавост и лесна интеграция в различни среди.

Със своите компактни размери и мощни функции, Orange Pi Zero2 е подходящ както за начинаещи, така и за напреднали потребители, търсещи надеждно и ефективно решение за своите проекти и приложения.



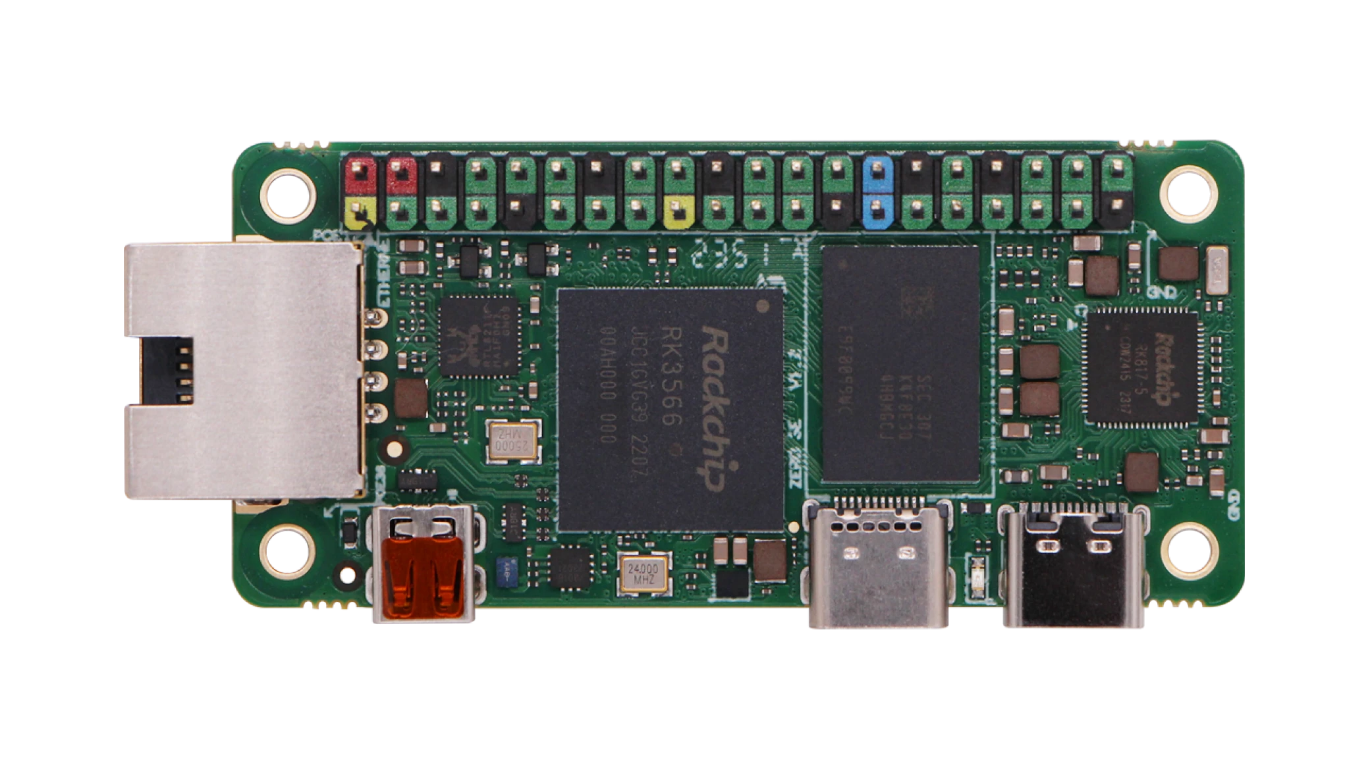
Фиг. Изображение показващо Техническите характеристики на Orange Pi Zero2 както и всички свободни пинове и портове.

Radxa zero 3e:

Radxa Zero 3e е миниатюрен компютър, проектиран от компанията Radxa. Той е част от серията Radxa Zero и е известен със своята компактност и мощност, съчетавайки малки размери с отлични технически характеристики. Устройството е оборудвано с мощен процесор и достатъчно оперативна памет, което го прави подходящо за различни приложения, включително ембедед системи, Интернет на нещата (IoT) проекти и мултимедийни задачи.

Radxa Zero 3e разполага с множество интерфейси и портове, които позволяват свързване към различни периферни устройства и модули. Това включва HDMI изход за видео, USB портове за свързване на периферия и microSD слот за разширяване на паметта. Устройството поддържа различни операционни системи, което го прави гъвкаво и лесно за интегриране в различни проекти.

Със своята комбинация от мощност, компактност и гъвкавост, Radxa Zero 3e е привлекателен избор за разработчици и ентусиасти, които търсят ефективно решение за своите технологични нужди.



Фиг. Изображение представящо Radxa zero 3e

ROCK Pi S

ROCK Pi S е малък, но мощен едноплатков компютър, който използвам за различни проекти. Този миниатюрен компютър е проектиран от Radxa и е идеален за IoT (Интернет на нещата) приложения, както и за вградени системи.

Ето основните му характеристики, които ме впечатляват:

**Процесор и производителност**: ROCK Pi S разполага с Rockchip RK3308 четириядрен ARM Cortex-A35 процесор, който осигурява достатъчно мощност за различни задачи, въпреки малките си размери.

**Памет и съхранение**: Има опции с 256MB или 512MB RAM, което е достатъчно за леки приложения и IoT задачи. За съхранение използва microSD карта и може да поддържа и NAND флаш памет.

**Свързаност**:

**Wi-Fi и Bluetooth**: Вградените 2.4GHz Wi-Fi и Bluetooth 4.2 ме улесняват в безжичната комуникация и свързване с други устройства.

**Ethernet**: 10/100Mbps Ethernet порт предоставя надеждна кабелна мрежова връзка.

**Портове и интерфейси**:

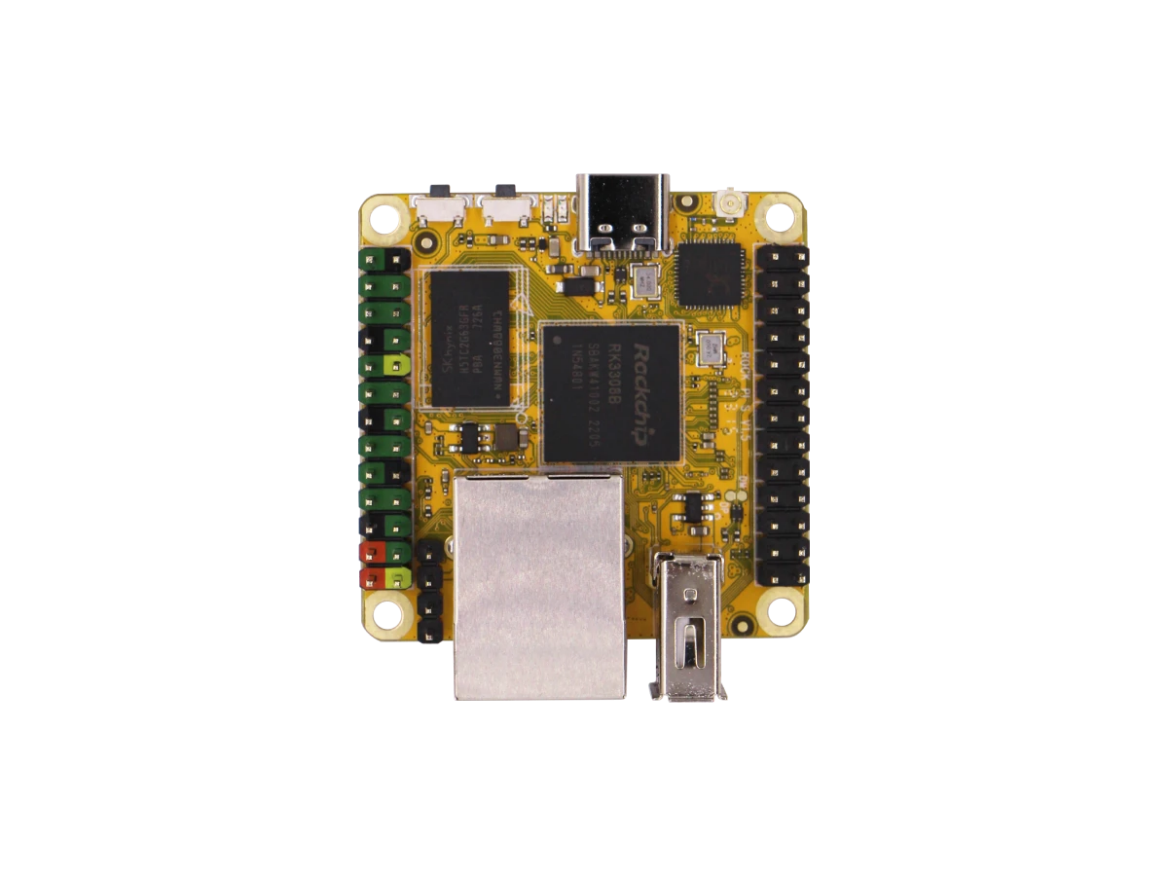
**USB**: Има няколко USB порта, които използвам за свързване на периферни устройства.

**GPIO**: General Purpose Input/Output пиновете ми позволяват да свързвам различни сензори и модули, което го прави идеален за проекти с вградени системи.

**Размери и консумация на енергия**: ROCK Pi S е изключително компактен и енергийно ефективен, което го прави подходящ за проекти, където пространството и консумацията на енергия са критични.

**Операционни системи**: Поддържа различни операционни системи като Debian и Ubuntu, което ми дава гъвкавост при разработването и интегрирането на различни приложения.

Използвайки ROCK Pi S, успявам да реализирам различни идеи и проекти, независимо дали става въпрос за домашна автоматизация, IoT устройства или вградени системи. Този малък компютър е мощен инструмент в моите ръце.



Фиг. Изображение на едноплатковия компютър ROCK Pi S

За целта на проекта е решено да се вземе модела Orange pi zero 2

### Контрол на версиите

Контролът на версиите е система, която позволява проследяване на промените в даден проект с течение на времето. Той записва всяка модификация, направена върху файловете, като съхранява отделни версии на проекта. Това дава възможност на разработчика да се връща към предишни версии, да сравнява различни ревизии и да разбира кой е направил определени промени и защо. При работа в екип контролът на версиите позволява множество хора да работят върху един и същ проект едновременно, като предотвратява конфликтите, свързани с редактирането на един и същ код или файл.

#### Технология за контрол на версиите

Git е разпределена система за контрол на версиите, която позволява на разработчиците да проследяват промените в техните проекти и да работят заедно по код. Тя съхранява пълна история на всяко изменение, като позволява лесно връщане към предишни версии на проекта. Git работи локално на компютъра на всеки потребител, което означава, че всеки разработчик има пълно копие на целия проект и неговата история.

Една от основните характеристики на Git е възможността за създаване на клони, което дава на разработчиците свободата да експериментират и работят върху нови функции, без да променят основния код. След като тези промени бъдат завършени и тествани, те могат да бъдат обединени с основния код. Git също така е много ефективен при управление на конфликтите, които възникват, когато няколко души работят едновременно върху един и същ проект.

Git се използва в днешно време, защото предоставя ефективен начин за управление на проекти и сътрудничество между разработчиците. Той позволява на екипите да проследяват промените в кода, да съхраняват историята на проекта и да работят паралелно без конфликти. Разработчиците ценят Git за неговата гъвкавост, тъй като могат да създават клони и да експериментират с нови функции, преди да ги обединят с основния код.

Git също така улеснява работата с отворен код, като позволява на множество хора да допринасят за един проект, независимо от тяхното местоположение. Системата за контрол на версиите осигурява стабилност и безопасност, тъй като позволява лесно възстановяване на предишни версии в случай на грешки. В допълнение, Git интегрира много инструменти и платформи, което го прави важен компонент в съвременния софтуерен процес на разработка.

#### Конкуренцията за хостинг на Git

Конкуренцията за хостинг на Git включва платформи като GitLab, Bitbucket и други подобни услуги, които предоставят хранилище за код и инструменти за управление на проекти. Всяка от тези платформи предлага сходни функции, като контрол на версиите, интеграция с CI/CD и възможности за сътрудничество. GitLab, например, се използва заради своите мощни DevOps инструменти и опции за самостоятелен хостинг, докато Bitbucket се интегрира добре с други продукти на Atlassian, като Jira.

GitHub обаче се предпочита заради широкото му разпространение и голямата потребителска база. Въпреки че конкурентите предлагат специфични функции, GitHub остава водеща платформа за хостинг на Git проекти поради своята популярност, лесен интерфейс и силна интеграция с екосистемата на разработчиците.

#### Платформа за хостинг на git

GitHub е онлайн платформа за хостване на софтуерни проекти и сътрудничество, базирана на системата за контрол на версиите Git. Тя позволява на разработчиците да качват своя код, да го споделят с други, както и да работят заедно по проекти, независимо от тяхното местоположение. GitHub предоставя функции като управление на версиите, проследяване на проблеми (issues), преглед на кода (code review), и създаване на клонинги (forks) на проекти. Той също така улеснява процеса на интеграция и доставка на софтуер (CI/CD), като е широко използван както от индивидуални разработчици, така и от компании по целия свят. В главата за автоматизация е разгледано как е приложени GitHub actions за проекта.

GitHub също така предлага възможност за създаване на хранилища (repositories), където кодът се съхранява и организира. Всяко хранилище съдържа пълна история на промените, направени върху проекта, което позволява на разработчиците да проследяват всяка версия и лесно да се връщат към предишни състояния на кода при нужда. Платформата поддържа концепцията за "pull requests", които позволяват на потребителите да предложат промени в кода на даден проект, а екипът на проекта може да прегледа и обсъди тези промени преди тяхното одобрение.

GitHub също така има социални аспекти – разработчиците могат да следват други потребители, да споделят проекти публично и да откриват нови проекти чрез търсене на хранилища или потребители с подобни интереси. Платформата е интегрирана с множество инструменти за разработка и автоматизация, което я прави мощен инструмент за управление на проекти и колаборация в света на софтуерното инженерство.

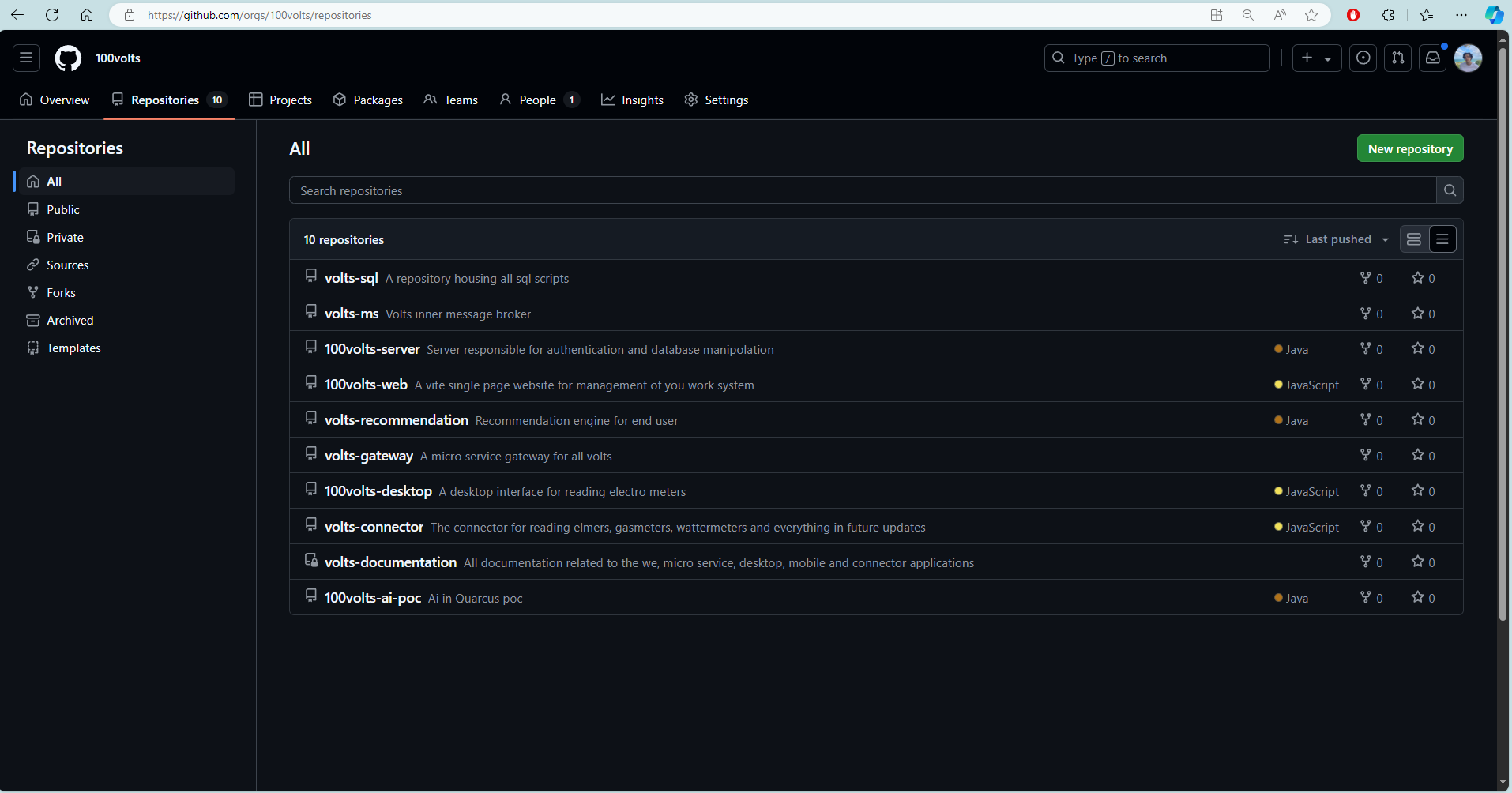
#### Организация на репозиториите

Репозиторията представлява хранилище за код и свързаните с него файлове в рамките на проект. Тя съхранява цялата история на промените, направени върху проекта, включително различни версии, добавяния и редакции. Разработчикът използва репозиторията, за да организира файловете и да проследява работата си, като лесно може да преглежда предишни версии и да създава нови клонове за експерименти. Репозиторията позволява съвместна работа с други разработчици, като предоставя начин за споделяне и съхраняване на проекта на едно централно място, обикновено на платформа като GitHub.

GitHub организацията представлява група от потребители, която споделя и управлява проекти на едно централно място. Тя позволява на различни членове да работят заедно, като предоставя контрол върху достъпа и разрешенията за различни хранилища. Чрез организацията могат да се създават и управляват екипи с различни нива на права върху проектите, което улеснява координацията и сътрудничеството в екипа. Организацията също така осигурява възможност за управление на множество проекти едновременно, като централизира ресурсите и улеснява работния процес. За целите на проекта е работено от един човек но много рядко един програмен продукт се разработва само от един човек. Затова е избран стандарта за организиране на рапозитории чрез организация.

GitHub организацията се конкурира с платформи като GitLab и Bitbucket, които също предлагат хостинг на код и инструменти за сътрудничество. Въпреки че тези конкуренти предоставят сходни функции като управление на хранилища и интеграция с CI/CD, GitHub организацията се избира заради широката си популярност и голямата общност от разработчици. Тя има добре изградена екосистема и предлага отлична интеграция с инструменти като GitHub Actions за автоматизация.

GitHub организацията също така се предпочита заради лесния си за използване интерфейс и стабилната си поддръжка на проекти с отворен код, което привлича много разработчици и фирми. Благодарение на силната си позиция в индустрията, GitHub често се възприема като стандартен избор за екипно сътрудничество и управление на проекти.



Фиг JJD. Всички репозитории в организацията на проекта.

За организацията на репозиториите е подходено по имплементация на микро сервиз. Тоест когато има нов микро сервиз или част от системата, се създава нова репозитория. Така кодът за една услуга остава изолиран в отделен проект, както е по модела на разпределените системи. Енкапсулирайки всеки сервиз, това позволява индивидуална настройка на репозиторията спрямо имплементацията. Така и за всяка репозитория се отделя и автоматизацията.

С този подход, всеки сервиз може да има своя специфична интеграция с инструменти за CI/CD, което позволява по-гъвкаво и независимо управление на процесите по разработка и доставка. Когато възникнат промени или подобрения в един микро сервиз, те не засягат директно останалите части на системата. Това улеснява мащабирането и поддръжката на системата, като всяка репозитория може да се оптимизира спрямо нуждите на съответния сервиз, без да се компрометира цялостното решение.

Когато има различни репозитории, те могат да се организират спрямо кодовите практики на съответния език, като се следват най-добрите стандарти за всяка технология. В зависимост от езика, разработчикът настройва репозиторията така, че да поддържа специфичната структура на проекта и инструментите, които са характерни за този език.

Например, за Java се използва структурата на директориите с разделение на папките src/main/java и src/test/java за основния код и тестовете. Интеграцията с инструменти като Maven или Gradle за управление на зависимости и изграждане на проекта е стандартна практика. Всяка Java репозитория може да има специфични правила за форматиране на кода с помощта на Checkstyle или PMD.

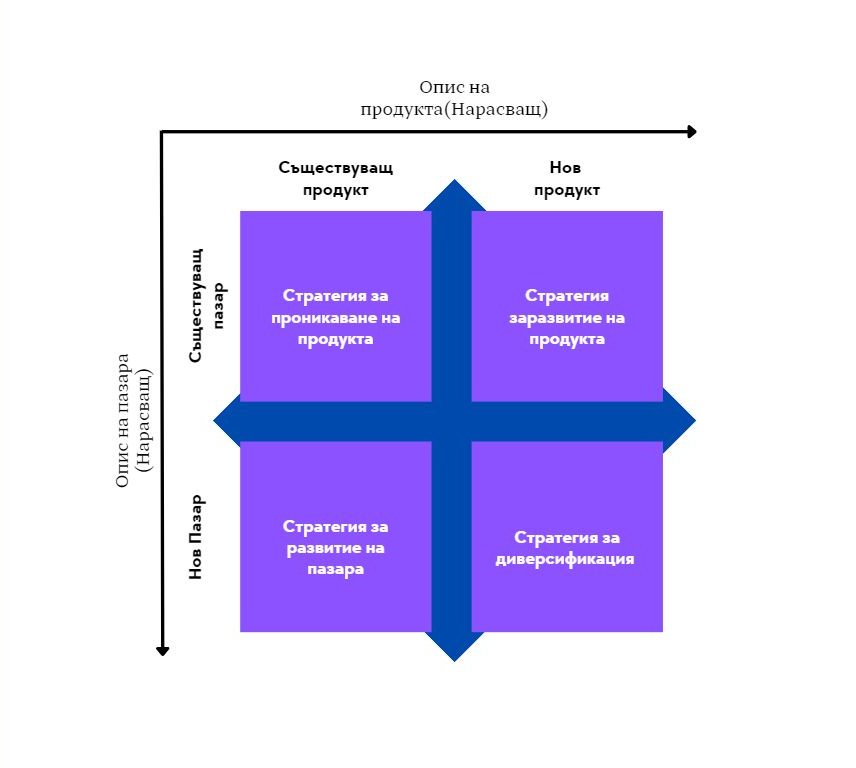
За JavaScript, репозиторията обикновено включва конфигурационни файлове като package.json за управление на зависимости чрез NPM или Yarn. Също така, ESLint се използва за поддържане на кодов стил и качество. Проектите могат да бъдат структурирани в зависимост от рамките, като например използване на директории src и dist при работа с React или Node.js.

При Rust, стандартната организация на репозиторията включва използването на Cargo.toml за управление на зависимостите и настройките на проекта. Структурата на папките включва директориите src за основния код и tests за тестовете. Кодовите практики се спазват с помощта на инструменти като rustfmt и clippy за формат и статичен анализ.

Всеки език предлага различни инструменти и подходи за автоматизация и контрол на качеството, които могат да се интегрират в съответните репозитории, за да отговарят на стандартите на екипа и проекта.

## Модел на Ансов приложен за системата.

Когато се разполага с информация конкретна за фирмите конкуренти, може да се изгради как желаният продукт изглежда на пазара, и той до колко е конкурентно способен. При проучване на конкурентно способността може да се приложи модела на Ансов. Прилагайки и употребявайки неговата матрица. Матрицата на Ансов представлява стратегията на фирмата спрямо в кой пазар се намира (Фиг. ГГС) и какъв продукт предлага. Разглеждат се следните точки:



Фиг. ГГС. Матрица на Ансов.

1. **Стратегия за проникване на продукта**

При стратегията за проникване на продукта, се следи как в съществуват пазар и съществуващ продукт и подхода на фирмата по дадените точки:

* Приход от закупува:

Предвижда се 120$ на месец за всяка инстанция или отдел на фирмата купувач, т.е. ако купувача има няколко обекта заплаща лиценз за всеки един от тях. В тази цена включва и поддръжка по емайл, както и поддръжка по телефон.

* Група на продукта:

Групата на продукта е пазара отговорен за мониторинг на продукцията и разходите. Така като цяло попада в по-широк пазар. Това означава че се конкурира продукта. Това е така, защото пазара за следене на производството и разходите по него е широк. Но малко приложения имат директно четене на разходомери на енергия.

* Отклонения:

Отклоняващата чарта на приложението е че пряко свързан мониторинг с разходомерите и производството. Тоест системата веднъж настроена не се нуждае от постоянна поддръжка. Единствено системата за правилно изчисление на разходите по производството се нуждае от впиване на енергийни индекси и въвеждане на не автоматизирано производство. И под не автоматизирано производство се има предвид производство не свързано с брояч.

* Новост:

Новостите на системата за мониторинг спрямо конкурентите е модерен уеб интерфейс за промени и следене на системата. Множество от конкурентите не се концентрират върху самата комуникация помежду клиент система. Друга новост спрямо конкуренцията е едновременното следене на всички видове енергетика. Повечето системи се концентрират върху един вид разход и главно той е електричеството.

* Процент 20-80:

В тази стратегия се разглежда дали 80% от приходите идват от 20% от клиентите. Тази стратегия към системата за менторинг ще се отнесе към това да привлечем клиент и да остане вечен клиент. Това са предприятия където се нуждаят само и единствено от една имплементация за следене на разходите към производството и не искат да се променят. Факт е че когато едно предприятие свикне с даден софтуер е че те не приемат алтернативи. главно поради нужните преизчисления и промени към вътрешната им структура на работа. Както системата спрямо избраната архитектура е заложено да запазва всичко на сървъра. Тоест клиента ако пожелае да си смени приложението с това на конкурент той няма да може да прехвърли вече съхранената информация.

* Икономия от мащаба:

Системата подържа икономия на мащаба поради архитектурата на системата. Имайки микро сервизи с различно натоварване може да се пускат и спират самите инстанции на контейнерите. Това води до голямо намаляване на разходите срещу това да имаш един колосален сървър, който да е готов за максимален товар постоянно. Тоест системата когато не е натоварена се подържа само автоматичното изчисляване на разходите и при голям товар се стартират допълнителни инстанции на изчисляващите сервизи. Както и архаизацията на данни по стари от дванадесет месеца във вторична база за архаизация намалява товара на главната система.

1. **Стратегия за развитие на пазара**

Когато се разглежда стратегията за развитие на пазара се взема в предвид, че приложим съществуващ продукт на нов пазар. За правилно прилагане на подобна стратегия, се разглеждат следните точки:

* Как може да се потвърди на пазара:

Софтуера като услуга може да се потвърди на пазара след като е тестван и потвърден от тестовия клиент. Както и продукта може да се потвърди на пазара не само с добри рецензии от тестови клиент но и от богатите си функции и статистики. Друг фактор който играе силна роля за потвърждаване на пазара е локализацията. Един клиент е по-настроен да използва един продукт ако той е локализиран на неговия език.

* Темпото на новия пазар:

Когато се стъпва на нов пазар продукта има за задача да навлезем с темпото на новия пазар, т.е. ако е утвърдено в един пазар, като на пример англоговорещия свят, и реши да се премине към хинди говорещ пазар то има изградена инерция да настъпи към по-малък но се така конкурентен пазар.

* Брой участници:

Броя участници за дадения продукт е ключов, защото при установили се продукти те трудно се сменят. Поради естеството на система, която гласи че се ползва безвъзвратно и продължително от клиента. Самата идея за смяна на системата, възниква или от неефективноста и/или от нарастващ поток от клиенти.

* Входни бариери:

Софтуера сам по себе си няма много бариери за преминаване със съществуващият продукт на нов пазар. Софтуера се закупува и се употребява. Естествено ще има първа инсталация и настройка на системата. След което се очаква да се до настройва докато не придобие очакваните характеристики.

* Уникалното:

Уникалността като стратегия за продукта на нов пазар, е достатъчно само да се спомене нейната гъвкавост на прилагане. Друг фактор за уникалност на нов пазар е и цента. За бизнес в Германия месечния абонамент ще се струва ниска и би могло да е фактор за приемането на системата.

1. **Стратегия за развитие на продукта**

Стратегията за развитие на продукта може да се разгледа като се реши да се надградим над избора на предефинирани шаблонно, които допълват и улесняват конфигурацията на системата. Това е само една от стратегиите, който могат да се предприемат когато има нов продукт на същия пазар. Друга подобна стратегия да се вкарат конкретни демографии към рекламата на продукта, т.е. да се преименуват готови шаблони като развити крайни продукти. Те мога да включват: пекарна, промишлен завод, млекопреработвател и други. Но за цялостна оценка по матрицата на Ансов трябва да се разгледат следните опорни точки за стратегията за същия пазар с нов продукт:

* Теми на пазара:

Темите на пазара влиае на това как се представя продукта като цяло. Ако един продукт се представи в среда на производители и преработватели то продукта ще се приеме за тях. Общо казано приставянето на продукта зависи от там където го приставиш. Системата би се представила пред пивоварна по един начин и пред пекарна по друг.

* Отразяване на конкуренцията:

При отразяването на конкуренцията трябва да се вземе в впрочем процента заетост на конкуренцията в дела, в който желае да се подобри продукта. За продукта може да се каже че е не конкурентно способен спрямо строго специализирани програмни продукти, като на пример фирмата „frodexim“, които целят да ти продадат техен продукт, като електромер или друг измервателен апарат, с който можеш да наблюдаваш разходите по производство или цялостната дейност на предприятието.

* Ниво на границите за пробиване:

Нова граница за пробиване може да се разгледа централизиран опит за проникване на продукта скоростно или да се подходи пасивно, т.е. продукта да се наложи на пазара като заместител на по-скъпа и по-специализиран продукт.

* Иновативност:

Иновативност разглежда дали системата е достатъчно отделена от конкуренцията на сегашния пазар. Продукта с неговата гъвкавост е достатъчно отличим от конкуренцията.

* Раздялата:

С разделността на пазара може да се разгледа дали системата попада в такава демография където за момента има спад на пазара или цялостна липса на продукт.

* Ниво на посоката:

Разглежда се ниво на посоката като стандартите, които са нужни за да е конкурентно способна системата. Пример: Дали системата подържа персонализиране на статистики или дали системата подържа CSV[[3]](#footnote-3) експортиране на данни.

1. **Стратегия за диверсификация**

Разглеждането на диверсификацията на системата се подхваща от страна на новият пазар и новият или подобреният продукт.

* Темпо на растене:

Темпото на растене се разглежда от страна на конкурентността и от страна на продукта, т.е. водейки до един среден отчет дали решението за диверсификация е възможно или влиза на сферата на пасивите. Темп на растене не може да се оцени поради етапа на проекта до сегашния му период. Но е нужна оценка когато продукта се развие.

* Наличие на собствена компенсация:

Под наличие на собствена компетенция се разглежда адекватността за диверсификацията, т.е. може ли даденият отрасъл да не е подходящ за приложение на системата. Спрямо архитектурата на системата тя притежава висока собствена компенсация. Това е така защото по дизайн система е изградена да може да се надгражда и усложнява когато времето настъпи за добавяне на логика към нея.

* Ниво на компетенцията:

Нивото на компанията е мярката за отклонение спрямо сходните продукти на пазара. Тоест дали системата е на едно ниви с конкуренцията или по-високо. Системата само по-себе си не е тествана върху широката гама от измерващи уреди на пазара. Тоест системата е по-ниска компетенцията от конкуренцията.

* Възможности на риска:

Възможности на риска е калкулацията на възможността за пробиване на продукта на нов пазар.

Пример: Ако не се успее да се конфигурира системата за нов клиент може ли да се да се платят неустойките.



Фиг. 5. Реализацията на матрицата за конкретна ситуация

# Проектиране

## 1. Разработка на база от данни.

Разработка на базата от данни започва със създаване първо на таблиците, които са от първа необходимост. Първите таблици са тези, които се явяват ядрото на цялата система. Създаването на базата от данни е първата и най-важна част от процеса на създаване на система, защото именно тя служи като основа за съхранение и управление на цялата информация, която системата ще използва. В този етап се определят структурата, връзките между данните и начинът, по който те ще бъдат достъпвани и обработвани. Това е критично, защото грешки или лош дизайн на базата от данни могат да доведат до проблеми с ефективността, сигурността и разширяемостта на цялата система. Затова първоначалното й създаване изисква внимателно планиране и анализ.

Създаването на база от данни от нулата може да се осъществи чрез два основни подхода: ръчно създаване и използване на автоматизирани инструменти за генериране на база от данни.

1. Ръчно създаване на база от данни

При този подход разработчикът използва език за управление на бази от данни (SQL[[4]](#footnote-4)), за да дефинира таблици, полета, ключове и връзки. Той самостоятелно планира и изпълнява дизайна на базата.

Позитивни черти:

- Осигурява висока степен на контрол върху структурата на данните.

- Позволява персонализирани решения според нуждите на проекта.

- Подходящ е за сложни проекти, които изискват специфични конфигурации.

\*\*Негативни черти\*\*:

- Отнема много време и усилия.

- Изисква висока квалификация и задълбочени знания за базите от данни.

- Лесно се допускат грешки при сложни схеми.

2. Използване на автоматизирани инструменти

Този метод използва софтуерни инструменти като ERD[[5]](#footnote-5) генератори или ORM[[6]](#footnote-6) библиотеки (напр. Entity[[7]](#footnote-7) Framework или Hibernate), за да създаде база от данни на базата на предварително зададени модели или визуални диаграми.

Позитивни черти:

- Спестява време и опростява процеса.

- Минимизира риска от грешки, като автоматично генерира код за базата.

- Лесно се поддържа и актуализира при промени в изискванията.

Негативни черти:

- Ограничен контрол върху детайлите на дизайна.

- Генерираната структура може да бъде неоптимална за специфични случаи.

- Зависимостта от инструмента може да създаде проблеми при интеграция с други системи.

Spring boot, който е избран за реализация на проекта, поддържа тази функционалност но за целите на проекта не е издран за ползване. Но е използвано ORM за персистиране на данни в формата на обекти. Но е изключена фунцианлноста за генериране на таблици спрямо Entity objects.

Изборът на подход зависи от сложността на проекта, наличните ресурси и времевите ограничения.

# Заключение

# Архитектура

# Автоматизация

Автоматизацията представлява процесът на използване на технологии за извършване на задачи с минимална човешка намеса. В съвременния контекст, автоматизацията обикновено се отнася до софтуерни и хардуерни системи, които автоматизират рутинни и повторяеми дейности, подобрявайки ефективността, точността и скоростта на изпълнение.

### Основни цели на автоматизацията

**Повишаване на ефективността**: Автоматизацията елиминира нуждата от ръчно изпълнение на задачи, което значително намалява времето за изпълнение на процеси и увеличава производителността.

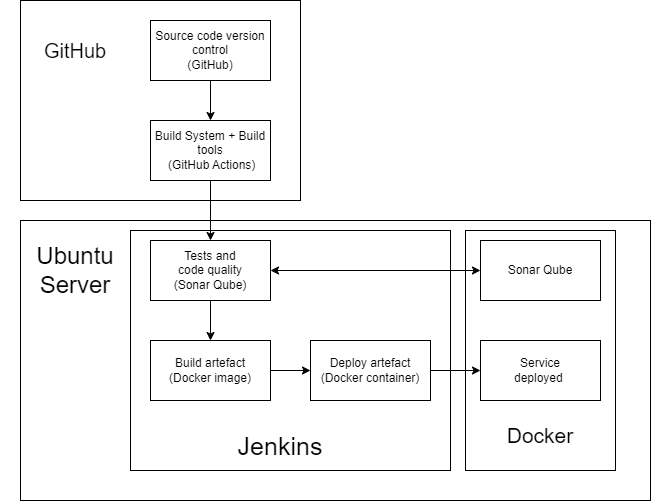
**Подобряване на качеството и точността**: Човешките грешки често водят до несъответствия и проблеми в качеството. Автоматизираните системи следват предварително дефинирани инструкции, което гарантира висока степен на точност и консистентност.

**Намаляване на разходите**: Чрез автоматизация се намалява необходимостта от ръчен труд и свързаните с него разходи. Това включва както преките разходи за заплати, така и непреките разходи, свързани с грешки и последващи корекции.

**Осигуряване на мащабируемост**: Автоматизираните системи могат лесно да се мащабират според нуждите на бизнеса, без да се изискват значителни допълнителни ресурси.

Автоматизацията играе ключова роля в съвременния свят, като трансформира начините, по които се извършват бизнес процеси и технологични операции. Тя осигурява значителни предимства по отношение на ефективност, качество, разходи и мащабируемост. Внедряването на автоматизирани системи продължава да бъде стратегически приоритет за много организации, които търсят начини да подобрят своята конкурентоспособност и да отговорят на нарастващите изисквания на пазара.

Pipeline в софтуерното инженерство представлява последователност от автоматизирани процеси, които се изпълняват с цел разработка, тестване и внедряване на софтуерни приложения. Тези процеси обикновено включват етапи като компилация, тестване, изграждане на артефакти и разгръщане на софтуера в производствена среда.



Фиг. Диаграма представяща цялостния работен поток за всеки Spring application в проекта.

#### Основни компоненти на Pipeline

**Източник на код (Source Code)**. Този етап включва изтегляне на последния изходен код от системата за контрол на версиите, като например GitHub, GitLab или Bitbucket.

**Компилация (Build).** На този етап изходният код се компилира, за да се провери дали няма грешки в кода и дали той може да се превърне в изпълним артефакт. Това включва и инсталирането на зависимостите, необходими за проекта.

**Тестване (Test)**. Тестовете се изпълняват автоматично, за да се гарантира, че новите промени не водят до регресия или грешки. Тестовете могат да включват unit тестове, интеграционни тестове, функционални тестове и др.

**Изграждане на артефакти (Artifact Creation)**. След успешната компилация и тестване, се създават артефакти, като например JAR файлове, Docker образи или други изпълними файлове, които ще бъдат разгръщани.

**Разгръщане (Deployment)**. Артефактите се разгръщат в различни среди, като тестови, staging или производствени среди. Този етап може да включва и изпълнението на миграции на бази данни, конфигурационни промени и др.

**Мониторинг и обратна връзка (Monitoring and Feedback)**. След разгръщането, системата се наблюдава за потенциални проблеми или аномалии. Обратната връзка от мониторинг системите може да бъде използвана за подобряване на следващите итерации на pipeline-а.

#### Приложение на Pipeline

Pipeline-ите се използват широко в практиката на DevOps и непрекъсната интеграция и разгръщане (CI/CD) за автоматизация на жизнения цикъл на софтуерната разработка. Те позволяват на екипите да постигнат бързо и надеждно разгръщане на нови версии на софтуера с минимална човешка намеса.

**Непрекъсната интеграция (CI)**. Pipeline-ите автоматизират интеграцията на кода от различни разработчици, като гарантират, че всяка нова промяна се проверява автоматично и не нарушава работата на съществуващия код.

**Непрекъснато разгръщане (CD)**. Pipeline-ите автоматизират разгръщането на нови версии на софтуера в различни среди. Това позволява бързо и често публикуване на нови функционалности и поправки на грешки.

**Подобряване на качеството на софтуера**. Чрез автоматизирани тестове и проверки, pipeline-ите гарантират високо качество на софтуера, като откриват проблеми рано в процеса на разработка.

Pipeline-ите са неотменима част от съвременната софтуерна разработка, която позволява на екипите да автоматизират и оптимизират процесите по изграждане, тестване и разгръщане на софтуера. Те осигуряват надеждност, ефективност и мащабируемост, като същевременно спомагат за намаляване на човешките грешки и подобряване на качеството на софтуерните продукти.

## GitHub Actions

Първата стъпка при автоматизацията започва от репозиторията съдържайки програминия код. Без код няма приложение. Доставчика на хостинг на репоситория избран е GitHub. GitHub е устойчив доставчик отличил се е на пазара от дълги години. 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).

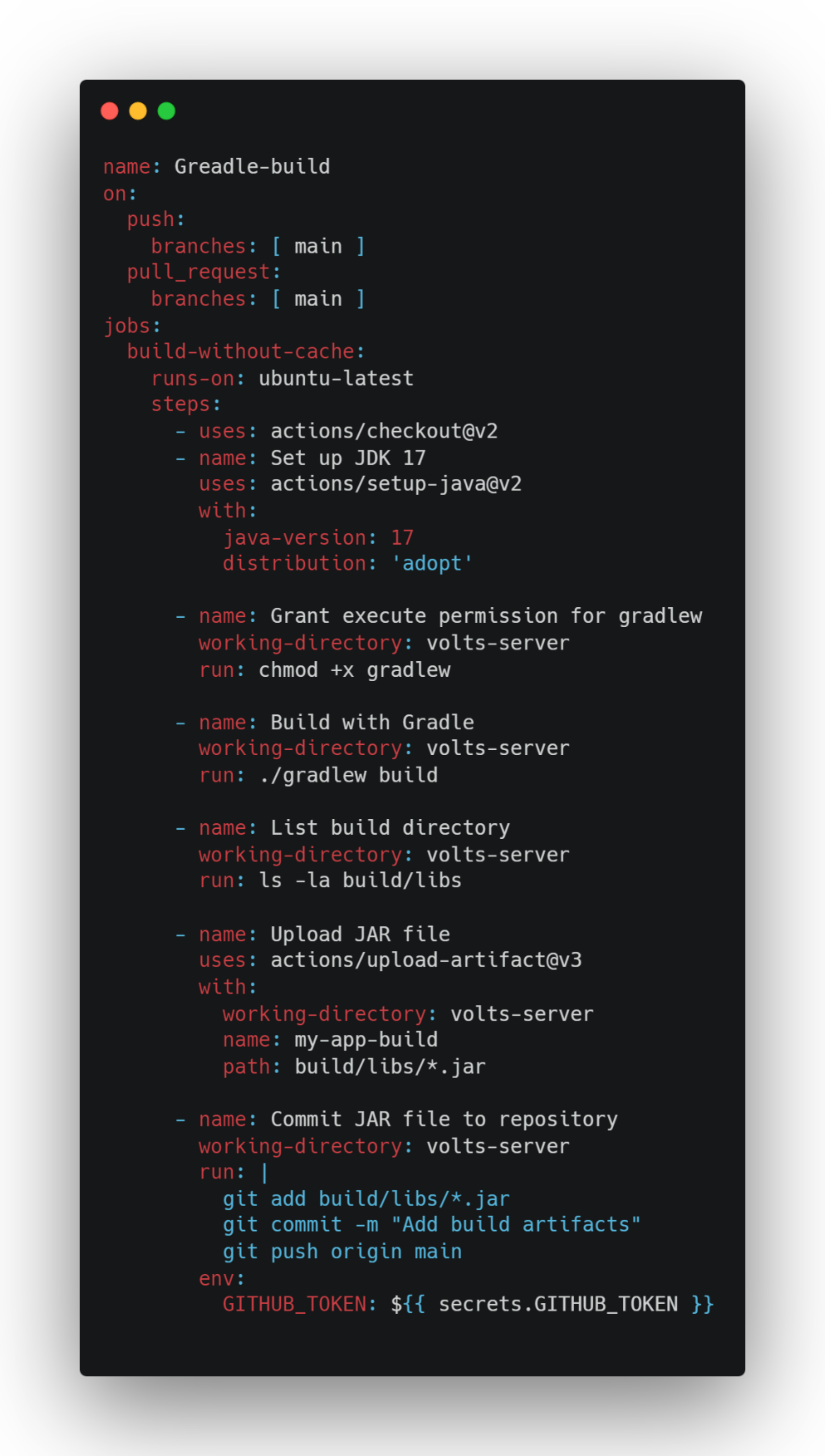
### Основни характеристики на GitHub Actions

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.

### Работен поток използван в проекта

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:

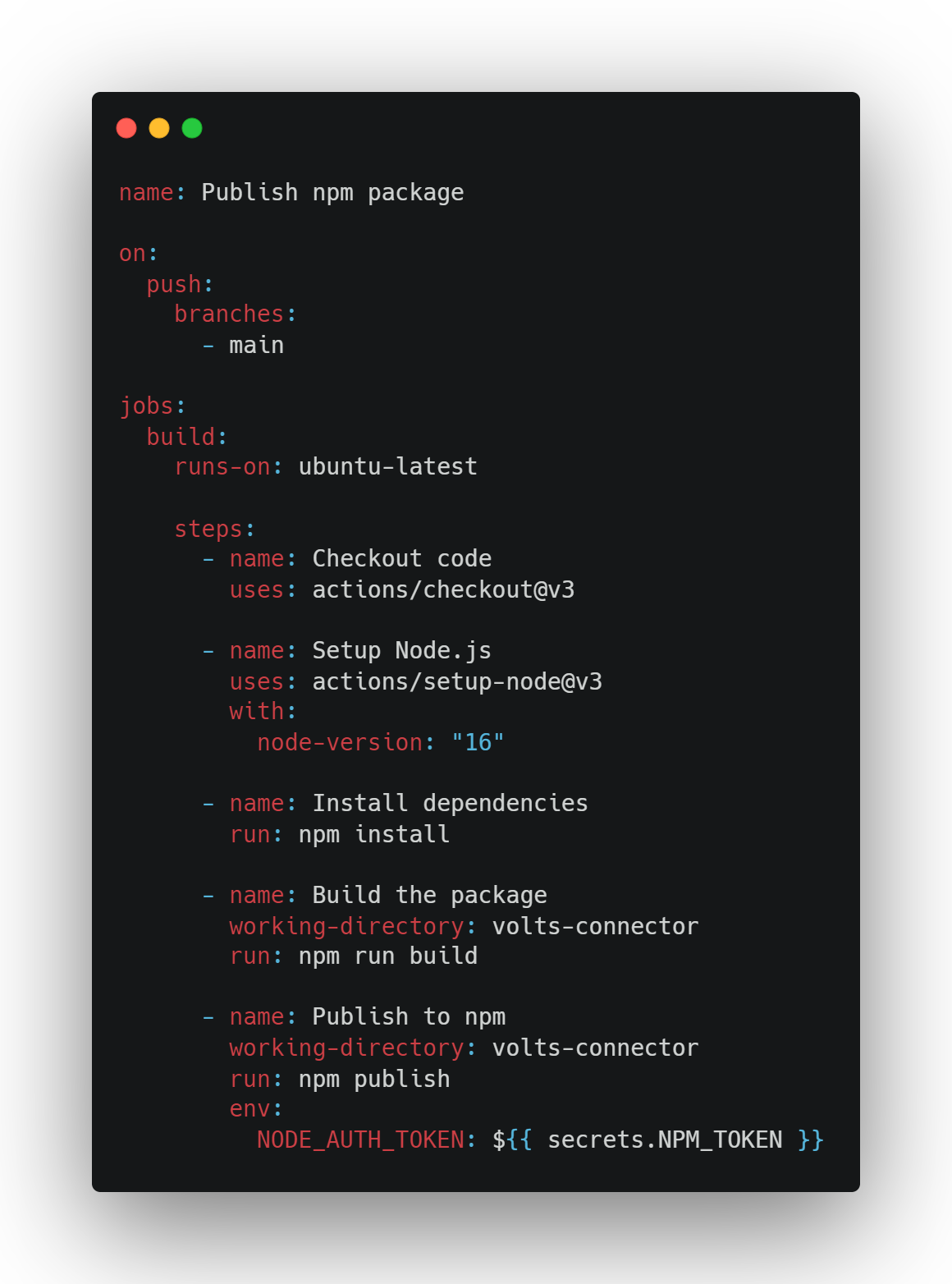
GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

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 работен поток илюстрира как GitHub Actions се 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.

### Работен поток за публикуване на npm пакет

Работният поток, дефиниран по-долу, автоматизира процеса на изграждане и публикуване на npm пакет при всяка промяна в основния клон (main) на хранилището. Това гарантира, че всяка нова версия на пакета се публикува автоматично в npm регистъра, без нужда от ръчно изпълнение на стъпките. Този работен поток е ключов за поддържане на актуални версии на софтуера инсталиран върху конекторите. Когато пакета е качен на облака може лесно потребителя да погледне активната версия и да я обнови. Тук инсталирането на нова версия зависи от потребителя тъй като веднъж инсталирайки и монтиране на устройството няма пряк достъп до него. Често един софтуер ако е отрит проблем той сам по себе си ще съобщи за грешка и нуждата от обновяване към нова версия. Когато е пакетиран кода и готов за инсталация в облака това се случва лесно. Затова ни е нужен този работен поток. За да може автоматично при излизане на нова версия да може да се пакетира и качи на облака. Така и улеснява създаването на нови дистанционни конектора.



Фиг. XIIX. Скрипт за работен потока пакетиращ и публикуващ към NPM облак

#### Обяснение на работния поток

**Име на работния поток**:

name: Publish npm package

Работният поток се нарича "Publish npm package", което указва, че основната му цел е публикуване на npm пакет.

**Тригери за изпълнение**:

on:

push:

branches:

- main

Работният поток се изпълнява при всяко push събитие към основния клон ("main") на хранилището.

**Дефиниране на работа**:

jobs:

build:

runs-on: ubuntu-latest

Определя се една работа с име "build", която се изпълнява на най-новата версия на Ubuntu.

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

* + **Изтегляне на кода**:

- name: Checkout code

uses: actions/checkout@v3

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

* + **Настройка на Node.js**:

- name: Setup Node.js

uses: actions/setup-node@v3

with:

node-version: "16" # Specify the Node.js version

Тази стъпка настройва Node.js версия 16, което е необходимо за изпълнение на npm команди и скриптове.

* + **Инсталиране на зависимости**:

- name: Install dependencies

run: npm install

Тази стъпка инсталира всички зависимости, дефинирани във файла package.json на проекта.

* + **Изграждане на пакета**:

- name: Build the package

working-directory: volts-connector

run: npm run build # Make sure your build script is defined in package.json

Тази стъпка изпълнява скрипта за изграждане на проекта, дефиниран в package.json файла. Работната директория е зададена на "volts-connector".

* + **Публикуване в npm**:

- name: Publish to npm

working-directory: volts-connector

run: npm publish

env:

NODE\_AUTH\_TOKEN: ${{ secrets.NPM\_TOKEN }}

Тази стъпка публикува изградените артефакти в npm регистъра. За аутентикация се използва секретният токен (NODE\_AUTH\_TOKEN), съхраняван в GitHub Secrets.

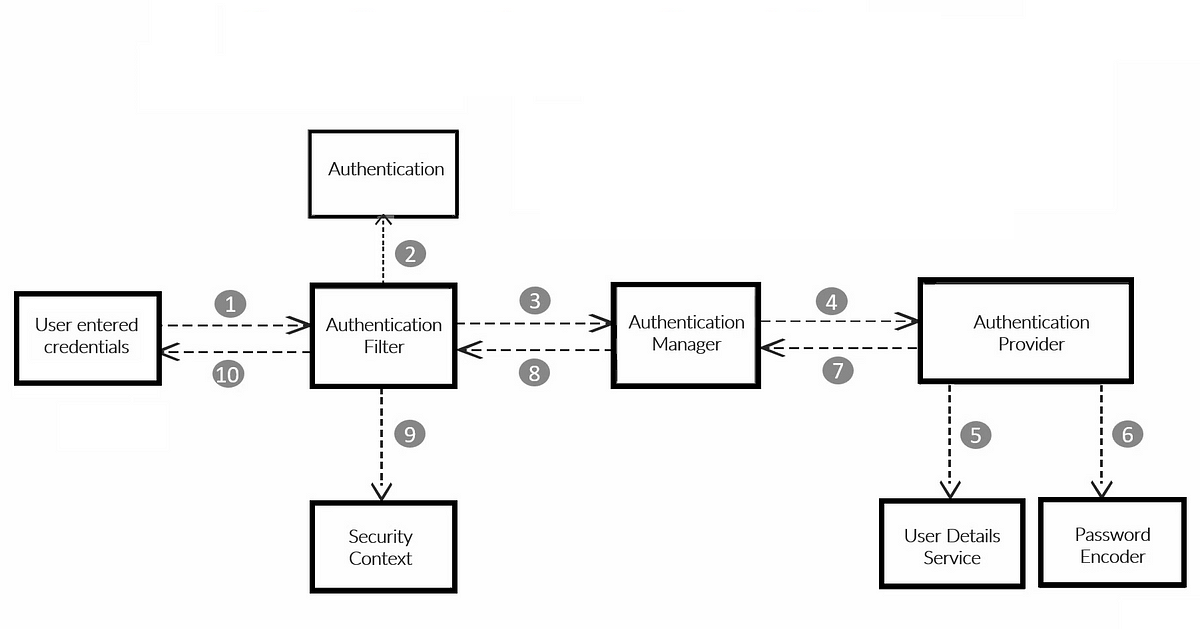
Този примерен работен поток илюстрира как GitHub Actions се използва за автоматизация на процесите по изграждане и публикуване на npm пакети. Автоматизирането на тези процеси осигурява последователност и надеждност при разпространението на софтуерни компоненти, като същевременно спестява време и усилия за разработчиците. Внедряването на подобни автоматизирани системи е ключов аспект на съвременната софтуерна разработка и допринася за по-ефективно управление на версиите и разпространението на софтуерни продукти.

# 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 слой

Защитният слой в MVC приложение, когато става дума за Java Spring Boot Gateway, представлява компонент, който служи за филтриране и контрол на заявките, преди те да достигнат до основната бизнес логика. Той се грижи за сигурността на приложението, като проверява и валидира входящите данни, контролира достъпа на потребителите и ограничава ресурсите, до които могат да се свържат. Gateway-ът обработва заявките чрез различни филтри, които могат да проверяват за автентикация и авторизация, да управляват потребителски сесии, както и да защитават приложението от потенциални заплахи като SQL инжекции [[8]](#footnote-8)и Cross-Site Scripting (XSS)[[9]](#footnote-9) атаки. Spring Security представлява рамка в Spring, която осигурява мощен и гъвкав механизъм за защита на приложенията. Тя се използва за внедряване на автентикация и авторизация в уеб приложения, като позволява на разработчиците лесно да добавят защита на ресурсите и да контролират достъпа до тях. Spring Security обработва входящите заявки, като проверява дали потребителят има необходимите права за достъп, и гарантира, че само оторизирани потребители могат да взаимодействат с определени части от приложението. Тази рамка включва и защита от различни типове атаки, като например Cross-Site Request Forgery (CSRF) и Clickjacking, и предоставя предварително дефинирани механизми за лесна интеграция със стандартни протоколи за автентикация, като OAuth2 и OpenID Connect.



Фиг. К0Л. Диаграма представляваща реда на работа в spring security

## Автентикация

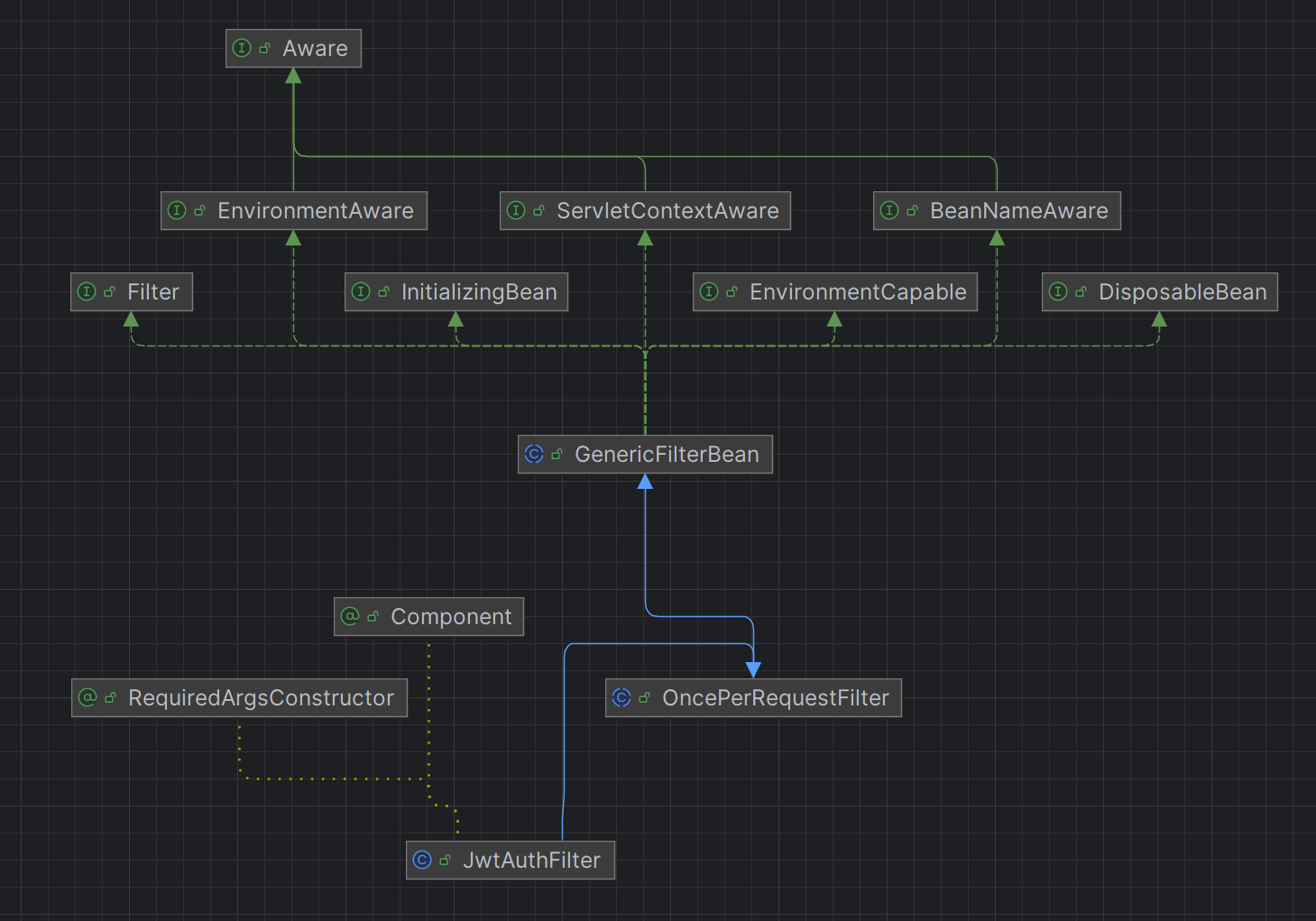
Автентикацията на потребителите в Spring Security се извършва на ниво защитен сервиз, като включва потвърждаване на самоличността на потребителя чрез предоставяне на данни за вход – емайл/потребителско име и парола. При успешна автентикация се генерира JWT (Json Web Token) токен, който служи като цифров подпис за потребителя и позволява лесна и сигурна комуникация между клиента и сървъра.

За да бъде потвърдена самоличността на потребителя, първо се приемат неговите данни за вход чрез бизнес обект, който представлява запис с полета за идентификация и парола:

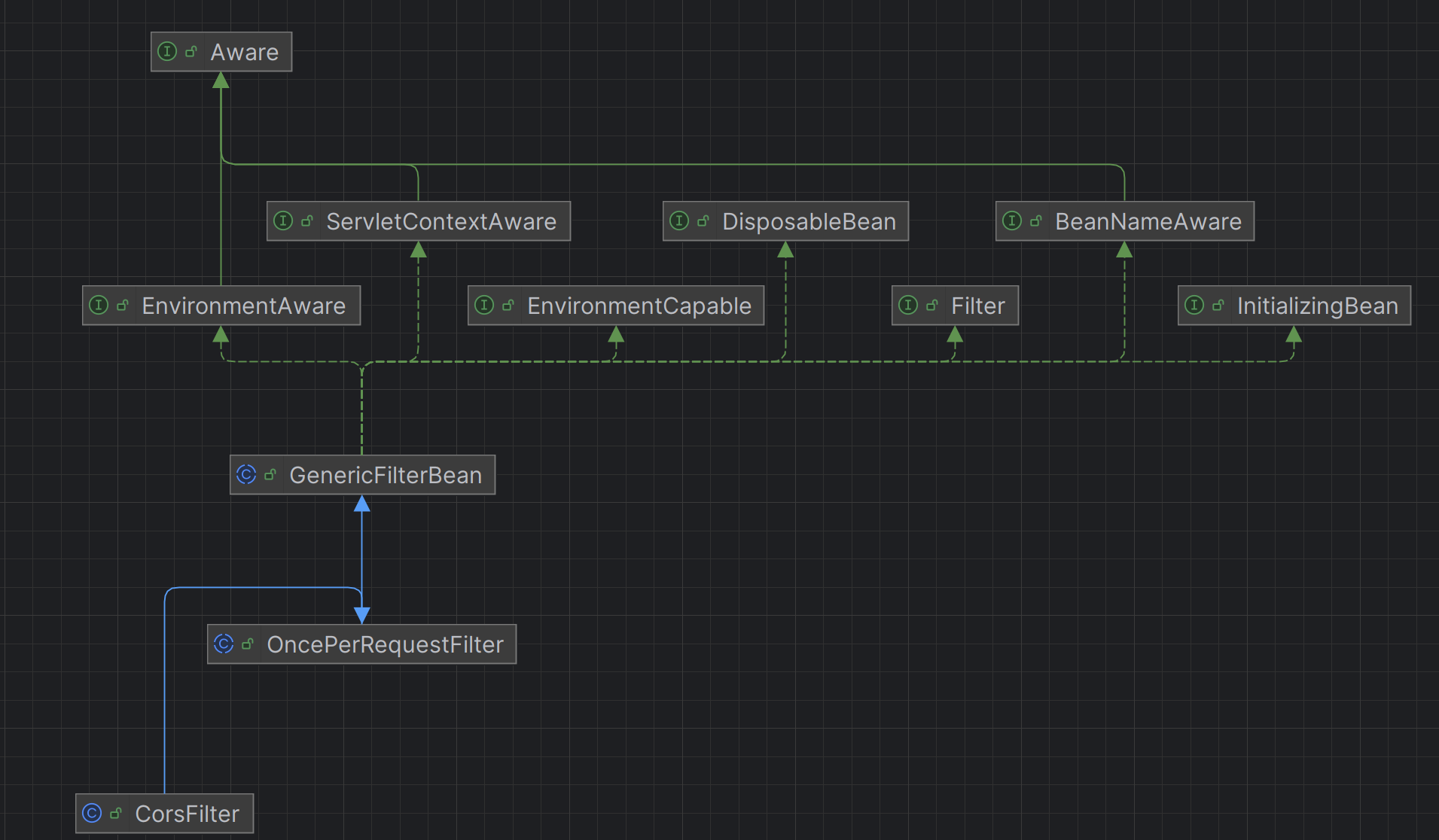
public record UserLogIn(String credentials, String password) {}

Този обект включва поле за идентификация, което може да бъде както емайл, така и потребителско име, предоставяйки два възможни начина за идентифициране на потребителя. Но за целите на проекта е прието да се изисква само емайл от потребителя. След като данните за вход се подадат на сървъра, те преминават през процес на верифициране, който проверява правилността на формата и типа на подадената информация. Всяка заявка към сървъра преминава през тези проверки, за да се гарантира, че данните са в очаквания формат и че са валидни.

Този процес на верификация и автентикация не само защитава приложението от неоторизиран достъп, но и поддържа интегритета на потребителските данни, като осигурява високо ниво на защита срещу атаки, като невалидни заявки, неправилни данни и опити за проникване.



Фиг. НХХ. Диаграма на наследяване за настройване на филтрите.



Фиг. НЙЙ. Фитриране на CORS

## Авторизация

Авторизацията в уеб приложенията служи за определяне на правата и достъпа на потребителите до специфични ресурси или функционалности в приложението, като по този начин установява кои потребители имат необходимите права за достъп до чувствителни данни и операции. Тя функционира като слой на защита, който ограничава достъпа само до оторизирани потребители, осигурявайки високо ниво на сигурност за съхранените данни и ресурси.

Системата за авторизация решава кой може да използва определени функционалности или да достъпва специфични данни, като това позволява персонализиран и сигурен потребителски опит. След успешна автентикация, приложението запазва информацията за сесията на потребителя, така че той да може да се придвижва в рамките на приложението с достъп до позволените ресурси, без да се налага многократно влизане. Това спомага и за предоставянето на персонализирани функционалности, които оптимизират потребителското преживяване.

Системата за авторизация поддържа и аудитни логове, които записват действията на потребителите в приложението, като така осигурява възможност за проследяване на дейности, идентифициране на проблеми и подпомага разследвания при сигурностни нарушения. Освен това, авторизацията играе критична роля в управлението на идентичността на потребителите, като включва контрол на пароли, обновяване на права за достъп и поддържа ефективна система за управление на потребителските роли и разрешения.

Като цялостен механизъм за сигурност и контрол на достъпа, авторизацията в уеб приложенията гарантира, че данните са защитени, а достъпът до ресурсите е управляван, което осигурява сигурност и стабилност на приложението и защита на потребителските данни.

## OAuth

OAuth 2.0 представлява индустриален стандарт за протокол за оторизация, който осигурява опростен и сигурен механизъм за управление на достъпа до защитени ресурси, без потребителят да разкрива своите дългосрочни идентификационни данни или лична информация. Този протокол, разработван от работната група IETF OAuth, е предназначен за уеб и мобилни приложения, настолни програми и дори устройства за ежедневна употреба. OAuth 2.0 предоставя различни потоци за оторизация, които са съобразени със специфичните нужди на приложенията, като същевременно улеснява разработчиците чрез стандартизирани и безопасни методи за достъп.

OAuth дава възможност на потребителите да предоставят контролирани разрешения на приложения на трети страни, като например достъп до снимки или лична информация, без да се налага да споделят директно своите пароли. Например, сайт за споделяне на снимки, който поддържа OAuth, може да позволи на потребителите си да използват услуга за печат на трета страна, която получава ограничен достъп до снимките им, без да може да управлява техния акаунт или да съхранява дългосрочните им идентификационни данни. По този начин OAuth се използва широко за влизане в приложения чрез акаунти от трети страни, като Google или Microsoft, улеснявайки първоначалния достъп на потребителите до приложението.

Използването на OAuth 2.0 намалява препятствията пред новите потребители, като съкращава процеса на регистрация, улеснявайки навлизането в приложението. Тази стратегия повишава ангажираността на потребителите, което в дългосрочен план може да се отрази положително на приходите, тъй като по-ниският праг за достъп увеличава шансовете те да останат активни в приложението.

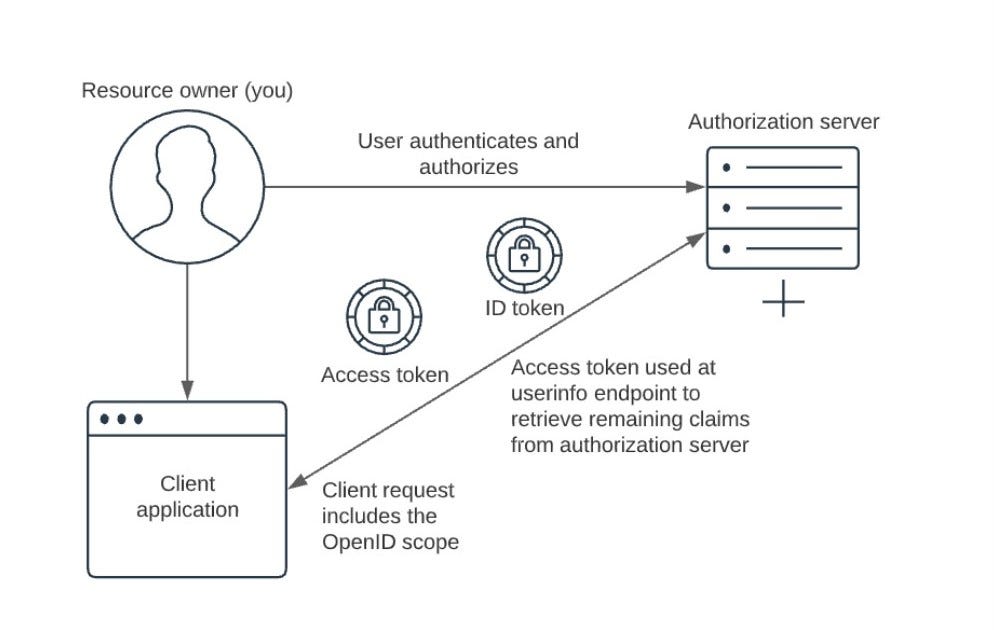
OAuth 2.0 включва следните ключови компоненти:

* Регистрация на клиента с разрешителния сървър;
* Протокол за получаване на токени за оторизация с предварителното съгласие на собственика на ресурсите;
* Протоколи за използване на токени за достъп до защитени ресурси.

OAuth е подходящ за уеб проекти, защото предоставя сигурен и ефективен начин за управление на достъпа до защитени ресурси, което намалява нуждата потребителите да въвеждат множество пароли или чувствителни данни директно в приложението. Той позволява на потребителите да влизат в системата чрез свои съществуващи акаунти, като тези в Google или Microsoft, което не само улеснява първоначалния достъп, но и повишава доверието в сигурността на приложението. Това е изключително полезно за уеб проекти, които се стремят да осигурят лесен и бърз вход за нови потребители, като в същото време запазват високи нива на защита на данните.

OAuth осигурява стандартизирана архитектура, която може лесно да бъде интегрирана в различни уеб и мобилни приложения. Чрез използването на токени за достъп, OAuth минимизира риска от излагане на лични данни и предоставя на уеб приложенията гъвкав механизъм за управление на правата на достъп. Освен това, внедряването на OAuth автоматизира и улеснява процесите за автентикация и оторизация, което намалява натоварването върху екипа по разработка, като същевременно оптимизира потребителския опит.

За проекти, които се конкурират за вниманието на потребителите в динамична среда, OAuth предоставя нисък праг за достъп и лесен начин за навлизане, което насърчава по-висока ангажираност и задържане на потребителите. Това прави OAuth особено подходящ за уеб приложения, които се стремят да осигурят сигурен и удобен потребителски интерфейс, който привлича и запазва вниманието на клиентите.



Фиг. ГГХ. Диаграма представляваща работния процес на OAuth 2.0.

Използването на един унифициран протокол за оторизация и сигурност стандартизира работния процес между различните екипи и разработчици, като насърчава спазването на утвърдени правила за защита на данните. Протоколът предоставя възможност за създаване на токени, които да персистират сесиите на потребителя, като в този случай е избран стандартът JWT (Json Web Token). JWT се използва широко поради своята универсалност и лекота при управление на сесии между сървъра и потребителския интерфейс.

Както ще бъде разгледано по-долу, тези токени се съхраняват в база данни от тип библиотека, която предоставя висока скорост при запис и извличане на данни. Това избрано решение допринася за бързото управление на сесиите и оптимизира производителността на системата за автентикация и оторизация.



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

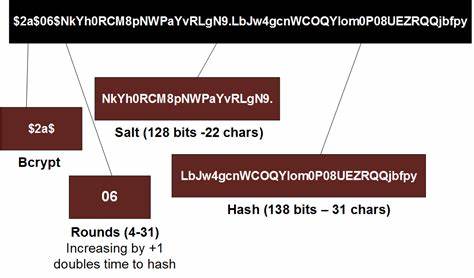
При съхранение на пароли в база от данни, приложението използва процес, наречен "хеширане," за да защити паролите от неправомерен достъп. Вместо да съхранява самата парола, системата създава и записва уникален "хеш" – криптографски преобразуван вариант на паролата. Хешът представлява поредица от символи, която е невъзможно да бъде върната обратно към оригиналната парола. Често се използват алгоритми като bcrypt, Argon2 или PBKDF2, които са специално разработени за хеширане на пароли и включват допълнителна защита, като „сол“ – уникална стойност, добавена към паролата, преди да бъде хеширана. Тази сол предотвратява използването на т.нар. "rainbow таблици" – предварително изчислени хешове, които злонамерени лица могат да използват за разбиване на пароли.

Сигурното хеширане на пароли е от съществено значение за защита на потребителските данни и предотвратяване на злоупотреби. Ако базата данни бъде компрометирана, съхраняването на хешове вместо действителните пароли значително намалява риска от достъп до потребителските акаунти, защото хешовете не могат лесно да бъдат обърнати в оригинални пароли. Хеширането също така защитава потребителите, които използват една и съща парола в различни платформи – ако тяхната парола бъде разкрита в една система, вероятността тя да бъде използвана за достъп до други системи е намалена.

Този подход създава допълнително ниво на защита и осигурява висока степен на сигурност за потребителите на приложението, като гарантира, че дори при компрометиране на базата данни, личните данни на потребителите остават защитени.

BCryptPasswordEncoder е клас в Spring Security, който се използва за хеширане на пароли, като прилага алгоритъма bcrypt – един от най-популярните и сигурни алгоритми за тази цел (Фиг. ММК). Този енкодер създава уникален хеш за всяка парола, като добавя „сол“ (random salt), което предпазва от атаки с предварително изчислени таблици (rainbow tables) и прави хеширането по-трудно за разбиване чрез brute-force атаки[[10]](#footnote-10). При всяко ново хеширане се генерира уникален хеш, дори ако паролата е същата, което допринася за допълнителна сигурност.

BCryptPasswordEncoder се използва основно за съхранение и проверка на пароли в приложения, където защитата на потребителските данни е критична. При създаване на потребителски акаунт или при промяна на парола, класът генерира хеш на паролата, който се съхранява в базата данни. При опит за вход в системата, BCryptPasswordEncoder проверява дали подадената парола съответства на съхранения хеш, без да разкрива оригиналната парола.



Фиг. ММК. Изображение представящо елементите на хеширащия алгоритъм Bcrypt

Предимства на BCryptPasswordEncoder:

* Сигурност: bcrypt алгоритъмът е устойчив на brute-force атаки благодарение на използваната сол и забавянето при генериране на хешове.
* Надеждност: bcrypt автоматично генерира различни хешове за една и съща парола чрез уникалната си сол, което прави хешовете уникални и трудни за предвиждане.
* Конфигурируемост: BCryptPasswordEncoder позволява настройка на сложността чрез параметъра "work factor", който определя колко ресурсоемък е процесът на хеширане. Това позволява на приложението да увеличи сигурността, като настройва work factor-а в зависимост от наличните ресурси и изискванията за сигурност.

Недостатъци на BCryptPasswordEncoder:

* Бавно изпълнение: Алгоритъмът bcrypt е предназначен да бъде "бавен", за да се предотвратят brute-force атаки, но това може да бъде недостатък при системи с голям обем от потребители или при приложения, които изискват висока производителност.
* Не е подходящ за нехеширани данни: BCryptPasswordEncoder е специално създаден за хеширане на пароли и не е оптимален за обработка на други видове данни, които може да изискват различни методи на защита.

BCryptPasswordEncoder се препоръчва за уеб приложения, където защитата на потребителските пароли е приоритет, тъй като осигурява силно ниво на сигурност и надеждност.

# Микросервизи

Микросервисите представляват архитектурен подход в софтуерното развитие, при който едно приложение се разделя на малки, независими компоненти, наречени сервиси. Всеки сервис е отделен модул с ясно дефинирана отговорност и обикновено изпълнява конкретна бизнес функция, като например управление на потребители, обработка на плащания или управление на инвентара. Микросервисите комуникират помежду си чрез леки протоколи като HTTP или съобщения, използвайки API-та за взаимодействие. За какво се използват микросервисите:

Повишат гъвкавостта на разработката. Позволяват на различни екипи да работят независимо върху отделни сервиси, което ускорява разработката и улеснява поддръжката на софтуера.

Мащабират системата ефективно. Дават възможност за мащабиране само на конкретни сервиси, които имат по-високо натоварване, без да е необходимо да се мащабира цялото приложение.

Подобрят надеждността. Като един сервис спре да работи, той не засяга работата на останалите, което намалява риска от прекъсване на цялата система.

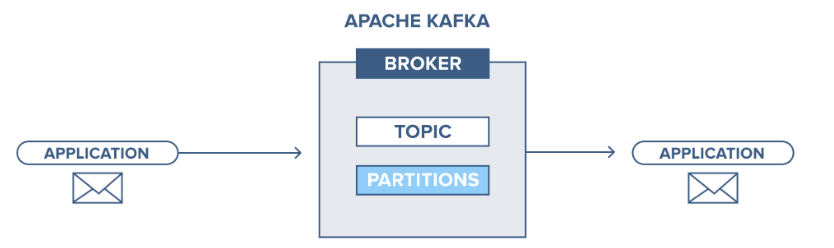
Улеснят внедряването на нови функции. Позволяват по-бързо внедряване на нови версии и функции, тъй като промените в един сервис не изискват пълно прекомпилиране или рестартиране на цялото приложение.

## Как се използват микросервисите

Микросервизите се използват по множествено начини в една система. За целите на проекта микро сервизите са използвани за:

Разделяне на бизнес функциите. Разработчикът първо идентифицира основните бизнес функции и ги разпределя в самостоятелни сервиси. Всеки микросервис обикновено има собствена база данни и свои API-та за комуникация. За целите на проекта бизнес функциите се разделят на ключови, такива които отговарят за главната функционалността на приложението, анализи, защита на приложението и препоръки. Ключовата функционалност включва всички сервизи за вземане на данни от базата данни отговорни за визуализацията на крайния клиент. Както и всички сервизи отговорни за вписване на нова информация в базата данни. Анализите отговарят за следене на потребителската дейност и как системата се справя с тази дейност. Тази официалност служи за подобряване на системата и реванширане на бъдещи промени чрез следене на трафик, товари и цялостна дейност на потребителите. Защитата на приложението се представя като отделна функционалност поради множеството движещи елементи които я изграждат. Тя главно служи за валидиране на клиента и неговия статус в приложението. Давайки му достъп само и единствено до функционалност предназначена само за него. Нищо повече нищо по малко. Отделяйки тази функционалност позволява преправяне на защитни практики и надграждане без рис към цялостната система. Отделената функционалност за препоръки служи за анализиране на входните данни на потребителя за аномалии след което ги прекарва през изкуствен интелект за да се генерират добри практики и препоръки за съответната клиентска аномалия. Тази функционалност работи незабележима само следейки клиентските данни за аномалии. Затова тя може да работи като отделна част от ключовите части на системата.

Осигуряване на комуникация. Сервисите комуникират помежду си чрез протоколи като REST, gRPC или чрез съобщения чрез message brokers като RabbitMQ или Apache Kafka. За целите на проекта е подбрано да се използва Kafka. Когато един програмен проект с микроуслуги използва Kafka за вътрешна комуникация между услугите, той избира този подход, защото Kafka осигурява надеждно, асинхронно и мащабируемо предаване на съобщения. Kafka се отличава със своята способност да обработва голям обем от данни в реално време, като съхранява съобщенията за определен период и позволява на различни услуги да ги консумират при нужда. Това дава на проекта по-голяма гъвкавост, тъй като позволява разделяне на логиката на услугите, което води до по-ниска зависимост между тях. Kafka също така осигурява устойчивост на грешки и възможност за репликация, което прави системата по-стабилна и надеждна. Чрез нея може да се избегне проблемът с претоварване на основните услуги, като същевременно улеснява обработката на данни в паралел, благодарение на механизма си за stream processing.



Фиг. КЙК. Изображение на фигура представяйки елементарно как Kafka приема и препраща известия.

Имплементация на мониторинг и логване. За да следи здравето на микросервисите, се добавят инструменти за мониторинг и логване. Това позволява на екипите бързо да откриват и отстраняват проблеми, които възникват в сервисите. За целите на проекта то се извършва от логващия сервиз който е отделен елемент от системата. Логващия сервиз работещ и с сервиза за анализ служат да покажат как системата работи като цяло. Всеки може да има представя как работи неговата система но докато не попадне системата в потребителските ръце само можеш да предполагаш как тя ще се използва.

Управление на разпределената среда. Тъй като микросервисите обикновено се хостват на разпределени среди (като Docker контейнери или Kubernetes), те се управляват и мащабират автоматично според нуждите. За целите на проекта се използва само Docker. Когато един програмен проект с микроуслуги използва Docker за управление на разпределената си среда, той прави това, защото Docker предоставя стандартизиран начин за опаковане, разгръщане и управление на всяка микроуслуга като отделен контейнер. Това позволява на проекта да гарантира, че всяка услуга разполага с нужната среда и зависимости, независимо от базовата инфраструктура. По този начин Docker осигурява консистентност на изпълнението между различните етапи на разработка и разгръщане – от локална среда до продукция. Docker също така позволява на проекта да скалира микроуслугите лесно, като стартира множество копия от един и същ контейнер при нужда от повече ресурси. Чрез Docker проектът постига по-бързо разгръщане, опростена поддръжка и автоматизация на операциите, което улеснява управлението на разпределената среда и намалява вероятността от конфликти между различните зависимости на услугите. За автоматизацията на всеки от микро сервизи е разгледано в детайл в главата за Автоматизация.

Микросървисите са подходящи за големи и комплексни системи, където изискванията към мащабируемостта и гъвкавостта са високи, като позволяват на компаниите да разработват и внедряват софтуер по-бързо и стабилно.

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

Gateway е точка за достъп, която приема входящи заявки от клиентите и ги насочва към правилните вътрешни сървиси или ресурси в системата. Той се използва, за да централизира маршрутизирането на заявките и да контролира трафика, като действа като посредник между клиента и вътрешните микросървиси или приложения. Gateway обикновено предоставя допълнителни възможности като автентикация, авторизация, ограничаване на скоростта на заявките (rate limiting), събиране на статистики, трансформация на заявките и други функции, които оптимизират и защитават връзката между клиентите и сървисите. Gateway се използва, за да:

Намали сложността на клиентите. Предоставя централен адрес за достъп до множество сървиси, така че клиентите да не се налага да познават структурата и адресите на всеки отделен сървис. Това позволява лесно добавяне на нови модули които извикват функционалности на системата. С други думи добавянето на извикващи методи е лесно тъй като системата не си променя входните точки. Въпреки че системата вътрешно може да се е променила тя ще поддържа същите крайни точки.

Контролира и филтрира трафика. Прилага правила за ограничаване на скоростта на заявките и предпазва от неоторизиран достъп. Ограничаването на трафика е ключово при големи натоварвания. Така се предпазва от множество видове атаки над системата. Както и филтрирайки само държави от където е подържана системата и не допускайки връзки от други държави.

Осигурява сигурност. Интегрира механизми за автентикация и авторизация, които проверяват дали заявките имат нужните права. Тази функционалност е ключова за начина по който работи системата. Системата използва spring security, като модул отговорен за сигурността на системата и предоставя функционалност за автентикация, авторизация и серийност. В детайли е разгледано в главата отделена по темата.

Оптимизира представянето. Събира метрики, които помагат за анализ на трафика, и прилага кеширане, което намалява времето за отговор. Всеки достъп до системата се записва в лог система.

Улеснява поддръжката на системата. Дава възможност за по-лесно управление и актуализиране на маршрутите без необходимост от промени в клиентските приложения. Видове Gateway включват:

API Gateway. Той обслужва заявки към API-та на сървиси. Примери за такива са Spring Cloud Gateway, Kong и Amazon API Gateway. Обикновено се използва в микросървисни архитектури, където сървисите са разпределени и управлението на достъпа до тях трябва да е централизирано.

Reverse Proxy Gateway. Изпълнява функциите на прокси сървър, който приема заявките и ги пренасочва към сървиси зад мрежата, като например Nginx или HAProxy. Освен маршрутизация, този тип Gateway може да включва кеширане, компресиране и други функции за оптимизация на производителността.

Service Mesh Gateway. Често се използва в архитектури с Kubernetes, за да управлява комуникацията между различни сървиси в клъстера. Този вид Gateway осигурява подробен контрол над трафика вътре в инфраструктурата чрез мрежови политики и мониторинг на комуникациите. Пример за такъв е Istio Gateway.

Gateway играе ключова роля в съвременните архитектури, като предлага сигурност, управление на трафика и централизирано обслужване на клиентските заявки, което улеснява изграждането на мащабируеми и стабилни системи.

Spring Gateway представлява модул в екосистемата на Spring, който служи като API Gateway — входна точка за маршрутизиране на заявки към различни вътрешни микросървиси в дадена система. Той действа като филтър, който позволява управление на потока на заявките, обработка на маршрутизиране, управление на сесии и защитни механизми, както и прилагане на логика за трансформиране и валидиране на заявките. Spring Gateway се използва, когато е необходимо централно управление на трафика между различни микросървиси, като помага за опростяване на достъпа до тях, намалява времето за отговор и подобрява сигурността.

## Вътрешна микросервизна комуникация

## Сервиз за анализ ( Volts-analytics)

Analytics е процес на събиране, обработка и анализ на данни с цел извличане на информация и вземане на информирани решения. То включва събиране на различни типове данни като потребителско поведение, финансови показатели и оперативна ефективност, които след това се анализират за откриване на тенденции и модели.

Analytics се използва в широк спектър от индустрии, като дигитален маркетинг, бизнес мениджмънт, здравеопазване и образование. Например, то се прилага за измерване на ангажираността на потребителите в уебсайтове и мобилни приложения, като показва кои страници са най-посещавани или кои функции са най-използвани.

Analytics се използва чрез различни инструменти и платформи, като Google Analytics, Adobe Analytics и Power BI. Те събират данни и предоставят визуализации и отчети, което помага на потребителите да разбират и подобряват своите услуги и продукти.

Analytics включва различни методи за събиране на данни, които помагат за разбиране на потребителското поведение и оптимизация на услугите.

Проследяване на потребителя ( User Tracking) проследява действията и пътя на потребителя в уебсайтове и приложения. То събира данни за кликове, навигация и прекарано време, като помага за анализ на ангажираността и идентифициране на най-популярните функции и страници.

Събиране на данни чрез формуляри включва информация, която потребителите въвеждат при регистрация или при попълване на контактни форми. То дава информация за демографски данни, интереси и предпочитания на потребителите.

A/B тестиране разделя потребителите на групи, които виждат различни версии на една и съща страница или функционалност. По този начин то събира данни за ефективността на различни дизайни и помага за избор на най-оптималния вариант.

Проследяване на събития (Event Tracking) събира информация за конкретни действия на потребителя като кликване на бутон, гледане на видео или добавяне на продукт в количката. Това помага да се разберат взаимодействията на потребителите с ключови елементи от платформата.

Топлинни карти (Heatmaps) визуализират къде и как потребителите кликат или задържат вниманието си на страницата. Те събират данни за най-активните зони, които привличат внимание, и помагат за подобрения в дизайна.

Всеки от тези методи предоставя ценна информация за потребителското поведение и спомага за оптимизиране на продуктовото изживяване.

### Проследяване на потребителя ( User tracking)

Проследяване на потребителя, познато и като User tracking, е процес, който позволява на софтуерно приложение да следи поведението и взаимодействията на потребителя с цел събиране на данни за него. То събира информация като кликове, време, прекарано на определена страница, действия в реално време и друга активност. Така приложението разбира по-добре как потребителят използва услугите му и може да подобри функционалността, както и персонализирането на изживяването.

Съществуват множество софтуерни продукти които са концентрирани единствено върху тази тема. Пример за проследяване на потребителските движения са:

Google Analytics следи активността на потребителите на уебсайтове и мобилни приложения. То събира данни за поведението на потребителите, като посещавани страници, време, прекарано на сайта, и извършени действия, което помага за анализ и оптимизация на съдържанието.

Mixpanel събира подробна информация за взаимодействията на потребителя с различни функции на приложението. То анализира действията на потребителите в реално време и предоставя статистики, които позволяват на разработчиците да подобрят функционалността и потребителското изживяване.

PostHog проследява взаимодействията на потребителите в уебсайтове и приложения, като събира данни за поведението им без необходимост от външни услуги. То анализира действия като кликвания, превъртания и използване на определени функции, предоставяйки на разработчиците информация за подобрения и персонализиране на потребителското изживяване.

### Събирането на данни чрез формуляри

Събирането на данни чрез формуляри събира информация от потребителите чрез попълване на полета с лични или демографски данни. То изисква потребителят да въведе информация като име, имейл, възраст и интереси, което помага на бизнеса да разбере по-добре аудиторията си. Този метод се използва често при регистрации за услуги, абонаменти и анкети, като събраните данни се анализират за подобряване на потребителското изживяване и персонализация на услугите.

Google Forms събира информация чрез формуляри, които потребителите лесно попълват онлайн. То предлага различни типове въпроси и автоматично организира събраните данни в електронна таблица за анализ.

Typeform събира данни чрез интерактивни и персонализирани формуляри. То ангажира потребителите с визуално привлекателни въпроси, като предоставя данните в лесен за анализ формат.

JotForm събира информация, като позволява на потребителите да създават и персонализират формуляри с различни типове въпроси. То съхранява и организира събраните данни, което улеснява достъпа и анализа им.

### A/B тестиран

A/B тестиране разделя потребителите на две или повече групи, като всяка вижда различна версия на една и съща страница или елемент. То събира данни за това как всяка версия се отразява на потребителското поведение, като сравнява резултатите за всяка група. Чрез A/B тестиране се определя кой вариант води до по-добро потребителско изживяване и по-висока ефективност на съдържанието.

Facebook A/B тества различни версии на дизайн за бутоните, за да види коя версия води до по-голямо ангажиране на потребителите.

Amazon A/B тества различни цветове и текстове за своите бутони „Добави в количката“, за да определи кой вариант увеличава вероятността за покупка.

Netflix A/B тества препоръчваното съдържание на началната си страница, като представя различни филми и сериали, за да установи кой подбор увеличава вероятността потребителите да започнат ново предаване.

### Проследяването на събития (Event Tracking)

Проследяването на събития (Event Tracking) събира данни за конкретни действия на потребителя в уебсайтове или приложения. То дефинира и измерва "събития," които представляват всяко взаимодействие на потребителя с определени елементи, като бутони, видеа, или линкове. Event Tracking предоставя подробна информация за това как потребителите взаимодействат със съдържанието и кои елементи привличат най-голямо внимание. Събраните данни помагат на бизнеса да оптимизира дизайна, функционалността и потребителското изживяване.

Проследяването на събития се прилага, като се определят ключови действия (събития) и се събират данни за тяхното изпълнение. Примерите включват броя кликвания върху бутон, продължителността на видеопреглед и честотата на добавяне на продукт в количката. След това тези данни се анализират, за да се разбере ефективността на елементите и да се направят промени, ако е необходимо.

YouTube проследява събития като стартиране, паузиране и завършване на видеоклип, което помага за разбирането на ангажираността и за подобряване на предложенията за съдържание.

Shopify проследява събития като добавяне на продукт в количката, извършване на плащане и разглеждане на промоции, което помага за оптимизиране на потребителското изживяване и увеличаване на продажбите.

Spotify проследява събития като слушане на песен, добавяне на песен в плейлист и харесване, за да персонализира препоръките и да разбере предпочитанията на потребителите.

### Топлинните карти (Heatmaps)

Топлинните карти (Heatmaps) визуализират взаимодействията на потребителите с уебсайтове или приложения, като показват къде потребителите кликат, скролват или задържат мишката си. Те представят данни в цветове, които обозначават интензивността на активността: по-горещите зони, обикновено оцветени в червено или жълто, показват най-активните области, докато по-хладните зони, оцветени в синьо или зелено, показват по-малко взаимодействие. Топлинните карти предоставят ценна информация за начина, по който потребителите се движат и взаимодействат с съдържанието, и помагат за оптимизиране на дизайна и разположението на елементите.

Топлинните карти се прилагат чрез специализирани инструменти, които проследяват действията на потребителите. Те събират данни за кликвания, скролиране и движения на мишката, след което генерират визуализации, които позволяват на анализаторите да видят кои части от страницата привлекат най-много внимание. Тези визуализации помагат на дизайнерите и маркетолозите да вземат информирани решения относно промените в дизайна и разположението на съдържанието.

Crazy Egg предоставя топлинни карти, които показват къде потребителите кликат на уебсайта. То помага на собствениците на сайтове да оптимизират разположението на бутони и линкове.

Hotjar генерира топлинни карти, които визуализират скролиране и кликове на страницата, позволявайки на анализаторите да разберат до каква степен потребителите взаимодействат с различни елементи.

Mouseflow проследява движенията на мишката и кликванията, за да създаде топлинни карти, които показват как потребителите се движат по страницата. Това помага за откриване на проблемни области и за оптимизация на потребителското изживяване.

## Сервиз за препоръки(Volts-recomendations)

Сервиза за препоръки служи за анализиране, изчисляване и връщане на бизнес ориентирани препоръки на клиента. Целта на сервиза е да бъде полезен на крайния потребител, като му препоръчва добри практики и централизирани съвети относно неговата продукция и разходи.

### Технология

Технологията зад имплементацията

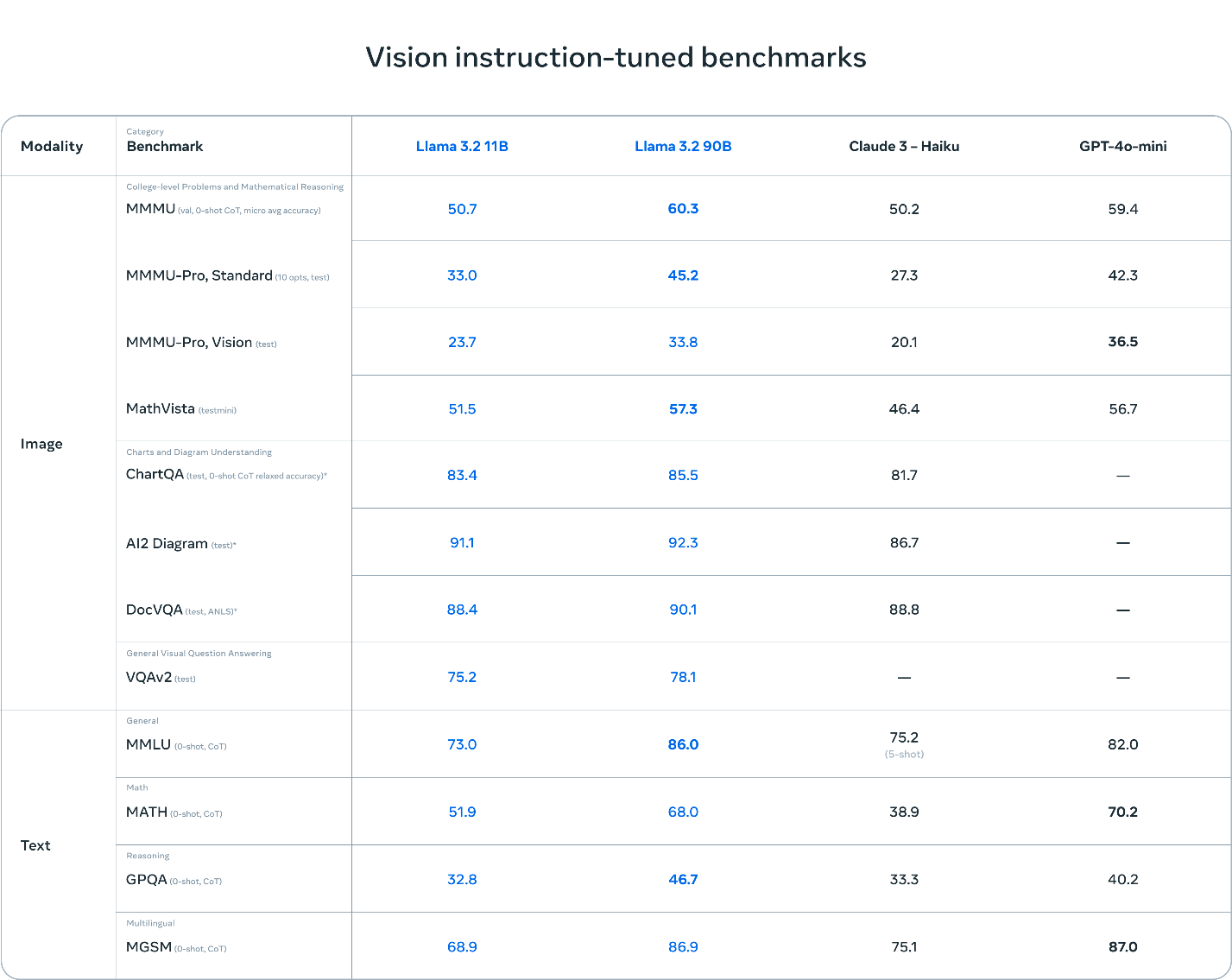
### Изкуствен интелект

Ollama е платформа с отворен код, създадена с цел да улесни използването на големи езикови модели (LLMs) на локални машини. Тя е разработена с акцент върху лесната инсталация и опростено управление на модели като Llama 2, Mistral и други. Ollama е насочена към разработчици, изследователи и ентусиасти, които искат локален контрол върху AI моделите си за задачи като генериране на текст, анализ на настроения или дори кодиране. Платформата е създадена с цел да бъде леснодостъпна и сигурна, като всички операции се изпълняват локално, без необходимост от облачни услуги, което предпазва потребителските данни и намалява разходите за използване на облачни ресурси.

Ollama се инсталира лесно на Linux сървъри чрез проста командна линия. За целите на поетка е стартирана на същата Linux машина както и всички контейнери. Самата инстанция на изкуствения интелект отнема 4 гигабайта от оперативната памет на машината, както и 90 гигабайта от пространство на харддиска.

Предимството на Ollama пред конкурентите като OpenAI е именно в това, че работи локално, което гарантира по-голяма сигурност на данните и елиминира зависимостта от външни сървъри. Докато OpenAI, с модели като GPT-4, е по-мощна платформа, тя изисква свързване към облак и често е по-скъпа, особено за по-малки компании. Ollama, от своя страна, предоставя по-достъпно решение, особено за потребители, които не разполагат с високопроизводителен хардуер или големи бюджети за облачни услуги.

Сред конкурентите на Ollama са и по-леки AI платформи като BERT и други модели с ниски хардуерни изисквания, но те често се нуждаят от повече ресурси и по-сложна настройка. Ollama предоставя по-лесна инсталация и използване, особено за по-неопитни потребители.



Фиг. НЙД. Оценка и сравнение на изкуствения интелект Llama по критерий свързани с текст и изображения, спрямо други модели изработени от Google и OpenAI.

 Фиг. НЙД. Оценка и сравнение на изкуствения интелект Llama по критерий свързани с конкретни задачи като общи знания, разсъждения, математика, използване на инструменти и многоезични умения., спрямо други модели изработени от Google.

### Имплементация

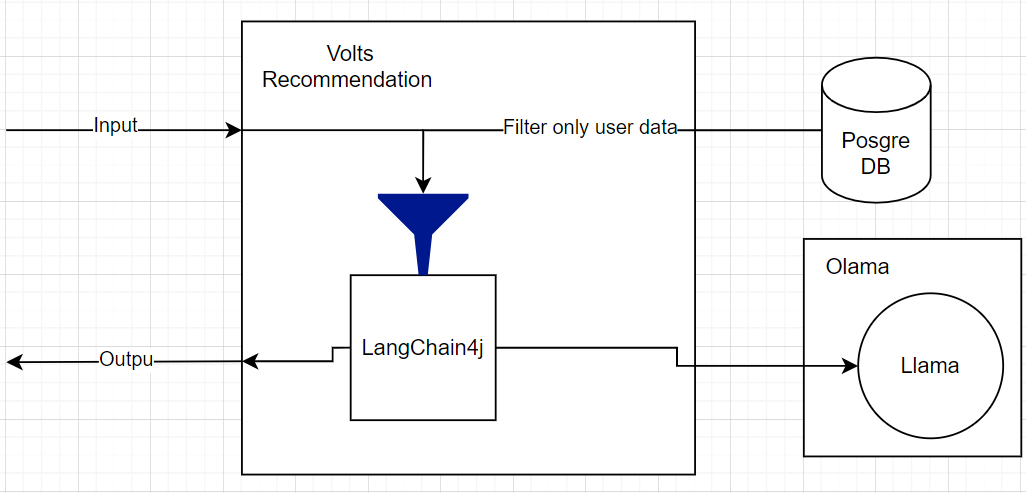
Имплементацията на сервиза за препоръки разчита главно на изкуствения интелект разгледан в предишния сервиз, както и на специализираната библиотека langChain4j. LangChain4J е библиотека за изкуствен интелект, създадена през 2023 година, като отговор на нуждата от подобна платформа за Java разработчици. Тя е вдъхновена от LangChain, популярна библиотека за Python, и има за цел да интегрира големи езикови модели (LLMs) и други AI технологии в Java приложения. Основната й роля е да свързва различни компоненти като LLM модели, текстови делители, изходни парсери и векторни хранилища, позволявайки създаването на персонализирани AI решения с минимален код.

LangChain4J предлага лесни за използване интерфейси и абстракции за работа с LLMs и поддържа интеграция с над 15 доставчика на AI модели, както и с много хранилища за векторно търсене и текстови модели. Разработчиците могат да създават приложения с поддръжка на обработка на естествен език и генеративен AI, както и да работят с мултимодални данни (текст и изображения).

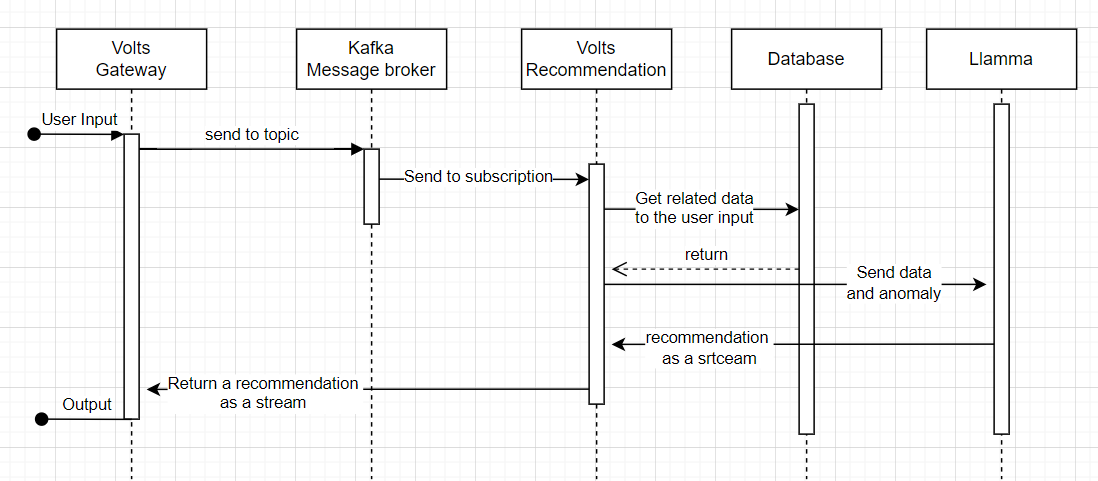
Сред предимствата на LangChain4J е фактът, че предоставя високо ниво на абстракция, което прави работата с LLM по-лесна за разработчиците, като скрива сложността на комуникацията с AI модели. Също така библиотеката предлага интеграция с Java фреймуърци като Spring Boot и Quarkus, което я прави подходяща за различни Java среди.

Конкурентни платформи като Haystack и LlamaIndex също предлагат подобни функционалности, но LangChain4J се откроява с по-добра съвместимост за Java и разширена поддръжка на различни модели и хранилища. Един недостатък, според някои разработчици, може да бъде, че библиотеката все още е в активна разработка и не всички функционалности са напълно завършени.

LangChain4J е идеална за разработчиците, които искат да създават AI приложения с Java без да навлизат в Python или други езици, и да използват AI услуги за специфични бизнес нужди. Точно затова е подбрана за използване в проекта. Библиотеката работи перфектно с Spring екосистемата която е избрана за проекта. Така лесно се имплементира абстрактен слой помежду клиента, неговите данни и изкуствения интелект. Нужна е този абстрактен слой, тъй като трябва безопасно да предадем само клиентските данни и нищо повече. Внимателно се постъпва когато всичките клиентски данни са в една база от данни и ги дели само един идентификационен номер. При липсата на подобен абстрактен слой, изкуствения интелект няма да открие разлика помежду данните на един и друг клиент. Затова употребата на междинен слой служи за ограничаване ресурсите на клиента.



Фиг. ККЛ. Структура на микро сервиза отговорен за препоръки. Възпроизведен като блок диаграма.



Фиг. ККЛ. Структура на микро сервиза отговорен за препоръки. Възпроизведен като диаграма на последователността.

Apache Log4j е популярна библиотека с отворен код, използвана основно в Java приложения за регистриране на системни поведения и събития. Тя е разработена през 1996 г. и широко се прилага както в малки, така и в големи проекти. Нейната цел е да улавя важни данни по време на изпълнение на приложенията за нуждите на дебъгинг, мониторинг и анализ. Тя използва йерархична структура на логерите, което позволява гъвкавост при регистриране на различни части от приложението. Log4j поддържа различни формати за изход и множество места за запис на логовете, включително файлове, бази данни и мрежови сокети.

Основните предимства на Log4j включват нейната ефективност, мащабируемост и гъвкавост, които позволяват на разработчиците бързо да диагностицират проблеми, по-лесно да поддържат приложенията и да оптимизират производителността. Въпреки това, както всяка логинг система, Log4j добавя известен товар на приложението, особено ако не е правилно конфигурирана. Лошо написан код за регистриране може да доведе до намаляване на производителността или ненужна сложност.

Разработчикът избира Log4j, защото тя поддържа различни изходни апендери, е нишково безопасна и ефективно управлява Java изключенията. Библиотеката също така е съвместима с други програмни езици като Python и C++ и се интегрира лесно с модерни инструменти за разработка като Maven и Gradle.

Въпреки своите предимства, Log4j има конкуренти в екосистемата на логинга, като SLF4J и java.util.logging, които могат да се предпочетат в определени ситуации поради различни набори от функции или по-проста конфигурация【34†source】【35†source】.

# Източници

Agile manifesto: [Манифест за Agile разработка на софтуер](https://agilemanifesto.org/iso/bg/manifesto.html)

https://agilemanifesto.org/iso/bg/manifesto.html

Източници за защитна стена:

<https://auth0.com/docs>

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

1. Agile Manifesto е документ, който формулира принципи за гъвкаво управление на софтуерни проекти, фокусирани върху адаптивност и сътрудничество. [↑](#footnote-ref-1)
2. Python е лесен за учене програмен език с широки приложения. [↑](#footnote-ref-2)
3. CSV, или Comma-Separated Values, е файлов формат, който съхранява таблични данни в текстов вид, като всеки ред представлява запис, а отделните полета са разделени със запетая. [↑](#footnote-ref-3)
4. SQL представлява език за управление на бази от данни, който се използва за създаване, манипулиране и извличане на данни от релационни бази. [↑](#footnote-ref-4)
5. ERD (Entity-Relationship Diagram) представлява графично представяне на структурата на база от данни. То визуализира основните обекти (ентитети) в системата, техните атрибути и връзките между тях. [↑](#footnote-ref-5)
6. ORM (Object-Relational Mapping) представлява технология, която позволява преобразуването на данни между обектно-ориентирани програми и релационни бази от данни [↑](#footnote-ref-6)
7. Entity представлява основна концепция в контекста на ERD и ORM. То обозначава обект или елемент от реалния свят, който трябва да бъде съхраняван и управляван в база от данни. [↑](#footnote-ref-7)
8. SQL инжекцията е атака, при която злонамерен потребител вмъква зловредни команди в заявка към базата данни, за да получи неоторизиран достъп. [↑](#footnote-ref-8)
9. Cross-Site Scripting (XSS) е атака, при която злонамерен код се внедрява в уеб страница и се изпълнява в браузъра на потребителя, позволявайки достъп до лична информация или манипулация на сесиите. [↑](#footnote-ref-9)
10. Brute-force атаката представлява метод за отгатване на парола чрез систематично изпробване на всички възможни комбинации, докато не се открие вярната. [↑](#footnote-ref-10)