# 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

# 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 комуникационния 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 ).

## REST Комуникация

РЕСТ (Representational State Transfer ) комуникацията или още позната като REST-FULL, представлява архитектурен стил за проектиране на мрежови приложения. Той се използва за комуникация между клиент и сървър чрез HTTP протокола. Основната идея на REST е да се използват стандартни HTTP методи, като GET, POST, PUT, DELETE, за изпращане на заявки към сървъра и получаване на отговори. Когато клиентът изпраща заявка, REST използва URL адресите, за да идентифицира ресурси. Тези ресурси могат да бъдат всичко – данни, услуги или други видове информация. След като сървърът получи заявката, той връща отговор, който обикновено е в JSON или XML формат. REST комуникацията се използва в разработката на уеб услуги и API-та, където сървърът и клиентът могат да бъдат различни системи. Това позволява на приложенията да взаимодействат помежду си, без да зависят от конкретни технологии или платформи. REST архитектурата е популярна поради своята простота, мащабируемост и лекота на интеграция. REST се използва пред други протоколи поради своята простота, лекота на използване и гъвкавост. Той не изисква сложни формати или допълнителни слоеве, както правят някои други протоколи като SOAP, което го прави по-лесен за разбиране и внедряване. REST използва стандартни HTTP методи като GET, POST, PUT и DELETE, които вече са добре познати и широко използвани в уеб разработката. Освен това, REST е лек и не налага строги ограничения върху формата на данните, като позволява използването на JSON, XML или дори обикновен текст. Това го прави по-гъвкав в различни среди и приложения. REST също така се интегрира лесно с уеб браузъри и мобилни приложения, тъй като използва основни принципи на интернет комуникацията. Една от ключовите причини, поради които се предпочита, е неговата мащабируемост и способността да поддържа голям брой клиенти и заявки. В сравнение с други протоколи, REST е по-ефективен и оптимизиран за работа с големи системи и микроуслуги. И затова е избран за главен комуникационен протокол за проекта.

## Брокер на съобщения сKafka

# Защитен сервиз

## Автентикация

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

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

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

## Авторизация

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

## OAuth

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

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

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



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

Криптиране на пароли

## База от данни отговорна за потребителски сесии

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

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



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

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

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

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

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

# Микро сервизи

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

## Сервиз за сигурност ( Volts-security)

## Сервиз за анализ ( Volts-analytics)

## Сервиз за изкуствен интелект (Volts-ai)

# Потребителски интерфейс

Потребителският интерфейс представлява частта от софтуера, с която потребителят взаимодейства директно. Той служи за мост между потребителя и функционалностите на дадена програма или приложение. Интерфейсът включва визуални елементи като бутони, менюта, полета за въвеждане на текст, икони и графики, които улесняват навигацията и изпълнението на задачи.

Чрез потребителския интерфейс софтуерът предоставя достъп до своите функции по ясен и интуитивен начин. Той играе ключова роля за потребителското изживяване, като влияе на това колко лесно и приятно потребителят може да използва даденото приложение или система. Добре проектираният потребителски интерфейс прави взаимодействието с софтуера по-ефективно, като намалява броя на грешките и подобрява производителността.

## UI/UX

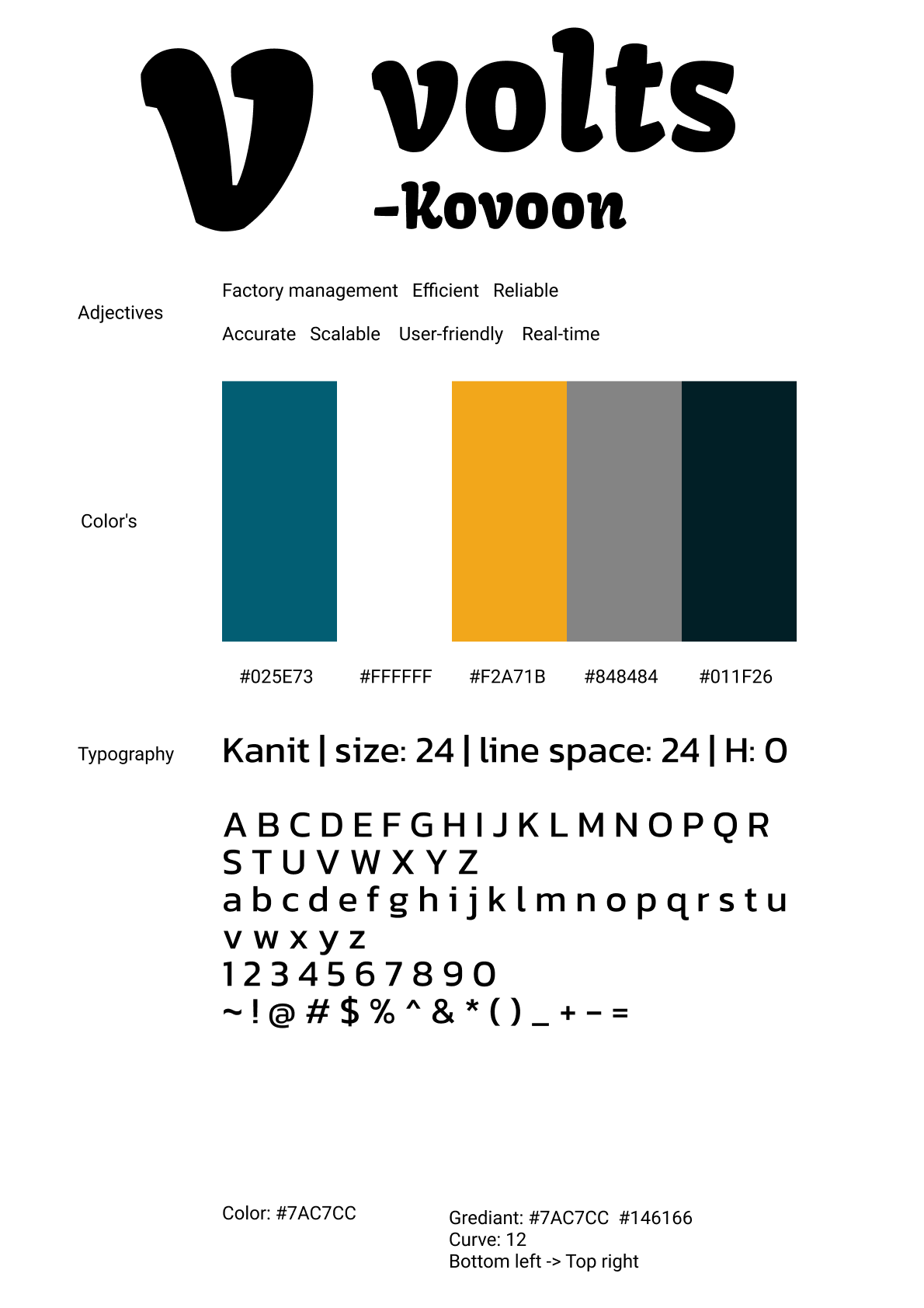
UI (User Interface) и UX (User Experience) са два свързани, но различни аспекта на дизайна, които засягат взаимодействието на потребителя със софтуер или продукт. UI (User Interface) се отнася до визуалната част на продукта, тоест как изглежда и как потребителят взаимодейства с него. Това включва елементи като бутони, менюта, икони, цветове, типография и оформление. Основната цел на UI дизайна е да направи интерфейса привлекателен, интуитивен и лесен за използване. UI дизайнът се фокусира върху естетиката и върху това как потребителят вижда и използва елементите на интерфейса. UX (User Experience) се отнася до цялостното изживяване на потребителя при взаимодействие с продукта или услугата. Това включва усещанията, емоциите и удовлетвореността на потребителя, както и това колко лесно той постига целите си в рамките на продукта. UX дизайнът обхваща не само интерфейса, но и структурата, навигацията и логиката зад продукта, така че взаимодействието да бъде възможно най-приятно и ефективно. Целта на UX е да осигури лесна, логична и полезна интеракция. Връзката между UI и UX е, че те са взаимно зависими. Докато UX се фокусира върху това как продуктът работи и как потребителят го преживява, UI се занимава с това как той изглежда и как се използва визуално. Добрата UX без добре изпълнен UI може да доведе до функционален, но неестетичен продукт, докато добрият UI без правилен UX може да направи продукта красив, но труден за използване. Идеалният баланс между тях осигурява както приятна визия, така и удобство при използване.

В проекта UI/UX е помислен и установен след дълги консултации с предметната зона и експерти в нея. Когато се задава UX посоката на софтуера първо трябва да разпишем ключовите елементи, които софтуера притежава. Тези ключови елементи са от голяма час от пазарната оценка разгледана в предишна глава. Когато се разполага с такава пазарна оценка се предвижда спрямо конкуренцията какви са клиентските очаквания от самия софтуер. В тези очаквания фигурират: функционалности, разположение на итеративните елементи на интерфейса и цветовите наклонности към елементи от интерфейса. Тези клиентски наклонности са строго свързани с пазара в който се намира софтуера. Крайния клиент винаги търси нещо сходно до продукт, който ползва или нещо поне познато за него. Това се отнася към всички групи софтуер. Например социалните медии които имат много сходни функционалности: създаване на публикации, качване на снимки, чатове с приятели. Както и повечето социални мрежи имат сходни цветове.

Официалния документ за вкарване на тези правила към UI и UX се нарича таблица със стилове( style sheet). Style sheet представлява документ или файл, който определя визуалните стилове и оформления за потребителския интерфейс (UI) на даден софтуер. Той съдържа правила за това как различни елементи на интерфейса, като бутони, заглавия, текстови полета и фонове, трябва да изглеждат и какви цветове, шрифтове и размери да използват. В контекста на уеб разработката, най-често се използва CSS (Cascading Style Sheets), за да се описват стиловете на HTML елементи. Style sheet се използва за осигуряване на консистентност и единство в дизайна на UI. Когато всеки елемент от интерфейса следва едни и същи стилови правила, потребителят получава по-последователно и приятно изживяване (UX). Разработчиците и дизайнерите използват style sheet , за да отделят визуалните аспекти на софтуера от неговата логика и функционалност, което прави поддръжката и актуализацията на интерфейса по-лесна и ефективна. За проекта е създаден style sheet спрямо всички изисквани и проучвания по пазара, клиентите и конкуренцията (фиг. Xtxty).

Шрифтът Kavoon използван като главен в проекта и е създаден от полската дизайнерка Виктория Грабовска и представлява декоративен дисплей шрифт, вдъхновен от експерименти с четка и мастило. Kavoon се отличава със своята игрива и позитивна визия, правейки думите по-изразителни и живи. Шрифтът е подходящ за заглавия или друг текст, който изисква по-големи размери, като може да се използва за уеб и печатни проекти. Основната му цел е да придаде индивидуалност и визуална привлекателност на дизайна, като често се използва за лога, плакати и други проекти, които изискват артистичен или ретро вид. Той е безплатен за лична и комерсиална употреба, като се разпространява под лиценз OFL (Open Font License), което позволява свободно използване и разпространение.

Цветовете описани в таблицата със стилове биват подбрани конкретно спрямо конкуренцията и техните цветове. Главния цвят използван е синият. Синият цвят конкретно към софтуер придава чувство за пропорционалност и доверчивост. Синия цвят се свързва с много успешни софтуерни компании, като: IBM, Pay pall, Dell, HP, Cisco и много други. Избран е жълтия цвят като акцентен цвят. Жълтото седи на противоположна страна на синьото. Цветове които седят на обратни страни на колелото на цветовете си подхохождат и този начин на извиране на цветове се нарича комплементираща цветна схема. Комплементарните цветове създават висок контраст и визуално привличане, защото взаимно се подсилват. Например, червено и зелено, синьо и оранжево, или жълто и лилаво са класически примери за комплементарни двойки. Тази цветова схема често се използва, за да се създаде силно въздействие в дизайна и визуалното изкуство.

Фиг. № xtxty

## Уеб интерфейс

Уеб интерфейсът представлява интерфейсът, който потребителят използва за взаимодействие с уеб базирани приложения или уебсайтове. Той се зарежда и използва чрез уеб браузър и предоставя визуални и интерактивни елементи като бутони, връзки, форми и менюта, за да улесни навигацията и управлението на съдържанието.

Чрез уеб интерфейса софтуерът предоставя функционалностите си в достъпен формат, който е независим от конкретна операционна система или устройство. Уеб интерфейсът използва стандартни уеб технологии като HTML, CSS и JavaScript, за да осигури динамично съдържание и интерактивност. Той позволява на потребителя да извършва действия като търсене на информация, попълване на форми, качване на файлове и други, директно през уеб браузъра. В проекта за реализация на уеб интерфейс е използван така наречения front-end framework.

Front-end frameworks представляват софтуерни библиотеки, които предоставят готови инструменти и структури за създаване на потребителски интерфейси и изграждане на уеб приложения. Те улесняват процеса на разработка, като предлагат предварително дефинирани компоненти, шаблони и структури, които могат да бъдат използвани повторно.

Чрез използването на front-end frameworks разработчикът ускорява създаването на уеб приложения, като се фокусира върху специфичната логика и дизайн, вместо да пише всичко от нулата. Тези фреймуъркове включват инструменти за управление на оформление, стилове и динамични елементи, както и за работа с различни устройства и браузъри. Примери за такива фреймуъркове са React, Angular и Vue.js, които предлагат ефективни и добре организирани структури за изграждане на сложни и интерактивни уеб интерфейси. За реализацията на проекта е избран Astro фреймърк със допълнени React компоненти за допълнителна реактивност на уеб интерфейса.

Astro фреймърк

Astro.js представлява модерен JavaScript фреймуърк, който се използва за създаване на уебсайтове и статични уеб приложения с висока производителност. Той е проектиран с цел да подобри скоростта на зареждане и оптимизацията на уебсайтове, като минимизира количеството JavaScript, което се изпълнява на клиентската страна. Astro.js следва принципа "Island Architecture", където JavaScript се зарежда само за интерактивните компоненти, а останалото съдържание е статично.

**Island Architecture** представлява подход за изграждане на уебсайтове, при който отделни части от страницата (или „острови“) са интерактивни и динамични, докато останалата част от съдържанието е статична. Този модел се фокусира върху това да се зарежда JavaScript само за специфичните интерактивни компоненти, без да се обременява цялата страница с ненужен код. Тъй като на нас ще ни е нужно да зараждаме само графики и форми за въвеждане на енергетика и произведена стока. Тази методология се вписва перфектно в начина на работа на приложението.

В Island Architecture, всяка "островна" секция се държи като отделен модул, който може да бъде рендериран независимо от останалите части на страницата. Например, интерактивни елементи като формуляри, галерии, или чат модули са острови, докато статичното съдържание (като текстове и изображения) се рендерира като обикновен HTML. Тази архитектура оптимизира зареждането на страниците, защото ненужният JavaScript не се изпълнява на клиентската страна за статичните части.

Island Architecture се заражда като отговор на необходимостта от по-ефективно управление на JavaScript в модерните уеб приложения. Този архитектурен модел става популярен с възхода на статичните сайтове и интензивната оптимизация за производителност в уеб разработката.

Исторически, Island Architecture произлиза от тенденцията да се намалява количеството на JavaScript, който се изпълнява на клиентската страна. Тя става актуална, когато уеб разработчиците започват да търсят решение за проблема с бавните уеб страници, причинен от големи JavaScript пакети, които натоварват браузърите. Фреймуъркове като Astro.js и Next.js започват да използват тази архитектура, за да отделят интерактивните елементи като независими „острови“, докато останалата част от страницата остава статична.

Тази архитектура води началото си от желанието за създаване на супер бързи уебсайтове, особено за случаи като блогове, новинарски сайтове и маркетингови страници, където основното съдържание е статично, но същевременно има нужда от малки интерактивни компоненти.

Астро е избран за проекта поради неговите ползите и те включват:

* Висока производителност. Генерира статични HTML файлове и зарежда JavaScript само там, където е нужно, което намалява времето за зареждане на страниците.
* Гъвкавост. Поддържа различни фреймуъркове като React, Vue, Svelte и други, което позволява на разработчиците да използват любимите си технологии.
* Малко количество JavaScript. Генерира минимално количество JavaScript код, което води до по-бързо зареждане на страниците.
* Лесно интегриране със CMS. Работи добре с различни системи за управление на съдържанието (CMS) и инструменти за генериране на съдържание.

Но като всеки програмен инструмент Астро си има и недостатъците и те включват:

* По-малка екосистема. Тъй като Astro е сравнително нов фреймуърк, екосистемата му е по-малка в сравнение с утвърдени фреймуъркове като React или Next.js, което може да ограничи наличието на готови решения и поддръжка.
* Липса на универсални функции. Фокусът върху статични страници и минимално използване на JavaScript може да не е подходящ за динамични уеб приложения, които изискват по-тежка клиентска логика.
* По-малка общност. Тъй като фреймуъркът е по-нов, общността около него е по-малка, което може да затрудни намирането на помощ и ресурси в сравнение с по-утвърдени технологии.

### React компоненти

React представлява JavaScript библиотека за изграждане на потребителски интерфейси, която се фокусира върху създаването на интерактивни и динамични уеб приложения. Той се използва основно за създаване на компоненти, които могат да се използват повторно и които управляват състоянието на приложението, без да е необходимо презареждане на страницата. React е създаден от Facebook през 2013 година. Той е разработен, за да реши проблемите, свързани с бързото и ефективно обновяване на потребителските интерфейси в големи уеб приложения. Основната идея зад React е концепцията за "виртуален DOM" (Document Object Model), който прави актуализацията на елементите по-бърза и по-ефективна.

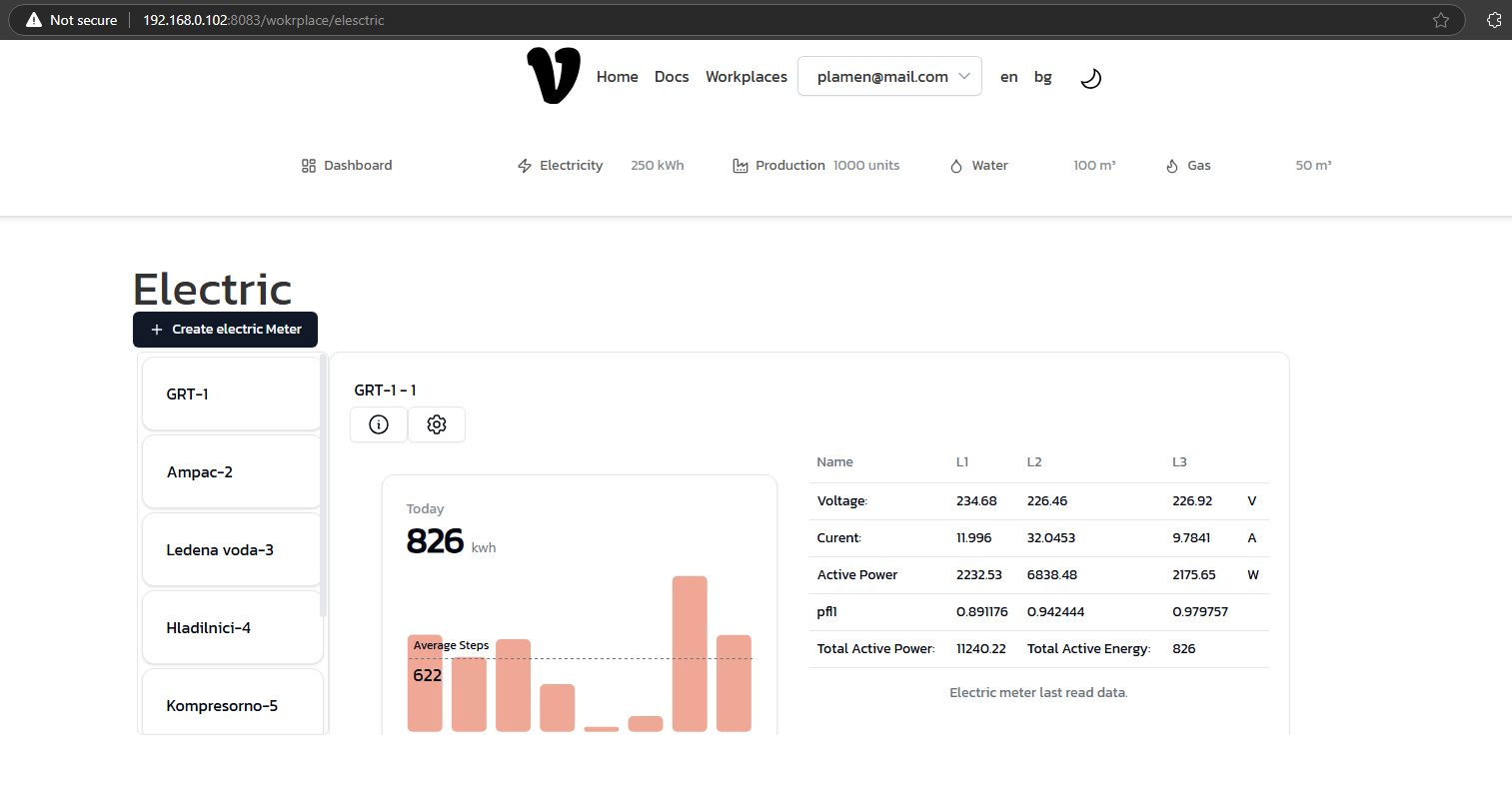
Сред основните плюсове на React са:

* Бързина и ефективност. Благодарение на виртуалния DOM, React оптимизира обновяването на потребителския интерфейс.
* Модулност. Компонентната структура позволява лесно управление и повторна употреба на код.
* Широка екосистема. Съществуват много инструменти и библиотеки, които разширяват функционалностите на React.
* Голямо и активно общество има множество ресурси, документация и готови решения, които помагат на разработчиците.

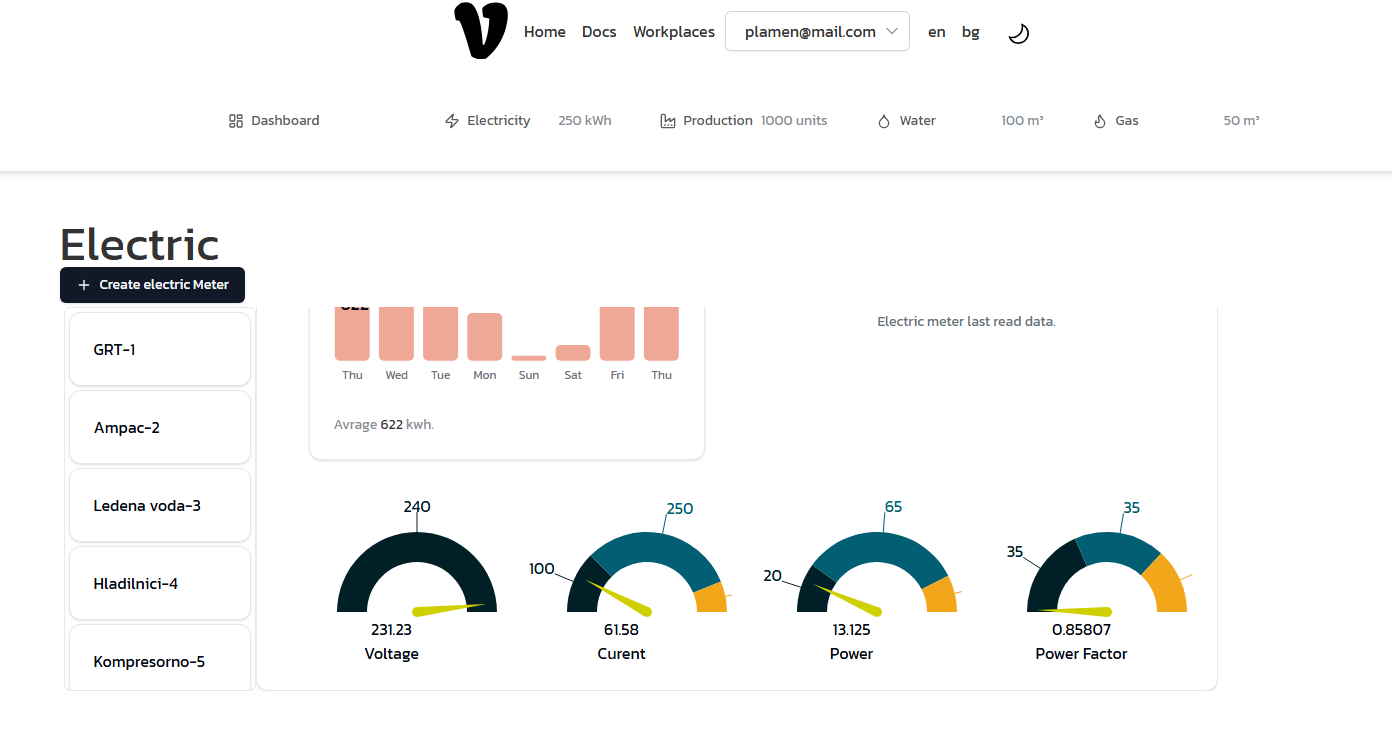
Като минуси може да се посочат:

* Стръмнo обучение– въпреки че React е мощен, неговото използване може да изисква време за свикване, особено с концепции като JSX и управление на състоянието.
* Бързо променяща се екосистема– React и свързаните с него инструменти се обновяват често, което може да доведе до необходимост от постоянно учене и адаптация.
* Не е пълен фреймуърк– React е само библиотека за изграждане на интерфейси, затова често е необходимо да се използват допълнителни инструменти за други аспекти на приложението, като маршрутизация и управление на състоянието.

Реак компонентите използвани за реализация на потребителския интерфейс биват разбити на тяхната конкретна предметна зона. Всяка отделна зона: електричество, вода, газ и производство, е съставяна от визуализация, създаване на нова инстанция, въвеждане на данни и поправяне на инстанция или данни. Тези зони са предвидени така поради естеството на софтуер концентриран върху енергийна ефективност. Започвайки от компонентите за визуализиране на електроенергия се нуждаем да визуализираме първо електромерите и техните показания (Фиг. KOKY). Трябва да се отбележе че всикити данни които са представени на изображенията са реални данни от електромери и са обновяват през един час.

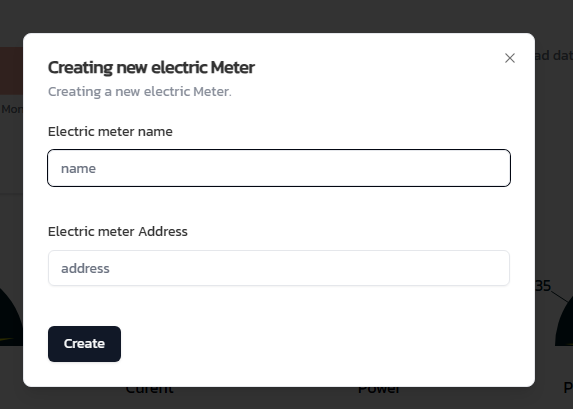


Фиг. KOKY. Визуализация на електромери и техните данни



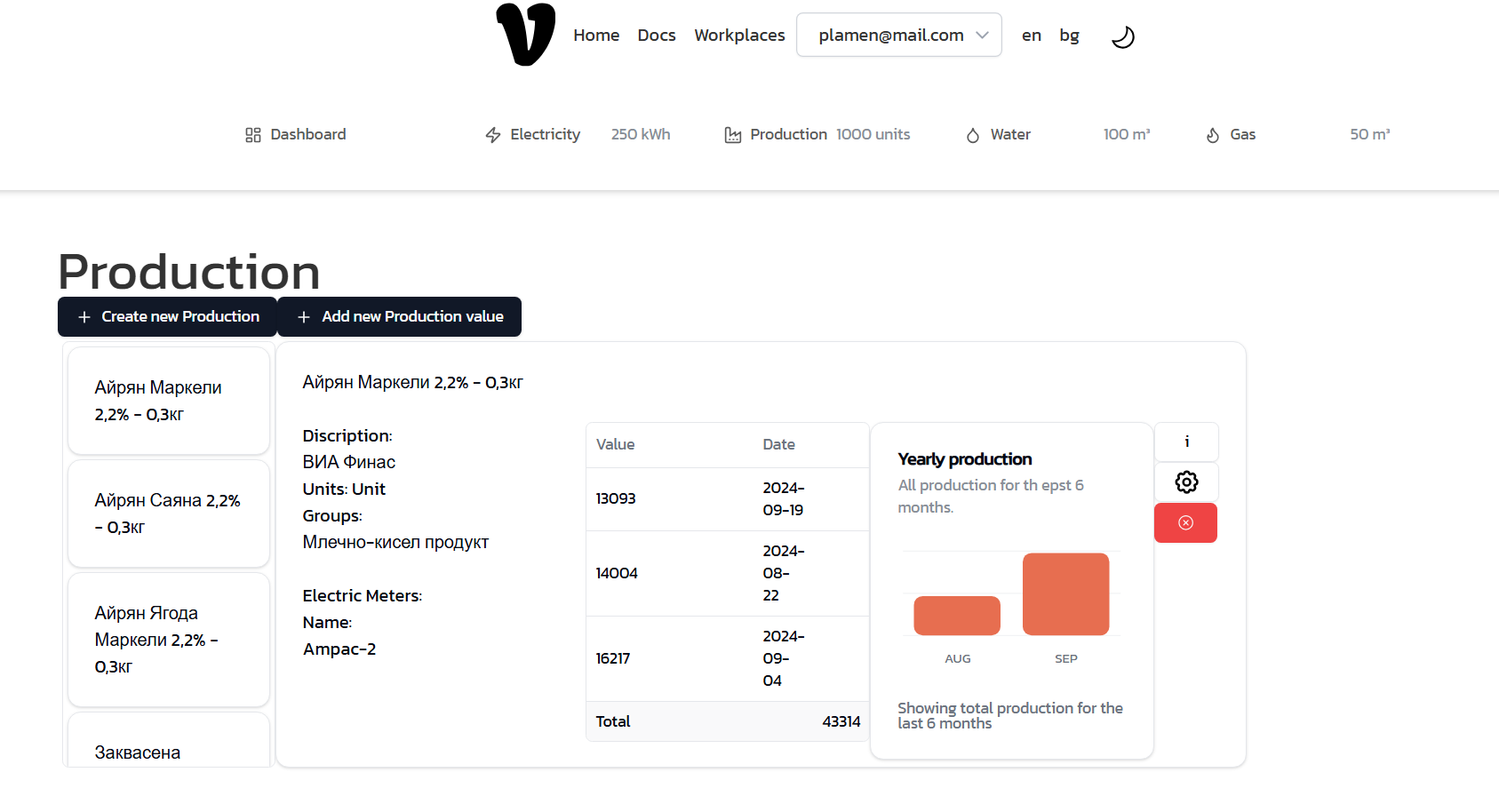
Фиг. KOKY2. Визуализация на електромери и техните данни графично изображение на средните стойности

Всеки отдел си има и за създаване на нова инстанция (Фиг. NNB). Всяко поле за въвежда си има настройка за поразгневяване на допустими стойности на параметъра. Като за целта е използвана библиотеката zod която идва с предефинирани проверки за полета, както и лесно за разбиране шаблони за създаване на свой проверки. Като на пример проверка за емайл е предоставен директно от библиотеката zod и не се нуждаеш да пишеш своя имплементация на вече дефинирани стандарти. Zod е подобна библиотека като стандартната библиотека за проверки на java. Или по потопяваща hibernate-validator за валидации преди създаване на entity обекти в персистентния слой. Системата допържа множествено езици. Това е разгледано в детайли в главата за локализация. Но трябва да се отбележи че се допуска да се въвежда имена на електромери на множествено езици. От които латиница и кирилица са тествани и потвърдени че работят. Както и тестовия клиент работи със системата пряко на Български език и държи на това системата да допържа Български език.

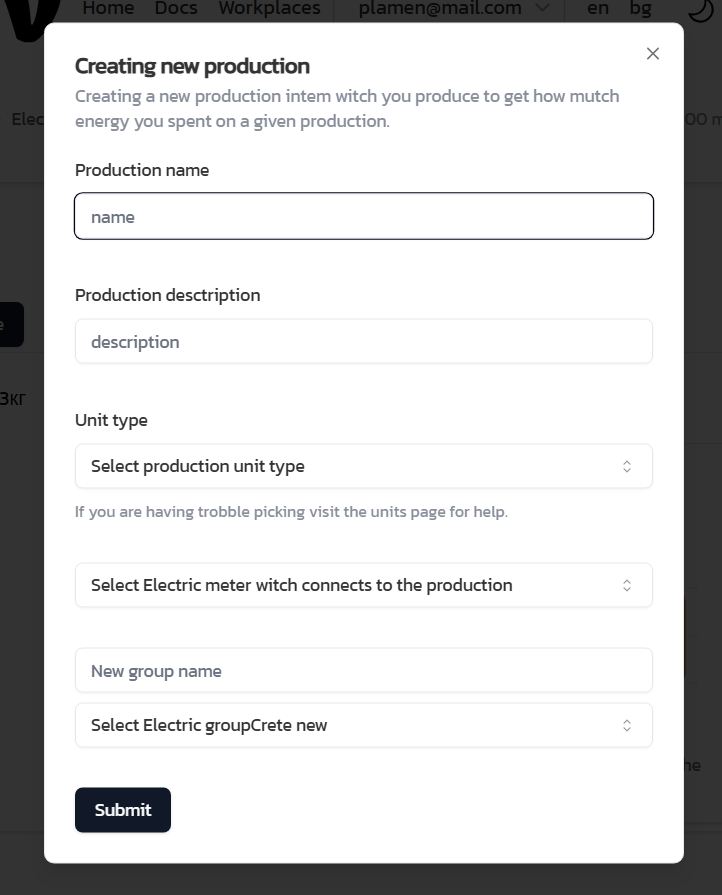


Фиг. №. NNB Формуляр за създаване на нова инстанция на електромер.

Следващи отдел на уеб потребителското приложение е свързано с продукция. Продукцията е нужна като индикатор за да може да се пресметне точно и правилно енергоспестяване на една клиентската система (фиг. № HHP).

Фиг. № HHP. Уеб потребителския интерфейс показващ панела отговорен за продукцията

Както предишното описан формуляр този не е по-различен. Формулата за създаване на производство използва библиотеката zod за валидации( фиг. № JJP.). Но тука позволяваме на крайния потребител да избира набор от групи които желае да впише в съответното производство. Както и мерната единица на съответното производство (Units). Тази мерна единица след това се използва в калкулирането на средния енергиен разход както и за индивидуалния енергиен разход. Тук може и да се добави електромер за който се отнася директно даденото производство. Тъй като в едно предприятие може да има множество на брой машини всяка с нейната си цел, разходи и прочие. Затова е важно да се определи това производство с кои електромери е свързано. Така и само така може да се пресметне конкретната ефективност за производство на продукта. Крайния потребител може и да избера входния електромер на предприятието ако не е създал или свързал конкретен на производствената линия.

Фиг. №. JJP. Формуляр за създаване на нова продукция.

# Източници

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

# Приложение

## Входни точки (End points)

### Сервиз за сигурност (Ink-security)

Вписване като потребител:

Post

/auth/..

Body:

{

‘username’:’user1’,

‘password’:’!pasword12345678’

}

Регистриране като потребител

Вземане на потребителски данни

Пропоръчване