Project Plan
Energy Grid System

Infiniot

Electrify

Date	17/04/2022
Version	1.0
State	WIP
Author	The Development team[1]

[1] The Development team consists of all team members (Aleksandar Popov, Alexander Tsvetkov, Dima Ratuşniuc, Dimitar Ivanov, Georgi Minchev, Kristian Lachev, Velimir Vukašinović)

Version history

Version	Date	Author(s)	Changes	State
1.0	17/04/2022	The Development Team	Creating the document	WIP

Distribution

Version	Date	Receivers
1.0	17/04/2022	Gertjan Schouten, Marcel Boelaars, Tom Meulensteen

Table of Contents

1 Introduction

1.1 Document Purpose

The purpose of this document is to provide a good picture of the Electrify system's architecture and explain the motivation of our choices and the technologies used.

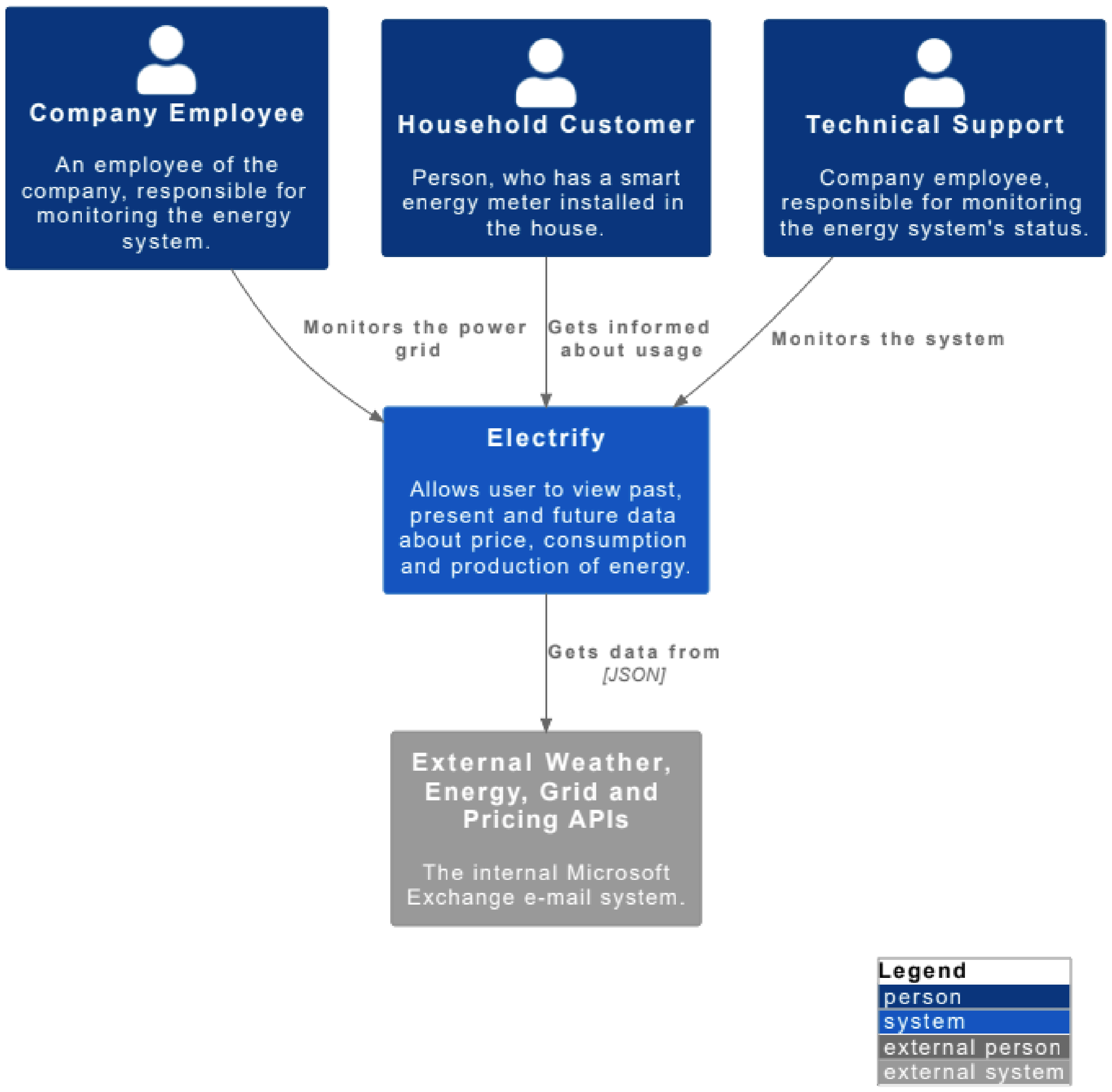
2 Architecture

2.1 C4 Model

The C4 Model architecture diagrams with description can be found on the following link.

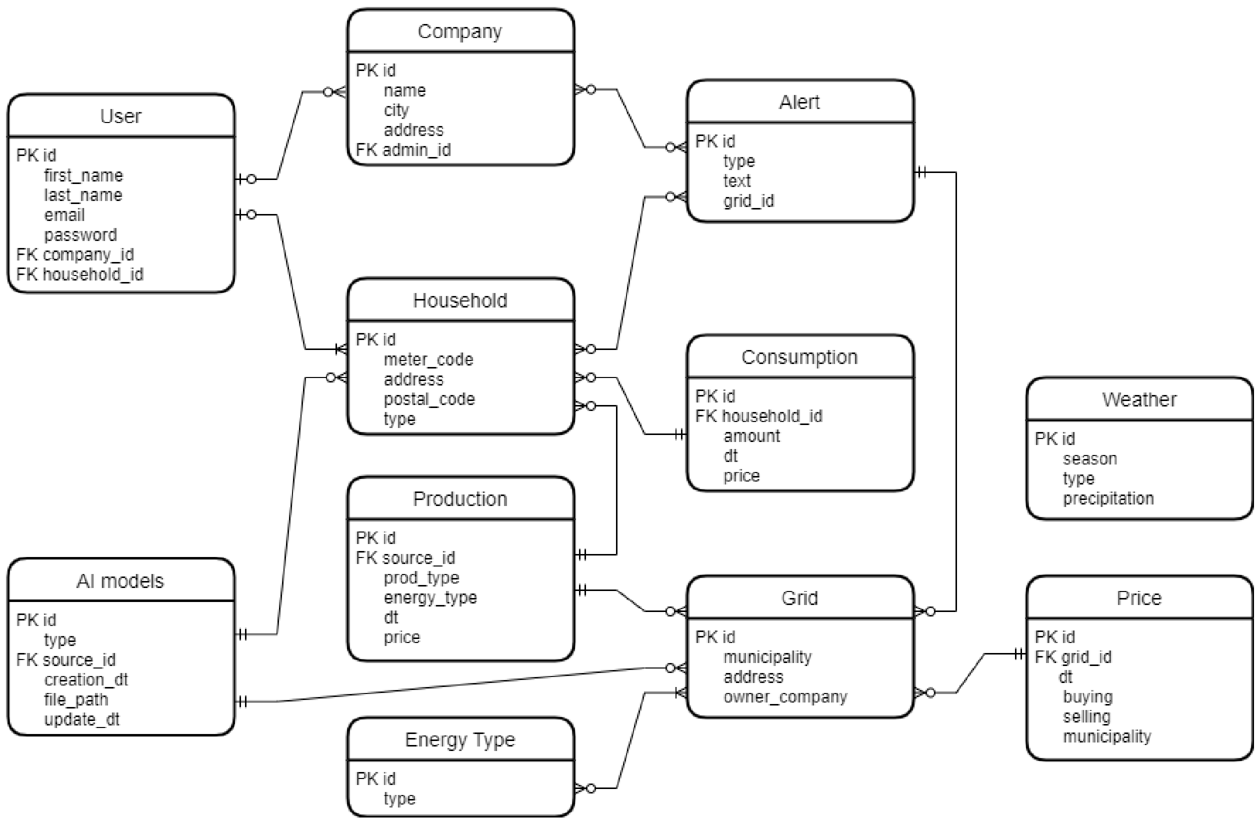
Level 1 System Context Diagram

This layer shows the software as a whole and the people who use it.


Company Employee
An employee of the company, responsible for monitoring the energy system.
Household Customer
Person, who has a smart energy meter installed in the house.
Technical Support
Company employee, responsible for monitoring the energy system's status.
Monitors the power grid
Gets informed about usage
Monitors the system
Electrify
Allows user to view past, present and future data about price, consumption and production of energy.
Gets data from
[JSON]
External Weather, Energy, Grid and Pricing APIs
The internal Microsoft Exchange e-mail system.
Legend
person
system
external person
external system

2.2 Entity Relationship Diagram

This diagram represents the data entities that will be used and describes how they are related to each other in the system. This visual representation will help all the stakeholders gain a better understanding of our system. Details about the related research can be found in the research document in question - What specific energy data do we need to display to each target group?.



Entity relationship diagram

3 Database

3.1 Database choice

The first choice we had to make was whether to use a relational database or a NoSQL database. Each of the options has its benefits and drawbacks and that is why we explored the good and bad practices in the context of our project. Since reading data from our databases is more important, choosing a relational database makes more sense. With a relational database we get more freedom when it comes to querying data while also keeping the read speeds high.

Please find more information about the database server we chose here.

3.2 ORM choice

We decided to use an ORM to access the data in our databases. This would save time from writing plain SQL queries and allow us to focus more time on other aspects of the development process. **Hibernate** is the ORM of choice for Java and Spring Boot. It is performant, lightweight and also keeps track of schema changes and updates the database accordingly. The repository classes Hibernate provides implementations of the most used queries such as CRUD operations, retrieving data by primary key and querying data. Moreover, Hibernate handles the database connection which means that we do not have to manage opening and closing it after a query has been executed. When it comes to security, queries done via Hibernate are SQL-injection proof if done properly according to documentation. To conclude, using an ORM would help us reduce boilerplate code and time spent writing queries while allowing us to focus our time and energy on more important tasks.

4 Continuous Integration and Deployment

For the CI/CD pipeline we make use of Gitlab CI variables. They allow us to keep sensitive information such as usernames and passwords separately and securely from our pipeline.

4.1 CI/CD pipeline backend

Please find the **.gitlab-ci.yml** here.

Explanation:

- **Lines 3-8:** Define the variables that are going to be used in the pipeline.

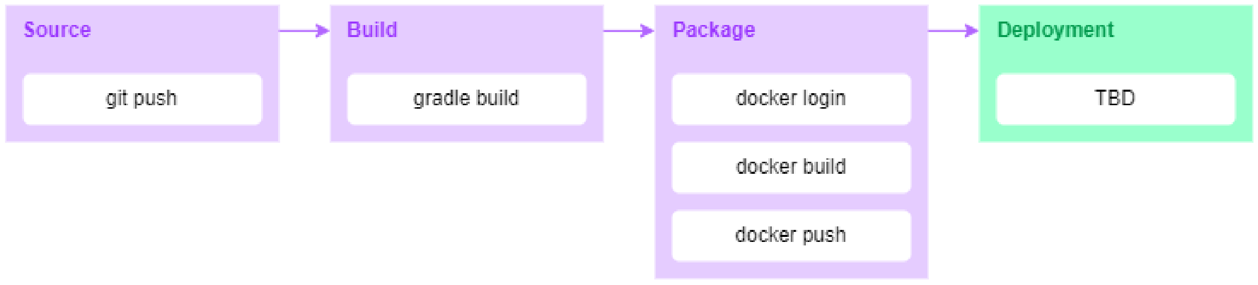
- **Lines 10-12:** Define the stages of the pipeline. There are 2 stages, the build stage and the package stage.

- **Lines 14-24:** The build stage. Builds the system and stores the .jar file for each microservice as artifacts for them to be used in later stages.

- **Lines 26-48:** The package stage. An Image is created for each microservice using the artifact created in the build stage.The image is then pushed to dockerhub.

- **wLine 38:** Logs into Docker Hub.

- **Lines 40-46:** Goes through each microservice, builds its image and pushes it to the correct repository on Docker Hub.



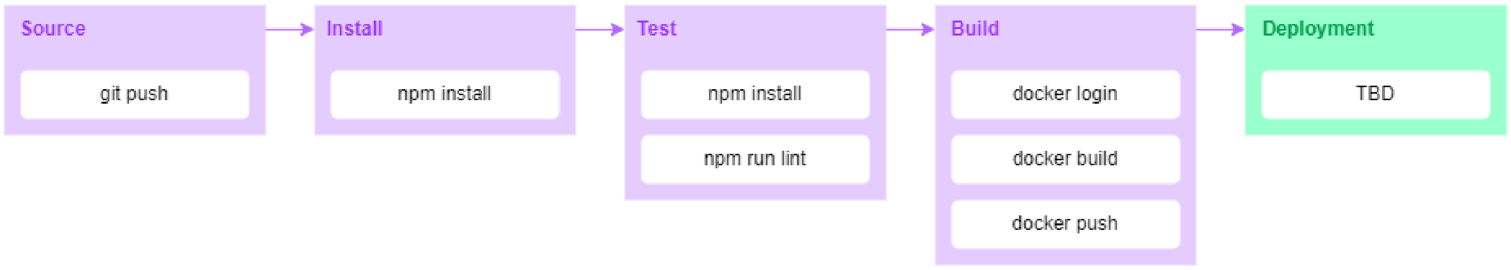
Backend CI/CD pipeline diagram

4.2 CI/CD pipeline web-application

Please find the **.gitlab-ci.yml** here.

Explanation:

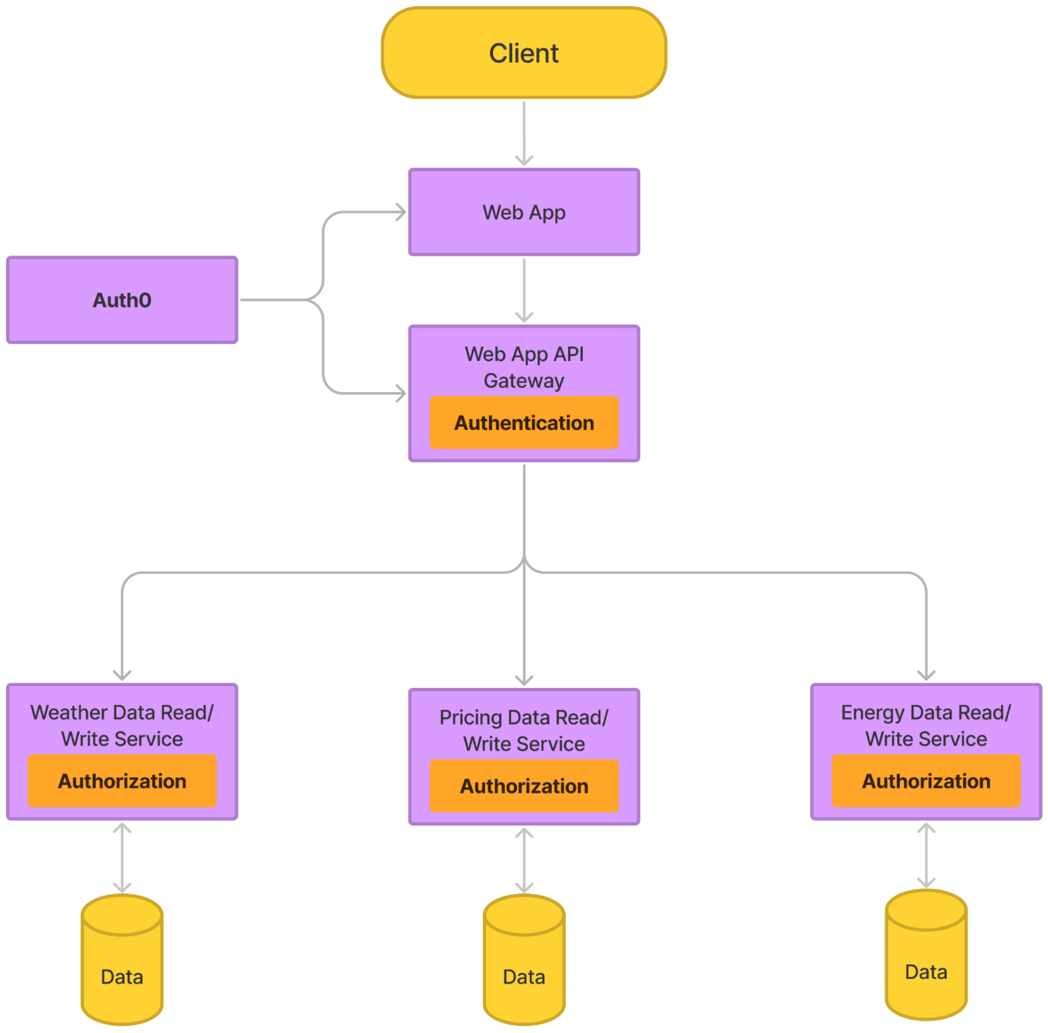
- **Lines 3-4:** Define the variables that are going to be used in the pipeline.
- **Lines 6-9:** Define the stages of the pipeline. There are 3 stages, the install stage, the test stage and the build stage.
- **Lines 11-22:** The install stage. Builds the system and stores the node_modules folder in the cache for reusing it later.
- **Lines 24-38:** The test stage. The code is tested for linting issues.
- **Lines 41-57:** The build stage. Logs into Docker hub, builds the image and pushes it to Docker hub.



Web app CI/CD pipeline diagram

5 System Security

As main security measures we made use of Auth0, which is an external identity server responsible for hosting centralized authorization and authentication. Auth0 uses the industry standard protocols **OAuth2** and **OpenID Connect** for authorization and authentication.
We choose to implement the authentication globally in the API Gateways and authorization per service as a best practice. The image below represents a part of our system which implements the strategy.



Authentication & authorization diagram

For all requests, the API gateway is a single endpoint entry. It gives customers that use these microservices more flexibility by functioning as a central interface. Rather than having access to many services, a client sends a request to one of the **API gateways**, which then routes it to the downstream service. Since the **API Gateways** are a single entry point, it's very convenient to implement authentication concerns in them as this reduces latency and ensures the authentication process is consistent across the application. The security component will enrich the request with the user/security context (identification details for the login user) and route it to a downstream service that will enforce the **authorization check** after successful authentication.
More information about the approach we used and motivation of our choice can be found in the research document in question How to ensure external actors don’t access energy grid information they are not supposed to see?

Authorization flow

Auth0 supports different authorization flows and different scenarios such as server-side, mobile, desktop, client-side, machine-to-machine, and device applications.
Electrify uses the **Authorization Code Flow with Proof Key for Code Exchange (PKCE)**. This is the recommended and most secure flow for our case as we use single page applications for the client side of the system and they come with special challenges which require additional security. The following source explains those challenges and all the steps of the flow - Authorization Code Flow with Proof Key for Code Exchange (PKCE).