Project documentation: Energy Community System – Group E

1. Project summary

Title: Energy Community System
Module: Distributed Systems
University: Fachhochschule Technikum Wien
Semester: SS2025
Link to the GitHub Repository: https://github.com/FHPhilippL/DISYS_GroupE
Team:

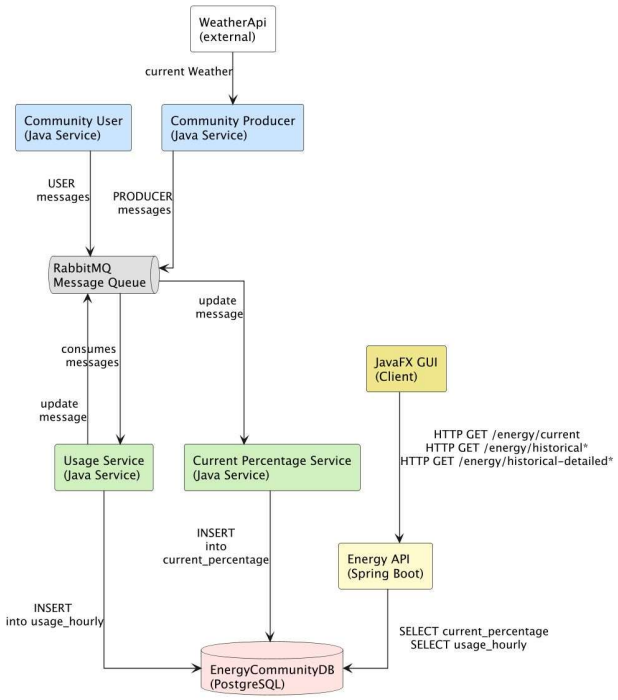
- Philipp Labner (JavaFX GUI & Current Percentage Service)
- Niklas Sterling (Community Energy Producer & User)
- Johannes Voraberger (Rest API, Datenbank & Usage Service)

2. Goals

The project simulates a shared energy-community that produces, consumes and distributes energy. The goal is to process production, consumption, and distribution data using multiple independently running services that interact to form a cohesive distributed system. The data includes production, consumption and distribution, which can be seen through the GUI.

3. System architecture

The system architecture is based on microservices with loosely coupled components that communicate through RabbitMQ. The GUI communicates with the database through the REST API. The producer uses a weather API for realistic production data.



usage_hourly			
hour	timestamp	PK	not null
community_produced	double		not null
community_used	double		not null
grid_used	double		not null

current_percentage			
hour	timestamp	PK	not null
community_depleted	double		not null
grid_portion	double		not null

4. System components in detail

4.1 Community Energy Producer

The Community Energy Producer simulates solar energy generation and sends production messages to RabbitMQ. The amount of energy produced (kWh) is calculated every few seconds based on the current time of day and real-time weather data fetched from OpenWeatherMap.

A custom WeatherAPI class determines a sunlight factor (between 0.2 and 1.0) based on conditions like clear skies, clouds, or rain. In case of clouds the percentage of clouds also affects the sunlight factor. Additionally, information about the sunrise and sunset hours is retrieved to allow dynamic production. The calculateKWh() method combines this factor with a random range. The resulting message includes type, association, kwh, and datetime, and is published in JSON format.

The range for production is between (0.72 kWh/h) and (18.0 kWh/h), averaging to around 3.6 kWh/h which should be realistic for a household-scale solar panel.

4.2 Community Energy User

The community energy user simulates the usage of energy within the community. The usage of energy is calculated, and a message is forwarded to RabbitMQ for further processing. It includes one class.

CommunityUser: This class connects to RabbitMQ and sends a producer message every 5 seconds. The message includes the type “USER”, the association “COMMUNITY”, the consumed kWh and the datetime. The class has one method calculateKWh() that checks the current time. Throughout the day a random kWh value between 0.001 and 0.003 kWh is consumed. The amount is generated randomly with a factor that is decided by the current time of day:

Time	Factor	Average kWh/hour
23:00 – 05:00	Method automatically returns 0.00015 kWh as this is meant to simulate the night	0.3
06:00 – 09:00, 17:00 – 21:00	1 (peak usage)	4.0
12:00 – 16:00	0.3 (low usage)	1.2
05:00 – 06:00, 09:00 – 12:00, 16:00 – 17:00, 21:00 – 23:00	0.6 (moderate usage)	2.4

4.3 Usage Service

The usage service is processing data from the incoming production and consumption messages in the RabbitMQ queue, which it then uses to calculate how much energy is

produced and used by the community, as well as how much of the energy has to be pulled from the grid if the community capacity is not enough.

It subscribes to the energy.input queue, from where it gets the messages sent by the producer/user. It parses the message and extracts the data, which then updates the usage_hourly table in the database. The given datetime from the message specifies which row in the table is updated (one row represents one hour). Depending on the type specified in the message, the corresponding kWh is added to either the community_produced or the community_used or (if the consumed energy exceeds the produced energy) the grid_used column. After the data in the table is updated, a message is sent to the queue energy.updated.

4.4 Current Percentage Service

The current percentage service calculates and updates the energy distribution percentages for the current hour based on the latest usage data. When the usage service sends an update message to the energy.updated queue then this component looks in the usage_ hourly table for the current data and recalculates the current percentages for the hour which is then saved in the current_percentage table.

4.5 REST API

This component is used to provide API endpoints to get data from the database. The /energy/current endpoint is for reading the current percentage from the current_percentage table, whilst the /energy/historical endpoint is for reading from the usage_hourly table (using two specified times). This component is used to fetch the data for the GUI.

4.6 JavaFX GUI

The JavaFX GUI provides a graphical interface for the energy community. The interface shows how much energy is used from the community pool and how much energy is used from the grid (“current” endpoint). In the second section two dates (including day and hour) can be entered to see how much energy has been used in the time between (“historical” endpoint).

As far as the design goes, we have tried to get as close as possible to the template specified in the moodle course. Since the intermediate hand-in this component has been updated to incorporate the received feedback.

5. Technology-Stack

Area	Technology
Language	Java 17 (Java 21 for GUI)
GUI	JavaFX
Backend	Spring Boot
Messaging	RabbitMQ
Database	PostgreSQL
Weather data	External weather API (OpenWeatherMap)
Tests	JUnit 5, Testcontainers
Build-Tool	Maven
Deployment	Docker / docker-compose

6. Tests

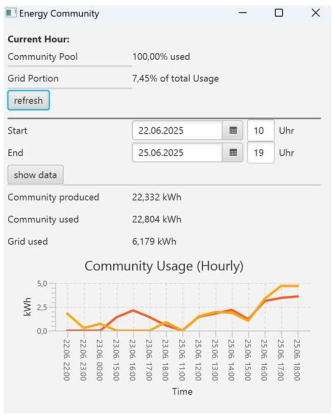
We implemented unit tests for methods such as “calculateKWh()” and integration tests for the message queue.

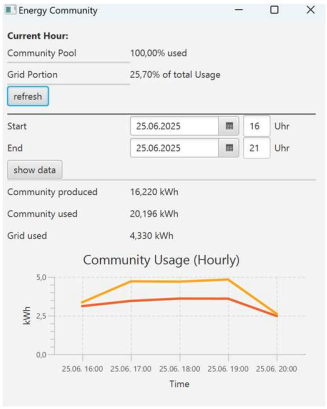
The API, Current Percentage Service and the Usage Service use a docker image for their test, separated from the original Database.

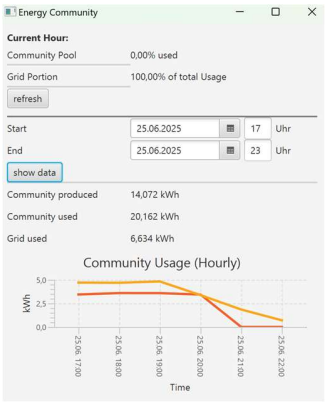
7. Development & Structure

Each component is its own maven module. There is a central pom.xml file so that dependencies can be managed for the whole project. PostgreSQL and RabbitMQ run through a docker-container. The corresponding files for setting them up can be found in the folder EnergyCommunityDB. The weather data gets fetched through the API from OpenWeatherMap to see the current conditions. The github repository was set up through Philipp during class and the components were built separately. Johannes gave the project a good structure by setting up a comprehensive readme and coordinating the team. He also contributed a lot of experience to working with Docker, RabbitMQ and PostgreSQL. Niklas was in charge of implementing the external weather API and choosing how the kWh values were calculated.

8. Screenshots from GUI







(The data in the screenshots is not accurate, as the usage data was set too high originally.)

9. Lessons Learned

- Working with distributed systems needs clear communication between both the team and the various services
- Separate components can be useful as each one is isolated and a change in one component does not affect the others
- We learned about working with various technologies such as RabbitMQ, Springboot, APIs, JavaFX, Postman
- Time management is crucial, especially in the final phases of the project
- The message queue proved useful for decoupling services
- Asking team members for help makes the work a lot easier
- Using logging is more convenient than just writing into the console

10. Time invested (code + tests)

Component	**Responsible**	**Time (in hours)**
Energy Producer	Niklas Sterling	16
Energy User	Niklas Sterling	10
Usage Service	Johannes Voraberger	20
Current Percentage Service	Philipp Labner	15
REST API	Johannes Voraberger	15
JavaFX GUI	Philipp Labner	19
Documentation	Niklas Sterling	6
Communication (Coordination)		15
Code Review		20
Total per Member		**~ 46**