Domäne und Anforderungen - Energiegemeinschaften

Marius Moser, Johannes Fuchs, Konstatin Galvan

Wintersemester 2025

1 Domänenanalyse Energiegemeinschaften

1.1 Überblick über Energiegemeinschaften

Energiegemeinschaften sind ein Teil des österreichischen Erneuerbaren-Ausbau-Gesetzes (EAG). Bürger können mit diesen ihre produzierte Energie an Konsumenten verteilen die sich ebenso in der Gemeinschaft befinden. Hierbei wird die Infrastruktur eines Netzbetreibers verwendet, um die Energie an die Konsumenten zu liefern. Die Mitglieder der Gemeinschaft können dadurch Energie untereinander Handeln. Die Abrechnung wird hierbei von der Gemeinschaft übernommen.

Die Mitglieder einer Gesellschaft lassen sich in folgende Rollen unterteilen.

- **Erzeuger**: Produzieren Energie, z. B. durch Photovoltaik.
- **Verbraucher**: Nutzen die erzeugte Energie.
- **Prosumer**: Kombinieren Erzeugung und Verbrauch.

1.2 Use Case Diagramm

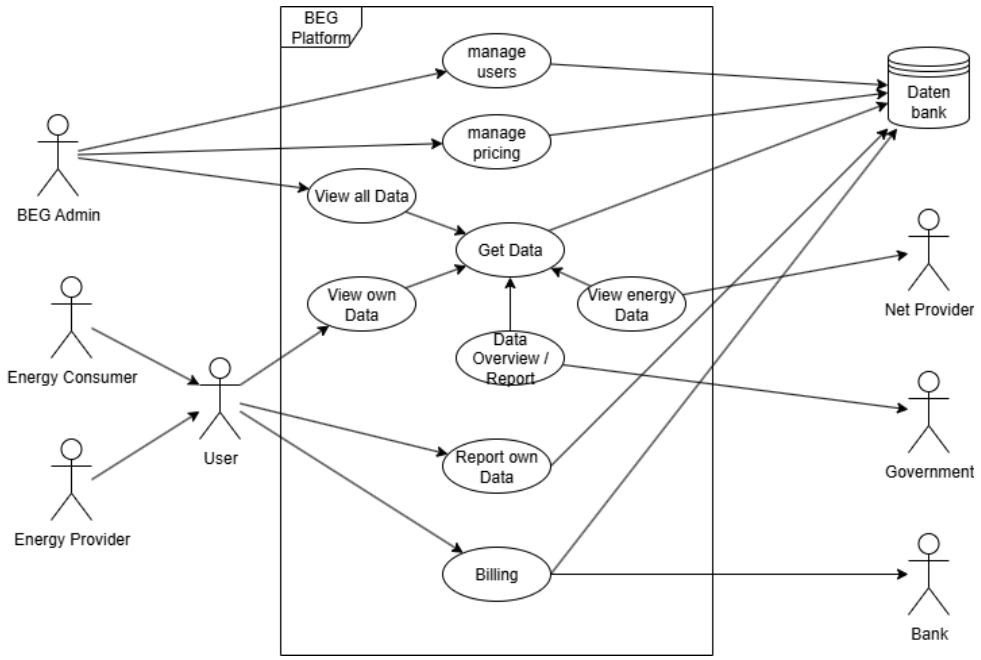

Abbildung 1: Use Case Diagramm für Energiegemeinschaft

Akteur	Use Cases
Energie-Producer	Energie einspeisen in BEG Einspeisemenge überwachen Vergütung erhalten
Energie-Consumer	Energie aus BEG beziehen Verbrauchsdaten einsehen Rechnung erhalten
Energie-Netzbetreiber	Netzstabilität überwachen Einspeise- und Verbrauchsdaten abrufen
Systemadministrator	Nutzer verwalten Datenintegrität sicherstellen Schnittstellen betreuen
Staat / Behörde	Datenzugriff für Monitoring Förderungen verwalten Einhaltung gesetzlicher Vorgaben prüfen

Tabelle 1: Rollen und Aufgaben in einer Bürgerenergiegemeinschaft

1.3 Priorisierte Anforderungsliste

1.3.1 Funktionale Anforderungen

Priorität	Anforderung	Begründung
Hoch	Datenerfassung von Smart Metern	Grundlage für alle weiteren Prozesse wie Abrechnung und Bilanzierung
Hoch	Abrechnungslogik innerhalb der Gemeinschaft	Zentrale Funktion zur finanziellen Transparenz und Fairness
Mittel	Verwaltung von Teilnehmern	Wichtig für Rollenmanagement, aber weniger zeitkritisch
Mittel	Energiebilanzierung	Relevant für Analyse und Optimierung, aber abhängig von Messdaten
Niedrig	Reporting und Visualisierung	Unterstützt Usability und Transparenz, aber nicht kritisch für MVP

Tabelle 2: Funktionale Anforderungen

1.3.2 Nicht-funktionale Anforderungen

Priorität	Anforderung	Begründung
Hoch	Datenschutz und Datensicherheit	Gesetzlich verpflichtend (DSGVO), besonders bei personenbezogenen Daten
Hoch	Verfügbarkeit und Ausfallsicherheit	System muss zuverlässig laufen, besonders bei Echtzeitdaten
Mittel	Skalierbarkeit	Wichtig für zukünftiges Wachstum, aber initial begrenzt relevant
Mittel	Performance (Latenz, Durchsatz)	Relevant für Nutzererlebnis, aber abhängig vom konkreten Use Case
Niedrig	Interoperabilität mit bestehenden Systemen	Nice-to-have, aber nicht zwingend für initiale Version

Tabelle 3: Nicht-funktionale Anforderungen

2 Softwarearchitektur

2.1 Service-Identifikation

Basierend auf Domain-Driven Design Prinzipien wurden folgende Microservices identifiziert:

- **Community Management Service**: Verwaltung von Teilnehmern, Rollen und Beziehungen.
- **Billing Service**: Abrechnung, Fakturierung und Zahlungsstatus.
- **Metering Service**: Erfassung und Aggregation von Smart Meter-Daten.
- **Analysis Service**: Visualisierung, Analyse und Export von Energiedaten.
- **Core Service**: koordiniert Services

Jeder Service besitzt eine eigene Datenbankinstanz (MongoDB) und kommuniziert über RabbitMQ (asynchron) sowie REST (synchron).

2.2 Architekturdiagramme

System Context Diagram (C4 Level 1)

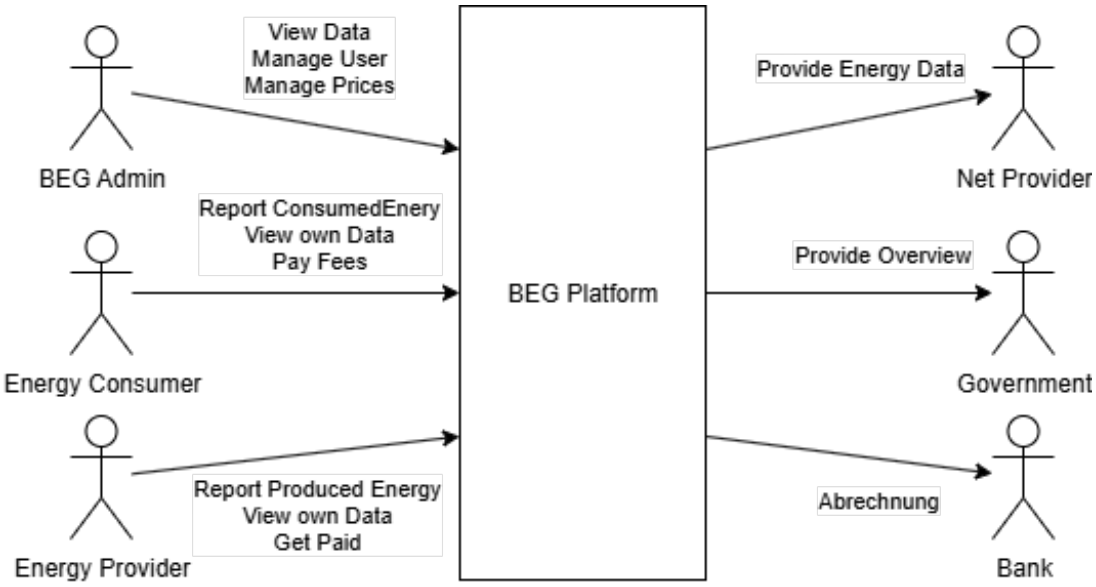

Abbildung 2: Systemkontextdiagramm der Energiegemeinschafts-Plattform

Sequenzdiagramm: Energieabrechnung

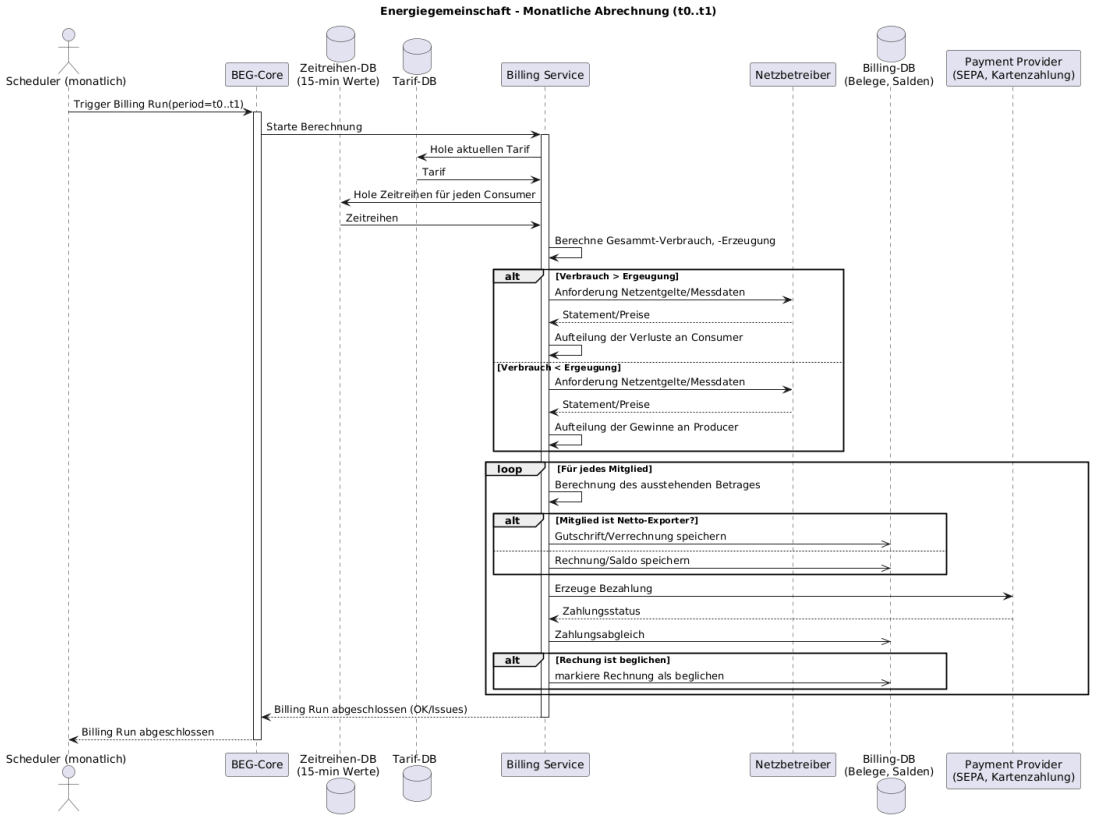

Abbildung 3: Sequenzdiagramm für Energieabrechnung

2.3 Kommunikationsdesign

Synchrone Kommunikation

- REST APIs sowie für Benutzerinteraktion und Admin-Funktionen als auch für interne Service-Kommunikation

Asynchrone Kommunikation über RabbitMQ

- **Exchange Types**: Topic Exchange für flexible Routing-Strategien
- **Queue Design**: Dedizierte Queues pro Service, z. B. `billing.queue`, `metering.queue`
- **Message Schemas**: JSON-basierte Payloads mit Validierung via JSON Schema

2.4 Technologieentscheidungen

Komponente	**Technologie**	**Begründung**
Programmiersprache	C# mit .NET 9	Moderne Sprache mit guter Microservice-Unterstützung und Performance
Datenbank	MongoDB	Schema-flexibel, ideal für heterogene Energiedaten
Message Broker	RabbitMQ	Etabliert, unterstützt Topic Exchange und hohe Zuverlässigkeit
Containerisierung	Docker	Standard für portable Deployments
Orchestrierung	k3s mit k3d	Lightweight Kubernetes für lokale Entwicklung
API-Kommunikation	REST	REST für externe Clients und interne Services

Tabelle 4: Technologieentscheidungen und Begründungen

3 Setup-Dokumentation

3.1 Entwicklungsumgebung

Für die lokale Entwicklungsumgebung wurden folgende Tools installiert und konfiguriert:

- Docker Desktop (Version 24.) mit WSL2-Integration (Ubuntu)
- .NET 9 SDK
- kubectl, helm, Git
- JetBrains Rider (IDE)
- MongoDB Compass (DataBase Viewer)
- Postman (API Network)
- Lens IDE (Cybernetes Cluster)

3.2 k3s Cluster Setup mit k3d

Ein k3s Cluster wurde erfolgreich mit k3d erstellt:

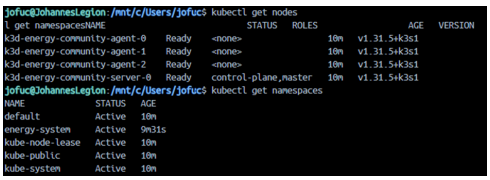

Abbildung 4: Screenshot der laufenden k3s Nodes

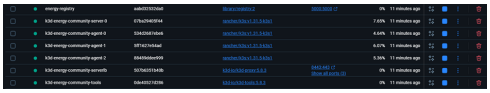

Abbildung 5: Screenshot der laufenden Services

3.3 Probleme und Lösungen

- **RabbitMQ und MongoDB Installation fehlgeschlagen** Docker Login auch in der WSL nötig gewesen.
- **Port Forwarding hat nicht funktioniert** In 2h nochmal probiert.
- **Kubernetes wurden von Lens nicht erkannt:** Herauskopieren der konfiguration von innerhalb von WSL.
- **MongoDB nicht verbindbar von außerhalb des Containers:** Verwendung von root und secretpass.
- **Veraltete WSL Version + Fehler in der Deinstallation:** Wiederholtes updaten.

- **Cluster-Commands darf man nicht aus adobe kopieren:** Datei in SumatraPDF öffnen und von dort kopieren.
- **:**