WDB Mini Challenge 1

28.04.2022

Etienne Roulet

Florin Barbisch

Yvo Keller

GitHub Repository:

https://github.com/Smart-Classroom-Challenge/SmartClassRoom-Backend

Übersicht und Hauptidee

Für die Smart Classroom Challenge wird eine API benötigt, die es IoT Geräten ermöglicht, Sensordaten zur Luftqualität kontinuierlich im richtigen Kontext (Klassenzimmer, Messgerät) zu persistieren. Für die API wurde Python Django evaluiert, das mit seinem eigenen ORM die Persistierung vereinfacht und es dem Entwickler ermöglicht in seinem OOP Spektrum zu bleiben. Django allein bietet nur eine API für Server Side Templating. Deshalb haben wir uns zusätzlich für das «Django REST framework» entschieden, um REST-Schnittstellen mit CRUD Support für die relevanten Entitäten zu bauen. TimescaleDB ist unsere Persistenz-Schicht.

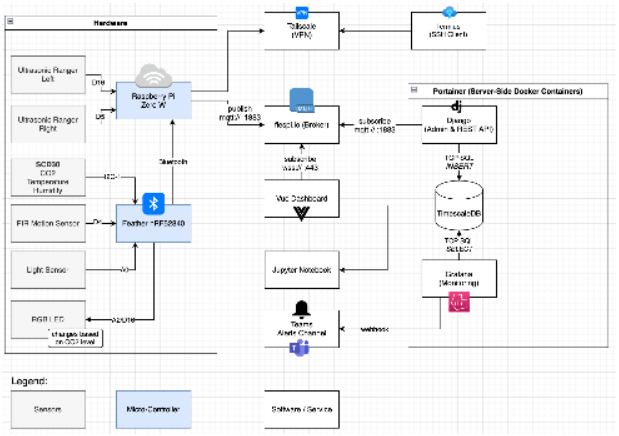
Evaluation: RESTful vs. GraphQL vs. OData

Wir haben uns für REST entschieden, da für uns Error Handling und einfache Filterung das Wichtigste ist. Eine komplexe Query Language für Schemata benötigen wir nicht. Würde das Projekt wachsen, könnte man es in Betracht ziehen. Dennoch ist mit der TimescaleDB und den Timescale Models in Django die Implementierung wesentlich einfacher im REST Umfeld, als zusätzliche Controller zu schreiben für GraphQL. Das grösste Argument für den Einsatz von GraphQL ist, dass bei Queries für das jeweilige Feature redundante und nicht gebrauchte Daten (overfetching, underfetching) in der Response nicht mitgeliefert werden. Unsere Applikation für den Smart Classroom hat jedoch keine Redundanzen, und das ORM Model von Django reicht völlig aus für die MQTT Schnittstelle. Da wir auch keine breite Serverlandschaft haben, mit mehreren Microservices und nur einem Monolith System (Django) mit Anbindung an den MQTT Broker und an die TimescaleDB Datenbank, wird REST empfohlen. Ausserdem ist der Mehraufwand Mutations in GraphQL zu schreiben den Ertrag nicht wert. Und VueJS müsste mit Apollo angebunden werden. Apollo ist eine weitere Library und was Performance betrifft ist in JavaScript weniger mehr, denn seit die async Fetch API voll funktionsfähig im Browser implementiert ist (ausser IE11), ist alles andere nur zusätzlicher Ballast und redundant. Einen Nachteil hat es, Pagination wäre sehr einfach in GraphQL umsetzbar, was in REST mit Clients etwas schwieriger ist. Ausserdem haben wir kein overfetching und nur wenige underfetchings. OData wäre da wahrscheinlich das Beste, jedoch gibt es keine Django Implementation von OData. Für die Evaluierung wurde GraphQL dem Branch graphql_implementation implementiert. **Siehe GitHub graphql_implentation Readme**.

Beispiel:

REST: GET / PUT / POST / DELTE	**GraphQL: Query**	**OData: Query**
https://service.com/person/1	{ person(id:"1") { firstName lastName } }	serviceRoot/person(id)/Name

Data Science Aspekte

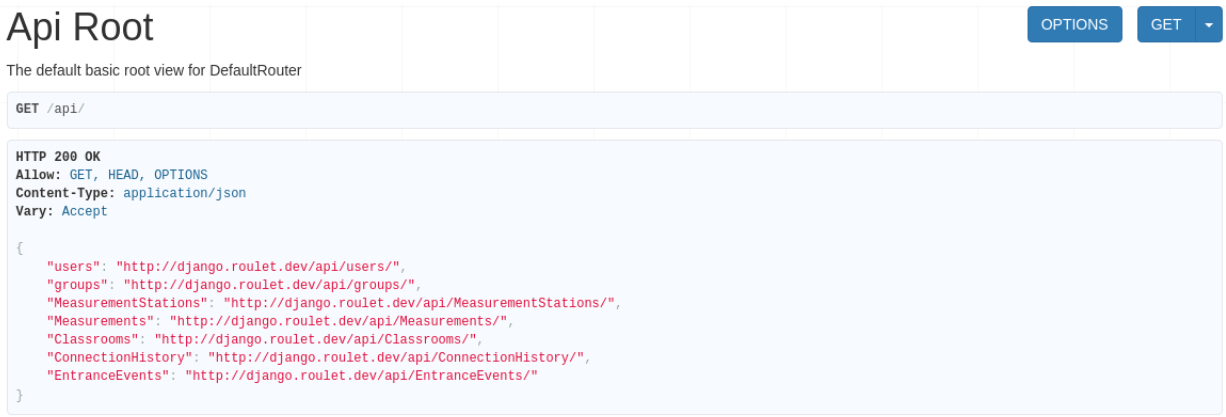


Als Data Science Aspekt haben wir die Kommunikation mit IoT Grove Sensoren, angebunden an ein Feather nRF52840, welcher über Bluetooth Daten an einen Raspberry Pi sendet. Die komplette Systemlandschaft sieht wie folgt aus:

Die Daten können verwendet werden, um ein Dashboard zur Überwachung der Daten zu erstellen oder den Zusammenhang zwischen CO2 und Anzahl Personen herauszufinden. Dies ist Teil der Smart-Classroom Challenge.

Spezifikationen

Die Schnittstelle ist mit **Django** und der «**Django REST framework**» Library implementiert.



Api Root

OPTIONS GET

The default basic root view for DefaultRouter

```
GET /api/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "users": "http://django.roulet.dev/api/users/",
    "groups": "http://django.roulet.dev/api/groups/",
    "MeasurementStations": "http://django.roulet.dev/api/MeasurementStations/",
    "Measurements": "http://django.roulet.dev/api/Measurements/",
    "Classrooms": "http://django.roulet.dev/api/Classrooms/",
    "ConnectionHistory": "http://django.roulet.dev/api/ConnectionHistory/",
    "EntranceEvents": "http://django.roulet.dev/api/EntranceEvents/"
}
```

Authentifizierung

Als Authentifizierung wurde ein JWT Token basierende Anmeldung verwendet. Es wurde bewusst bcrypt als Algorithmus verwendet, denn Argon2 ist sehr rechenintensiv und wird meist auf einer GPU ausgeführt.

Autorisierung

Jeder der einen validen JWT Token hat, ist autorisiert. Es wurde kein spezielles Rollen/Berechtigungs-System definiert und implementiert.

Relevanz für Anwendungsfälle in der echten Welt

Unsere API hat einen direkten Bezug zur echten Welt: sie ermöglicht es, Daten zur Luftqualität zu persistieren und später wieder abzurufen. Die Daten können dann in einem Web-Client dargestellt werden. Im nachfolgenden Bild ist Grafana mit einer solchen möglichen Visualisierung zu sehen (spezifisch Grafana greift direkt auf die DB zu und nicht über die REST API). Dennoch könnten wir jetzt ein solches Dashboard entwickeln, welches die Daten über die REST API lädt. Dies liegt jedoch ausserhalb des Umfangs dieser Mini-Challenge.

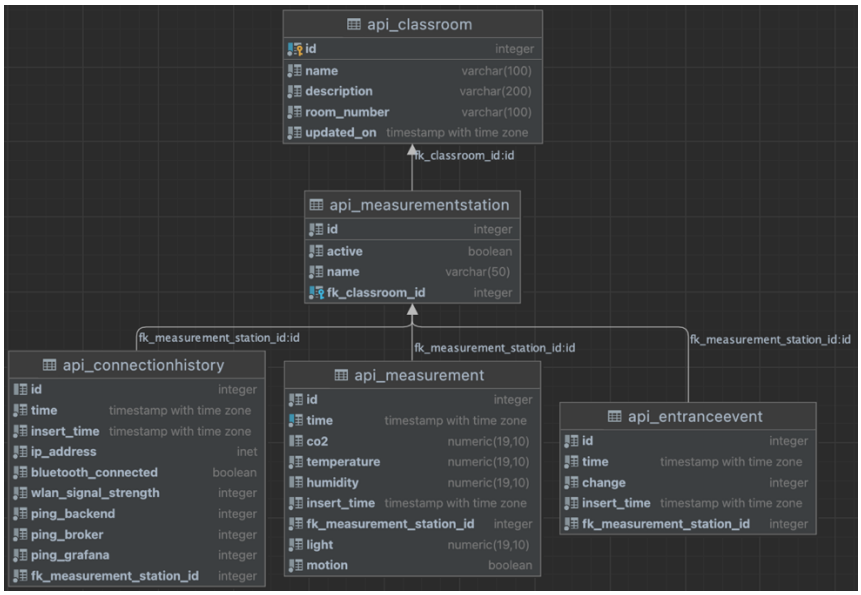
Resultate

Resultat ist eine **REST Schnittstelle** mit durchgehendem **CRUD Support**, die alle Daten in der TimescaleDB Datenbank persistiert. Informationen zum Aufsetzen und Nutzen der REST API können dem README in unserem GitHub Repository entnommen werden.

Die **Unit Tests** befinden sich unter SmartClassRoom/api/tests.py.

Die **API-Dokumentation** ist hier zu finden, Anmeldung erfolgt mit den Zugangsdaten im README auf dem GitHub Repository.

Nachfolgende **Entitäten** können über die entwickelte REST Schnittstelle verändert werden:



Anhang

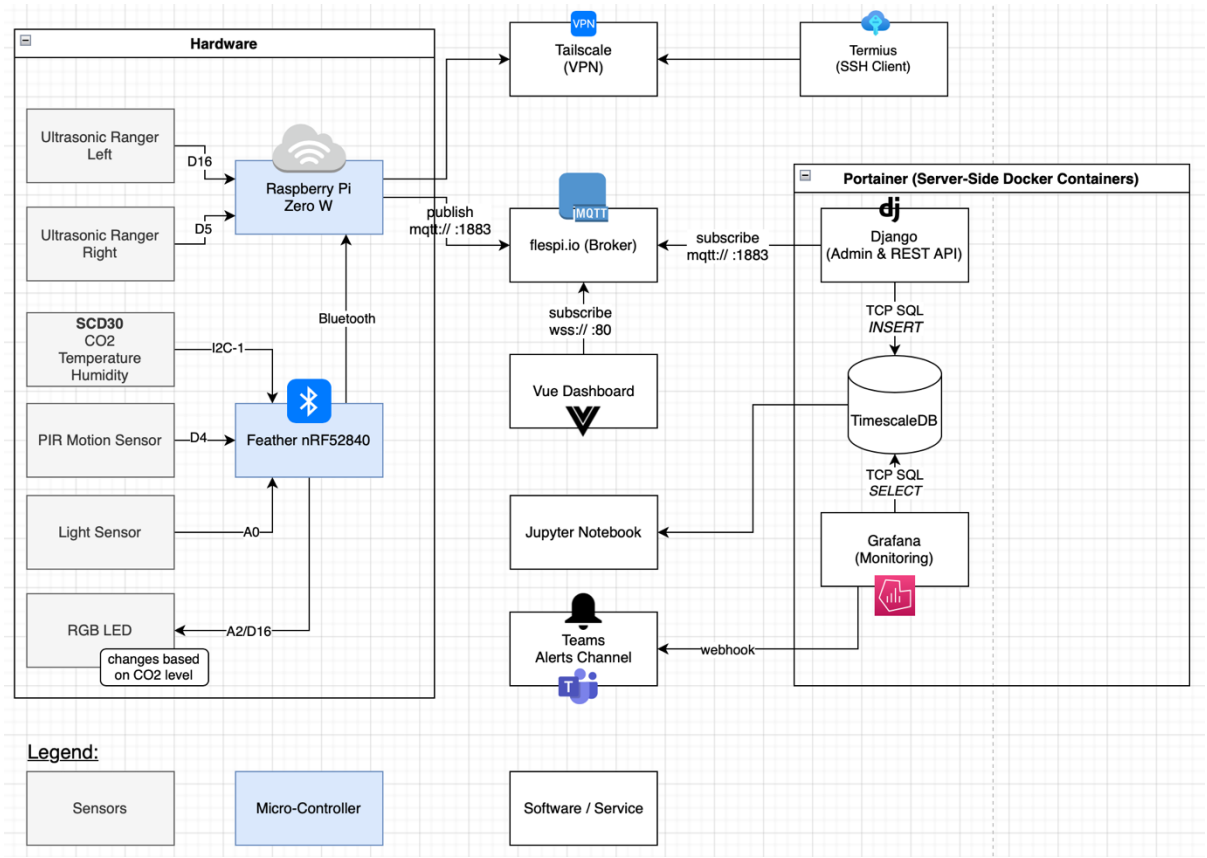


Figure 1 Unsere komplette Systemarchitektur, von der die REST API ein zentraler Teil ist

GraphQL vs OData vs REST

	GraphQL	OData	REST
Architecture	Client-driven	Client/Server-driven	Server-driven
Organized in terms of	Schema & Type system	Endpoints	Endpoints
Operations	Query Mutation Subscription	Get Post Put Delete	Create Read Update Delete
Data fetching	Specific data with a single API call	Fixed data with multiple API calls	Fixed data with multiple API calls
Community	Growing	Large	Large
Performance	Growing	Large	Large
Self-documenting	✓	⊖	⊖
File uploading	⊖	✓	✓
Web caching	✓	✓	⊖
Stability	Less error prone: automatic validation and type checking	Better choice for complex queries	Good for complex queries

Figure 2 GraphQL, OData und REST im Vergleich