Universitatea Națională de Știință și Tehnologie POLITEHNICA București
Facultatea de Electronică, Telecomunicații și Tehnologia informației

PROIECT

REST API service study and development for virtual eCare applications

Supervisor: Assoc. Prof. Alexandru Vulpe alexandru.vulpe@upb.ro

Studenți: Lupu Silviu-George

Grupa:AWC

1. REST API

REST stands for *Representational State Transfer* and is a software architecture style that defines a pattern for client–server communications over a network. REST provides a set of constraints that promote performance, scalability, simplicity, and reliability.

1.1 REST architectural constraints

1. **Stateless**: The server does not maintain state between requests.
2. **Client–server**: Client and server are decoupled and can evolve independently.
3. **Cacheable**: Responses can be cached by client or server.
4. **Uniform interface**: Resources are accessed in a uniform way, regardless of representation.
5. **Layered system**: Clients may connect through proxies, gateways, or load balancers.
6. **Code on demand (optional)**: The server may transfer code (e.g., JavaScript) to the client.

1.2 HTTP Methods

REST APIs listen for HTTP methods that define actions on resources:

Method	**Description**
GET	Retrieve an existing resource.
POST	Create a new resource.
PUT	Update an existing resource.
PATCH	Partially update a resource.
DELETE	Delete a resource.

1.3 HTTP Status Codes

When an API processes a request, it returns a status code to inform the client:

Code	Meaning	Description
200	OK	The request was successful.
201	Created	A new resource was created.
202	Accepted	The request was received but not yet processed.
204	No Content	Success, but no body in the response.
400	Bad Request	The request was malformed.
401	Unauthorized	Client not authorized.
404	Not Found	Resource was not found.
415	Unsupported Media Type	The request format is not supported.
422	Unprocessable Entity	Data was valid JSON but invalid/missing fields.
500	Internal Server Error	Generic server error.

1.4 Using Python

One method to use REST APIs in Python is with the requests library, which allows me to send HTTP/HTTPS requests easily.

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/todos/1")

if response.status_code == 200:

   print(response.json())
```

In my implementation I later used **httpx**, because it integrates better with FastAPI and supports async calls.

2. Background

vCare-as-a-Service is a REST API designed to integrate third-party health and rehabilitation platforms with clinical pathways. Normally, third-party systems provide services independently but are not linked to structured medical protocols. Through vCare-as-a-Service, a patient using a third-party platform can subscribe to a recommended clinical pathway.

3. Objectives

My objectives were:

1. Study how an API can be developed without knowing the full set of endpoints.
2. Develop a basic API that polls a service bus, retrieves a list of service IDs, and consumes the data from each service endpoint.
3. Test the basic API.
4. Present my findings in this report.

Assessment: API study (60%), development and testing (40%).

4. API Discovery

When I develop an API client without knowing the full set of endpoints in advance, I need a discovery mechanism that allows the client to adapt dynamically. I studied three possible approaches:

1. **Capabilities / HATEOAS** – the service provides a stable endpoint (e.g., /services/{id}/capabilities) that lists the available operations and their URIs. By following these links, the client can dynamically discover what actions are supported. This method is simple and allows the client to adapt when new endpoints are added.
2. **OpenAPI / Swagger Introspection** – some services expose an OpenAPI specification (commonly at /openapi.json). By parsing this specification, the client can identify all

endpoints, methods, and parameters, and even generate code automatically. This gives complete visibility, but it requires the provider to publish and maintain the specification.

3. **Centralized Service Registry** – in this approach, the client first queries a central registry that contains the list of all services and their endpoints. After retrieving this information, the client can connect to the correct service. This option simplifies discovery, but it creates a single point of failure and adds another dependency.

My choice:
For my implementation, I chose the **capabilities endpoint** approach. I find it the most straightforward, standardized, and extensible solution, because it allows me to automatically adapt to changes without relying on an external registry or a full OpenAPI specification.

---

5. Implementation

I built two FastAPI servers that talk over HTTP on **different ports** (two processes cannot bind the same port):

5.1 Mock Service Bus (port 8001)

- **Purpose:** publish available services and their **capabilities** and simulate data endpoints.
- **Key routes** (from mock_bus.py):
  - GET /bus/services → returns service IDs (e.g., svc-a, svc-b).
  - GET /services/svc-a/capabilities → returns endpoints: [{"rel":"telemetry","href":"/services/svc-a/telemetry"}].
  - GET /services/svc-b/capabilities → returns endpoints: [{"rel":"alerts","href":"/services/svc-b/alerts"}].
  - GET /services/svc-a/telemetry → sample data (timestamp/value).
  - GET /services/svc-b/alerts → sample alert.

It runs with Uvicorn:

```
uvicorn mock_bus:app --host 127.0.0.1 --port 8001 --reload
```

5.2 Consumer API (port 8000)

- **Purpose:** periodically **poll** the bus, discover endpoints dynamically, fetch data, normalize it, and expose it to callers.
- **Key elements** (from main.py):
  - **Config constants:**
    BUS_BASE = "http://127.0.0.1:8001" (bus address)
    POLL_INTERVAL_SEC = 5 (poll period)
    HTTP_TIMEOUT = 5.0 (HTTP timeout)
  - **Async HTTP client:** httpx.AsyncClient for non-blocking calls.
  - **Polling flow:**
    1. GET /bus/services → list service IDs;
    2. for each ID: GET /services/{id}/capabilities → get endpoints;
    3. for each endpoint href: GET the data;
    4. **normalize** into a common schema {service_id, kind, t, v};
    5. store **in memory**.
  - **Why async:** HTTP calls are I/O-bound; asyncio keeps the API responsive while waiting for network responses and scales better if services increase.
  - **Consumer routes:** typically expose GET /health, GET /data (aggregated results), and POST /admin/clear (reset in-memory storage).

It runs with Uvicorn:

```
uvicorn main:app --host 127.0.0.1 --port 8000 --reload
```

5.3 Data normalization

Incoming payloads can differ (list vs object; different fields). I map them to a **minimal uniform structure**:

```
{
  "service_id": "<svc-id>",

  "kind": "<capability/rel e.g. telemetry|alerts>",

  "t": "<timestamp or None>",

  "v": "<value or whole object>"

}
```

This keeps the consumer generic and decoupled from service-specific schemas.

6. Technologies and Tools Used

- Python 3 – programming language used for both servers.
- FastAPI – web framework to define REST endpoints for mock bus and consumer.
- Uvicorn – ASGI server to run FastAPI apps (--reload during development).
- httpx – asynchronous HTTP client used by the consumer to poll the bus.
- asyncio – Python's async runtime used to run the background polling loop.
- typing – type hints (List, Dict, Any, Optional) used for readability and better editor support.
- curl – manual testing from the terminal (e.g., curl http://127.0.0.1:8000/data).
- venv – virtual environment to isolate project dependencies.
- requirements.txt – reproducible dependency list (e.g., fastapi, uvicorn[standard], httpx).
- Git – version control; .gitignore excludes .venv/, __pycache__/, etc.

Note: There are no automated tests (pytest/respx) in the current codebase. Testing was performed manually with curl and with the interactive Swagger UI at /docs.

curl – used to manually test endpoints from the terminal (GET /data, POST /admin/clear).

7. Setup & Run

Create environment & install deps

```
python -m venv .venv
# Windows: .venv\Scripts\activate
# Linux/macOS:
source .venv/bin/activate
```

pip install -r requirements.txt

Run the mock bus (port 8001)

```
uvicorn mock_bus:app --host 127.0.0.1 --port 8001 –reload
```

Run the consumer (port 8000)

```
uvicorn main:app --host 127.0.0.1 --port 8000 –reload
```

Manual checks

```
curl http://127.0.0.1:8001/bus/services        # bus answers
curl http://127.0.0.1:8000/health              # consumer up
sleep 6 && curl http://127.0.0.1:8000/data     # aggregated data after a poll
```

---

8. Conclusions

Through this project I learned how to:

- Build a consumer API that adapts to unknown endpoints using the **capabilities approach** (HATEOAS).
- Implement **polling, normalization, and endpoint discovery** in Python.
- Work hands-on with **FastAPI** for building APIs, **Uvicorn** as the ASGI server, and **httpx** for asynchronous HTTP calls.

- Use **asyncio** to keep the consumer responsive while polling in the background.
- Organize a project properly with a **requirements document**, **requirements.txt**, and a **virtual environment** (venv) for reproducibility.
- Test endpoints manually using **curl** and the built-in **Swagger UI** at /docs.
- Understand better the difference between **hardware APIs** (direct function calls via drivers) and **web APIs** (endpoint-based communication over HTTP).

Future improvements

- Replace **in-memory storage** with persistent storage (SQLite for local demos, PostgreSQL for production).
- Add **retry/backoff logic** for unstable networks using libraries like tenacity.
- Add a /metrics endpoint and structured logging for **monitoring and observability**.
- Secure the consumer API with **authentication** (API keys or OAuth2).
- Support **configuration via environment variables** (e.g., bus URL, poll interval).
- Replace polling with **event-driven integration** (webhooks, Kafka, or a message bus).
- Write **automated integration tests** in the future to replace purely manual testing.
- Extend the mock bus with more dynamic data and multiple service types to simulate real-world heterogeneity.