API Study and Client Development for vCare-as-a-Service

Lupu Silviu-George
National University of Science and Technology POLITEHNICA Bucharest
silviu_george.lupu@stud.etti.upb.ro
Supervisor: Razvan-Alexandru VULPE

June 5, 2025

Section I: API Study

1. Introduction to vCare-as-a-Service

vCare-as-a-Service is a modular and extensible REST API framework designed to bring evidence-based clinical pathways into external health and rehabilitation ecosystems. Third-party platforms such as Imaginary or CC2U offer a variety of digital health services but often lack direct integration with structured medical protocols. As a result, patient guidance tends to remain generic, rather than condition-specific. Through vCare-as-a-Service, these platforms can subscribe users to appropriate clinical pathways and dynamically receive care guidance tailored to each individual's diagnosis and care plan.

This integration enhances personalization, continuity, and automation in patient monitoring. For example, a cardiac patient using a fitness coaching app could receive exercise recommendations that align with their rehabilitation stage, rather than generalized advice. This enables:

- More precise and personalized interventions,
- Longitudinal tracking of patient recovery,
- Reduction in duplicated services or conflicting advice,
- Adaptive coaching based on pathway-based clinical triggers.

Furthermore, vCare's architecture supports secure patient enrollment, discovery of services, and standardized data exchange between independent systems. The external platform continues to manage its own interface and orchestration logic while acting as a consumer of clinically validated pathway data.

2. Challenges in Consuming Unknown APIs

Building clients for APIs that evolve over time or lack complete documentation introduces several technical challenges. Most REST clients assume static endpoint structures and rely on fully defined schemas. However, vCare may dynamically expose services based on user profiles, organizational policies, or medical context.

Core technical challenges include:

- Absence of guaranteed endpoint stability or availability.
- Lack of comprehensive Swagger/OpenAPI documentation.
- The need for real-time endpoint discovery and dynamic invocation.
- Support for flexible or partial schema parsing.
- Managing endpoint-specific authentication strategies.
- Ensuring resilience against network delays or downtime.
- Handling cascading errors, retries, and partial failures.

To address these, a resilient client must:

- Periodically poll a centralized service discovery mechanism,

- Maintain a cache of known, validated service IDs,
- Dynamically construct REST URLs based on service metadata,
- Implement robust error-handling and timeout logic,
- Parse unknown or partial JSON schemas safely.

Such a design minimizes coupling to backend structures and supports long-term maintainability.

3. Strategies for Dynamic API Discovery

3.1 Polling a Service Bus

Polling is a straightforward strategy for dynamic service discovery. A known endpoint (the service bus) is queried periodically to retrieve a list of active services.

Example JSON response:

```
["service42", "service101", "service_trial"]
```

These IDs are used to construct target REST calls (e.g., `/api/service/42`). This allows clients to detect and consume newly available or updated services.

Spring Boot Example:

```
RestTemplate restTemplate = new RestTemplate();
List ids = restTemplate.getForObject(
"http://localhost:8080/service-bus", List.class);
for (String id : ids) {
String endpoint = String.format("http://localhost:8080/api/service/%s", id)
ServiceResponse response = restTemplate.getForObject(endpoint, ServiceRespo
System.out.println(response);
}
```

3.2 HATEOAS for REST Navigation

HATEOAS (Hypermedia as the Engine of Application State) allows API responses to embed links to related actions. Clients can navigate dynamically without hardcoded URLs.

Example:

```
{
"id": "service42",
"status": "active",
"_links": {
"details": { "href": "/api/service/42/details" },
"status": { "href": "/api/service/42/status" }
}
}
```

This model reduces client-server coupling and supports future extensibility.

3.3 Self-Describing APIs and Meta Endpoints

Where possible, systems should expose dedicated metadata endpoints:

- `/services` — list of available services.
- `/capabilities` — schema and version information.
- `/api-docs` — OpenAPI specification.

Fallback strategies include:

- Using JSON introspection for schema inference,
- Handling unknown payloads via generic maps,
- Maintaining manual configuration for known variants.

4. Proposed Architecture for API Client

The API client consists of three core modules:

- **PollingModule** — periodically fetches service IDs from the bus.
- **IDResolver** — filters, validates, and de-duplicates service IDs.
- **ServiceConsumer** — builds target URLs and processes REST responses.

Each component is independently testable, stateless, and designed for resilience.
Sequence Diagram Flow:

1. Client invokes `/service-bus`.
2. JSON array of service IDs is returned.
3. IDs are validated and transformed into REST calls.
4. Responses are parsed and mapped into data models.

Model Example:

```
public class ServiceResponse {
private String id;
private String status;
private String pathway;
// Getters and Setters
}
```

5. Technologies and Tools

- Java 17+
- Spring Boot 3.x
- Spring WebClient or RestTemplate
- Jackson for JSON mapping
- Postman and SwaggerUI for testing
- MockServer or WireMock for mocking endpoints
- JUnit and Mockito for testing
- Maven or Gradle for builds
- Spring Actuator for exposing metrics and health
- SLF4J and Logback for logging

6. Summary and Next Steps

This section has outlined the architectural principles and practical strategies for consuming dynamic REST endpoints from the vCare-as-a-Service platform. It emphasized polling-based discovery, dynamic URL construction, error handling, and schema flexibility as critical design components.

The proposed client architecture is lightweight, maintainable, and well-suited for evolving APIs. It supports long-term extensibility and seamless integration into partner ecosystems. In future stages, we will cover:

- Implementing unit and integration tests,
- Handling OAuth2-secured endpoints,
- Improving performance via caching and concurrency,
- Deploying the solution as a containerized service,
- Automating updates through CI/CD pipelines.

This foundation ensures that third-party platforms can reliably interact with vCare's evolving clinical services, ultimately leading to better patient outcomes and streamlined digital care experiences.