Assignment #10 & 11:
Hypermedia APIs and the Constrained Web of Things

Advanced Software and Systems Engineering - HS 2023

Submitted by:
Michael Bruelisauer, Daniel Leal, Kaan Aydin, Stephan Nef

Submission Date:
December 23, 2023

Task 4: Group Reflections

1.1 For each of the tasks, what is the information you had to hard-code into your executors?

1.1.1 Cherrybot

Cherrybot SPARSQL Query:

```java
private String buildSparqlQuery() {
    return """
        @prefix td: <https://www.w3.org/2019/wot/td#>.
        select ?x
        where { ?x a <https://interactions.ics.unisg.ch/ufactory#PhysicalxArm7Robot> }
        """;
}
```

Listing 1: Hard-coded SPARQL Query in CherrybotExecutor.java

Hardcoded Code for register- and deleteOperator:

```java
private String registerOperator(ThingDescription td) throws IOException {
        Optional<ActionAffordance> potentialRegisterAction = td.getActionByName("registerOperator");
        if (potentialRegisterAction.isPresent()) {
            ActionAffordance registerAction = potentialRegisterAction.get();
            Optional<Form> form = registerAction.getFirstForm();
            if (form.isPresent()) {
                TDHttpRequest tdHttpRequest = new TDHttpRequest(form.get(), TD.invokeAction);
                DataSchema dataSchema = registerAction.getInputSchema().get();
                System.out.println(tdHttpRequest);
                Map<String, Object> payload = new HashMap<>();
                payload.put("http://xmlns.com/foaf/0.1/Name", "Michael Bruelisauer");
                payload.put("http://xmlns.com/foaf/0.1/Mbox", "michael.bruelisauer@student.unisg.ch");
                tdHttpRequest.setObjectPayload((ObjectSchema) dataSchema, payload);
                System.out.println(tdHttpRequest);
                TDHttpResponse response = tdHttpRequest.execute();
                String location = response.getHeaders().get("location");
                int tokenIndex = location.lastIndexOf('/');
                String token = location.substring(tokenIndex + 1);
                LOGGER.info("Token: " + token);
                return token;
            }
        }
        return null;
    }

    private String deleteOperator(ThingDescription td, String token){
        Optional<ActionAffordance> potentialRegisterAction = td.getActionByName("removeOperator");
        if (potentialRegisterAction.isPresent()) {
            ActionAffordance deleteOepratorAction = potentialRegisterAction.get();
            Optional<Form> form = deleteOepratorAction.getFirstForm();
            if (form.isPresent()) {
                Form deleteForm = form.get();
                String baseUrl = deleteForm.getTarget();
                String finalUrl = baseUrl.replace("%7Btoken%7D", token);
                try {
                    URL url = new URL(finalUrl);
                    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                    conn.setRequestMethod("DELETE"); // Set method if it's not DELETE
                    int responseCode = conn.getResponseCode();
                    LOGGER.info("Delete Operator Response Code: " + responseCode);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
        return null;
    }
}
```

Listing 2: Hard-coded for registerOperator and deleteOperator Method

Our code consists of two methods, `registerOperator` and `deleteOperator`, which interact with a Thing Description (TD) in a Web of Things (WoT) context.

Hard-Coded Elements:

- In `processAction`, the matching of the action is hard-coded to check if it equals "setGripper" to define the datatype of the payload.
- In `registerOperator`, the hard-coded elements are the credentials.
- In `deleteOperator`, the method of the HTTP request could be considered as hard-coded.

Our Idea:

- `registerOperator`: Searches for a specific action, constructs an HTTP request, executes it, and processes the response so we can register an operator with the hard-coded name and email for the payload.

- `deleteOperator`: Looks for an action, constructs a URL for a DELETE request, and executes the request so the operator will be deleted after the task is done and the robot is free for another operator.

Reasoning for Hard-Coding: It simplifies our development and testing, especially in our early stages. It provides straightforward execution in controlled environments but we are aware that it also reduces the flexibility and scalability of our application.

1.1.2 Mirocard

Mirocard SPARSQL Query:

```
private String buildSparqlQuery() {
        return """
            @prefix td: <https://www.w3.org/2019/wot/td#>.
            select ?x
            where { ?x a <https://interactions.ics.unisg.ch/mirogate#
   Mirogate> }
        """;
}
```

Listing 3: Hard-coded SPARQL Query in MirocardExecutor.java

1.2 What are the advantages and disadvantages you see for using Hypermedia APIs?

Although there are many more advantages and disadvantages of using hypermedia APIs to consider, we list below the most important ones based on our (limited) experience.

Advantages of Using Hypermedia APIs in the WoT Context

- **Dynamic Interaction:** Hypermedia APIs allow clients to dynamically navigate across resources. Due to the dynamic discovery of action and resources, the requirement for prior knowledge of the API structure can be limited, making the API more adaptable to changes
- **Decoupling of Executors (clients) and Devices (server):** Hypermedia APIs allow for the separation of executors from the physical web-enabled devices they control. This promotes loose coupling between client and server so that changes to the server-side logic do not necessarily break the client - improving maintainability and evolvability. Decoupling primarily refers to the architectural separation between the clients (executors) and servers (devices). This separation enables each component to evolve independently, enhancing system resilience. Changes made on the server side, such as updates or modifications to the devices or their functionalities, do not necessitate corresponding changes on the client side. This is critical in maintaining system stability and allows for independent evolution and maintenance of different system components.
- **Flexibility and Scalability:** As the devices dictate the flow of an application state (which are discovered by the clients / executors), APIs can evolve over time without disrupting existing clients. This makes the system more adaptable to change and scalable. Flexibility is about the ability of the system, particularly the client side, to adapt to changes dynamically. In the context of Hypermedia APIs, this means the capability of clients to discover and interact with various resources and actions at runtime without requiring prior knowledge of the API's structure. Flexibility is what enables clients to navigate and utilize different functionalities of the server as they become available or change over time. It is the adaptability of the system to new requirements, devices, or scenarios that may emerge in a dynamic IoT environment.

- **Interoperability:** Hypermedia APIs enhance interoperability among diverse devices and systems in an IoT ecosystem, allowing different devices, (with varying protocols) to seamlessly communicate and work together

Disadvantages of Using Hypermedia APIs in the WoT Context

- **Complexity:** A hypermedia APIs introduce additional complexity in the design and implementation of systems compared to traditional APIs. They require the implementation of additional layers for dynamic link generation and handling, as well as more sophisticated client logic to interpret and navigate these links.
- **Performance Overhead:** The abstraction and runtime device interaction might lead to performance overhead compared to direct integrations. The use of hyperlinks and additional processing required to handle layers can increase response size and potentially affect performance.
- **Dependency on Network Reliability:** Hypermedia APIs rely heavily on network communications, thus any network-related issues can significantly impact system performance. This dependency makes the reliability of network connections a critical factor of systems using hypermedia APIs.
- **Security Concerns:** The increased connecitivity and flexibility can introduce greater security for systems using hypermedia APIs. Given the interconnected nature of IoT environments, these systems require more stringent security measures to protect against potential vulnerabilities and threats

1.3 How do Interface Definition Languages (IDLs) and Hypermedia APIs relate to one another?

Interface Definition Languages (IDLs)

Interface Definition Languages are used to describe interfaces in a language-neutral way for inter-program communication. They define data types and method signatures, ensuring seamless interaction between different systems. Examples include Web Services Description Language (WSDL) for SOAP and the Protocol Buffers language for gRPC.

Hypermedia APIs

Hypermedia APIs, or Hypermedia as the Engine of Application State (HATEOAS), extend RESTful APIs by ensuring that clients interact with the API entirely through dynamically provided hyperlinks. Unlike traditional APIs with predefined structures, Hypermedia APIs are self-descriptive and discoverable in real-time.

Interface Definition Languages (IDLs) and Hypermedia APIs represent two sides of the API design coin. IDLs are about defining clear, static contracts for API interactions. In contrast, Hypermedia APIs, like those following the REST style, thrive on their dynamic nature, using hyperlinks for real-time, flexible interactions. While IDLs prioritize structure and predictability, Hypermedia APIs emphasize discoverability and adaptability. The challenge lies in harmonizing these divergent approaches to create APIs that are both well-defined and flexible, an ongoing endeavor in API development (Scharhag, 2020, SmartBear, 2021, APIs, 2023).

Challenge of Harmonizing IDLs and Hypermedia APIs:

The quest to merge Interface Definition Languages (IDLs) and Hypermedia APIs stems from a desire to blend the best of both worlds: the predictability and clarity of IDLs with the dynamic, adaptive nature of Hypermedia APIs. IDLs excel in defining explicit contracts for APIs, ensuring a well-structured and understandable interface. However, they often lack the flexibility to adapt to changes without updating the client code. Hypermedia APIs, conversely, offer this adaptability and enable clients to discover actions dynamically, but they can introduce complexities and lack the straightforwardness of IDLs. Harmonizing these approaches aims to create APIs that are both robustly defined and agile enough to evolve, a balancing act that continues to challenge and innovate API development.

Relationship Between IDLs and Hypermedia APIs

An example to illustrate this point would involve a smart home system. This system integrates various IoT devices such as smart thermostats, lights, and security cameras. The goal is to create a flexible, interoperable environment where these devices can be controlled and monitored through a central application.

Use of IDLs:

In this system, we use an Interface Definition Language (IDL) to define the interfaces between the central application and the individual smart devices. The IDL provides a clear, structured way to specify the data types, methods, and protocols for communication. For example, the IDL defines methods like setTemperature, turnOnLight, and activateCamera, ensuring that the central application and the smart devices understand exactly how to interact with each other.

Integration of Hypermedia APIs

Alongside the IDL, the system also employs Hypermedia APIs following the RESTful principles with HATEOAS. These APIs are used for dynamic interactions within the smart home ecosystem. For instance, when the central application queries the smart thermostat, the response includes hyperlinks to related actions and resources, like adjusting temperature settings or retrieving historical temperature data. This dynamic approach allows the application to discover and interact with features of the devices in real-time, without needing prior knowledge of the API's structure.

Harmonization in Action

The harmonization occurs in how the system leverages both the predictability and clarity of IDLs with the flexibility and adaptability of Hypermedia APIs. The IDL component ensures that the basic interactions between the central application and the devices are stable and well-defined. In contrast, the Hypermedia APIs provide the system with the ability to adapt to changes, such as adding new types of devices or updating existing ones, without requiring significant modifications to the central application.

Benefits and Outcomes

This approach results in a smart home system that is both robust and adaptable. The IDLs maintain consistency and reliability in core interactions, while the Hypermedia APIs offer the system the flexibility to evolve over time, enhancing its scalability and user experience. It demonstrates a practical implementation of harmonizing IDLs and Hypermedia APIs, addressing the challenges of developing interoperable, dynamic systems in the IoT domain.

Relationship Between IDLs and Hypermedia APIs

Descriptive vs. Discoverable

- **IDLs** provide a static contract describing the structure and semantics of data and functions, such as WSDL in SOAP.
- **Hypermedia APIs** are dynamic and discoverable, relying on hyperlinks for interaction, as in the use of W3C WoT Thing Descriptions (TD) in IoT ecosystems.

Flexibility and Adaptability

- **IDLs** offer a rigid structure, suitable for stable interfaces but less adaptable to changes, requiring modifications in the IDL and client implementations.
- **Hypermedia APIs** provide greater flexibility, as seen with the Miro-Card in IoT, allowing runtime adaptation without recompilation.

Use Cases

- **IDLs** are ideal for environments with stable and well-defined interfaces, like enterprise applications and microservices.
- **Hypermedia APIs** excel in dynamic environments like IoT, where system components frequently change.

Conclusion

While IDLs enforce interfaces in a language-agnostic manner, their static nature can be limiting. Hypermedia APIs, conversely, offer flexibility and runtime discoverability, suitable for adaptive systems like in the TAPAS ecosystem. This demonstrates the evolving needs of modern software architectures in distributed and heterogeneous environments.

1.4 Addressing Feedback:

Our current implementation selects affordances from a Thing Description (TD) based on their names, such as ”toggle.” This approach is primarily driven by its simplicity and the straightforward nature of implementation, particularly useful in the early stages of development. Using names allows for quick identification and interaction with affordances, facilitating a more intuitive development process, especially for those new to the Web of Things (WoT) paradigm.

The alternative approach involves selecting affordances using semantic annotations, like ”https://saref.etsi.org/core/ToggleCommand”. Semantic annotations are URIs that link to a standardized vocabulary, offering a universal and explicit way of identifying affordances. This method is inherently richer and more precise, as it leverages a shared understanding of terms within the WoT community, fostering better interoperability and scalability.

In assessing the differences between using the names of affordances and semantic annotations in our implementation, several key aspects emerge. Firstly, interoperability and standardization are markedly enhanced with semantic annotations. Unlike the potentially ambiguous and inconsistent approach of using names, where different developers might use varied terminologies for similar functionalities, semantic annotations provide a universal and explicit method of identification. These annotations are linked to standardized vocabularies, ensuring a consistent understanding across various systems and developers within the Web of Things (WoT) community.

Another distinction lies in scalability and flexibility. The simplicity of using names, while beneficial in the initial development stages, can become a limiting factor as the system scales and interfaces with a broader array of devices and services. In contrast, semantic annotations promote both scalability and flexibility. Aligning with standards, they enable the system to adapt more readily to diverse and evolving IoT ecosystems, making them a more future-proof choice.

In conclusion, while our current approach using the names of affordances offers simplicity and immediate ease of development, the long-term benefits of employing semantic annotations are clear. These include enhanced interoperability, better alignment with global standards, and improved scalability and adaptability of the system.

References

APIs, N. (2023). Exploring the hidden powers of hypermedia.
Scharhag, M. (2020). Introduction to hypermedia rest apis.
SmartBear. (2021). What is hypermedia?