Integração de Sistemas

3º ano • 1° semestre • Ano Letivo 2025/2026

Project

SOMIOD: Service Oriented Middleware for Interoperability and Open Data

Project Overview

The Internet of Things (IoT) is increasingly recognized as a key enabler, or at least a significant contributor, to addressing a wide range of societal challenges, including climate change, energy efficiency, and sustainability. From smart homes and offices to smart cities, the seamless collection of data has the potential to benefit society at large, support the business sector, and empower local and state governments to develop innovative services that promote social cohesion, environmental stewardship, economic growth, and a more intelligent infrastructure.
However, many current and proposed IoT initiatives rely heavily on proprietary protocols and private cloud infrastructures. This reliance significantly hampers interoperability and data sharing across devices, applications, and platforms, a phenomenon often referred to as the "Silo of Things" [1].

The aim of this project is to develop a **service-oriented middleware** that standardizes how data is accessed, written, and notified, regardless of the application domain. This uniform approach promotes **interoperability** and supports **open data** principles ensuring that data interactions and application development follow a consistent model (e.g.: data is always accessed and written in the same way and applications are always created in the same way). As a result, users can seamlessly access data and extend or create new applications and services always using the same approach. To achieve this, the project proposes the use of **Web Services** and a **Web-based resource structure**, always based on the **open Web standards**.

More concretely, the following resources, features, and capabilities must be implemented:

The Web service URL must always start by the following structure:

```
http://<domain:port>/api/somiod/...
```

The SOMIOD middleware must support the following resources, organized according to the specified hierarchical structure, with a maximum depth of three levels:

```
application (A)
        container (1)
                content-instance (0..N)
                subscription (0..N)
        container (N)
                content-instance (0..N)
                subscription (0..N)
application (B)
        container (1)
                content-instance (0..N)
                subscription (0..N)
        container (N)
                content-instance (0..N)
                subscription (0..N)
 ...
```

Next, the list of properties, per resource type (**res-type**) is presented. It is **mandatory** to implement the middleware using the **exact property names** presented below, otherwise **existing applications** will not be able to use the middleware, and it will have negative impact on students' **assessment**. Additional properties can be added to handle storage or hierarchy, but they should remain hidden from the user/programmer perspective.

res-type values	**Resource type properties handled by user/programmer**
application	`resource-name <string>`
container	`resource-name <string>`
content-instance	`resource-name <string>; content-type <string>; content <string>`
subscription	`resource-name <string>; evt <int>; endpoint <string>`

An **application** resource represents a specific real-world application. Middleware should support multiple applications.

A **container** resource represents a way to group other resources (`content-instances` and `subscriptions`). Each `application` can have one or more `containers` as child resources.

A **content-instance** represents each data record created on a specific application `container`. The `content-type` field stores the format of the content being stored (e.g. `“application/xml”`, `”application/json”`, `“application/plaintext”`).

A **subscription** resource represents the mechanism able to trigger notifications when a **new** `content-instance` is added to the `container`, or when a `content-instance` is **deleted** from the `container` where the `subscription` is created.

Each of the resources listed above must be assigned a unique `resource-name` to ensure proper identification and avoid conflicts within the middleware. Through the Web Service RESTful API, it must be possible to **create**, **modify**, **obtain**, **delete** and **discover** each available resource. The **content-instance** and **subscription** resource types do not support (allow) the update operation.

For **creation** and **discover** operations, the used URL should point to the parent resource[1], while the **delete** and **get** operations the URL should point to the target resource name. As `containers` support two types of child objects (e.g. `content-instance` and `subscription`), the strait URL to target resource name is reserved to address `content-instances` while `subscriptions` are addressed using the virtual (non-existent) node called “**subs**”. Here an example of addressing a `content-instance` of resource name “ci-a” and `subscription` with resource name “sub-x” in the same `container` with resource name “cont-p”, when dealing with delete or get operations:

```
http://<domain:port>/api/somiod/app_resource_name/cont-p/ci-a
http://<domain:port>/api/somiod/app_resource_name/cont-p/subs/sub-x
```

For subscriptions, besides the `resource-name` field, they should support two types of events (`evt`): **creation** or **deletion**, or both, and an `endpoint` where the notification should be fired. For each triggered notification, the resource created and/or deleted must be part of the notification information, as well as the type of triggered event (creation, deletion). It must be possible to trigger a notification via **MQTT** or **HTTP** (e.g., subscriptions should support MQTT and HTTP endpoints). If required, the channel name used to fire the subscription should have the same name of the path to the source container resource (e.g. `api/somiod/app-xx/container-yy`).

When **creating** or **updating** a resource via the SOMIOD middleware, the system should return the appropriate HTTP status code along with the full set of properties for the resource type that was created or updated. This is especially critical when the resource name is auto-generated to ensure uniqueness, as the client must be informed of the exact identifier and attributes of the resulting resource.

[1] e.g.: POST `http://<domain:port>/api/somiod/app-xx` : this URL allows to identify the existing parent where the `container` resource should be created.

The SOMIOD middleware should persist resources and resource's data on a database. Remember that, inside database, the presented SOMIOD resources can have extra fields. However, RESTful API just deals with the resource fields presented above.

Transferred data should always adopt the JSON format, while the **content** field of the **content-instance** resource can be formatted using XML, JSON or plain text.

The HTTP action used in the RESTful API request should identify the target resource using the unique `resource-name` field (**and not any kind of resource id**).

The **discovery** operation is represented by the **GET** HTTP action and by the presence of an **HTTP header** called **"somiod-discovery:** <res-type>**"**, where <res-type> should be **application**, **container**, **content-instance** or **subscription**. The return is always a **list** of **paths** to find a `resource-name`. For example, is the user/programmer launches a discovery operation for URL **http://<domain:9876>/api/somiod/app5** targeting **content-instances**, the following sample JSON list could be returned:

```
["/api/somiod/app5/cont1/ci1", "/api/somiod/app5/cont1/ci2", "/api/somiod/app5/cont2/ci1"]
```

To clarify the difference between a standard GET operation and the **somiod-discovery** operation, take the following examples:

Example of some endpoints used with the somiod-discovery operation	
`GET -H "content-type: application/json"` `http://<domain:9876>/api/somiod/app5`	returns "app5" data properties.
`GET -H "content-type: application/json"` `http://<domain:9876>/api/somiod/app5/cont1/data1`	returns the content-type "data1" record.
`GET -H "content-type: application/json" -H` **`"somiod-discovery: application"`** `http://<domain:9876>/api/somiod`	returns the list of paths to all applications inside api/somiod path (recursively).
`GET -H "content-type: application/json" -H` **`"somiod-discovery: container"`** `http://<domain:9876>/api/somiod/app5`	returns the list of all paths to all containers that are child of "app5" (recursively).
`GET -H "content-type: application/json" -H` **`"somiod-discovery: content-instance"`** `http://<domain:9876>/api/somiod/app5`	returns the list of paths to all content-instances records that are child of "app5" (recursively).
`GET -H "content-type: application/json" -H` **`"somiod-discovery: content-instance"`** `http://<domain:9876>/api/somiod`	returns the list of paths to all content-instance records that are child of somiod (recursively). It returns all existent content-instances.
...and all the other endpoints already addressed and those that can be applied in this context.	

The **GET *all*** HTTP action **should not be supported** in the middleware, as it instead supports the **GET** HTTP action for service discovery using the HTTP header `somiod-discovery` as previously exemplified.

More details about the properties that must be followed:

- Each resource created in the database must have a creation timestamp named **creation-datetime** in a simplified ISO date format, e.g., 2024-09-25T12:34:23.
- The **evt** property can only contain number values: 1 for creation or 2 for deletion.
- The MQTT endpoint property simply stores the endpoint of the messaging broker. The channel is guessed during run-time. The HTTP endpoint points to a URL that supports the POST action.
- Every time a resource is requested to be created **in the absence** of a **unique name** for each resource type, the software should create a unique name for it.
- Property **res-type**, usually referred to in the **HTTP body** must be `application`, `container`, `content-instance`, or `subscription`.

The SOMIOD middleware (RESTful API) must be **well-documented using Swagger**.

SOMIOD Middleware in Action - Implementation Use Cases

The second part of the project deals with the application "layer" to test the SOMIOD middleware.

Students Implementation scenario

Using the upcoming use case as inspiration, design and implement your own test application scenario based on a different context. **Your scenario should be distinct from the example provided**, demonstrating originality while applying the same middleware principles. An additional feature must be implemented to ensure that all incoming notifications for each application are serialized into **XML files**, validated against a predefined **XML schema**.

Example Use Case

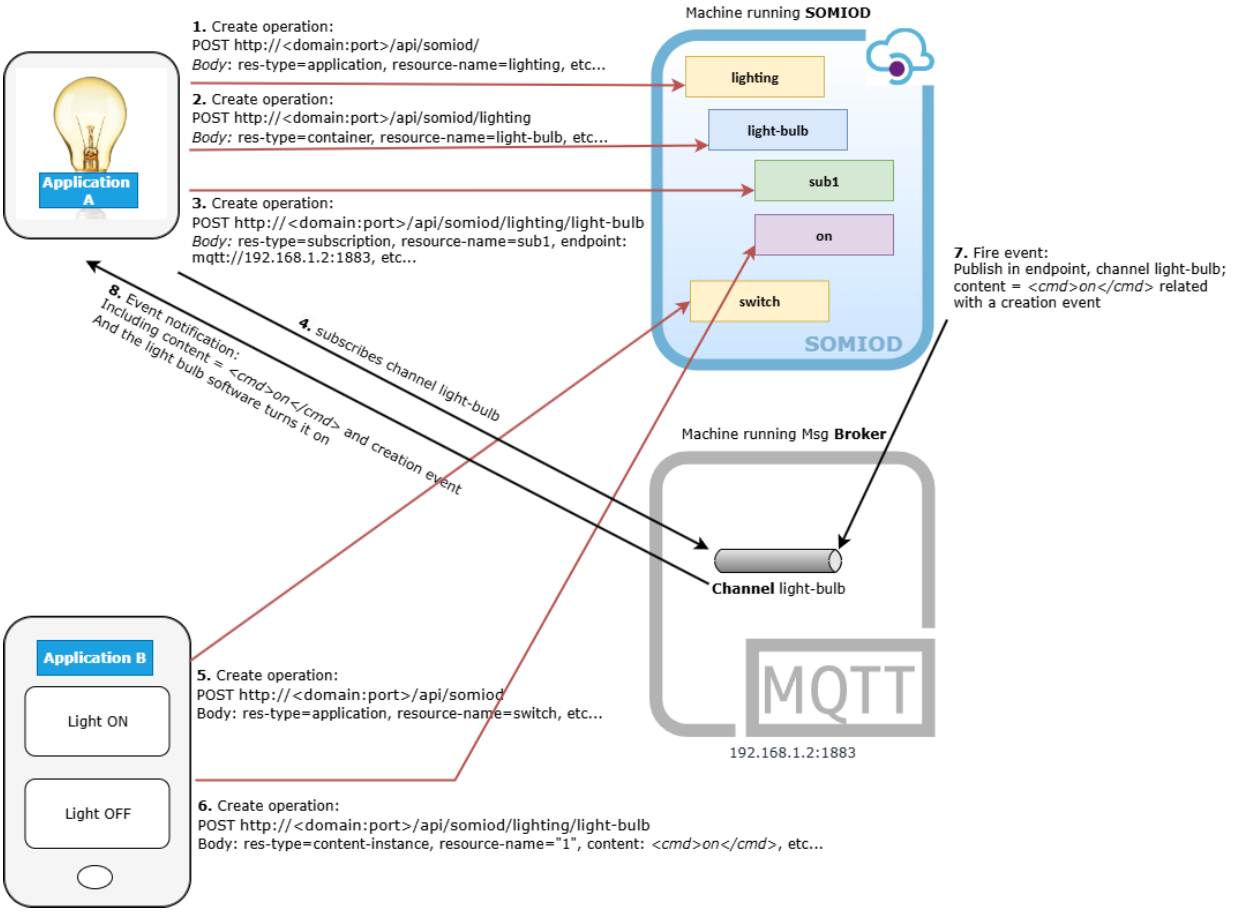
In the context of Internet of Things, imagine we want to create or implement a light bulb (**Application A**) to be controlled through a mobile application (**Application B**). Consider that the light bulb includes a RPI zero inside or any other form of single board computer.

Hence, when the light bulb is attached to the support for the first time, it creates its representing application resource with the resource-name (for example) **lighting.** Besides the application resource, the light bulb also creates a container resource called **light-bulb**.

On the other hand, (by definition) the first time the mobile app is executed, it creates its representing application called **switch**.

The light bulb should create a subscription on its container for event **creation**. Hence, to turn "on" or "off" the light bulb the mobile *app* has just to create (write) a content-instance resource on the "light-bulb" container, as this action will trigger the creation notification, and the light bulb will be notified about this new state.

This sample application is represented in the next block diagram.



1. Create operation:
POST http://<domain:port>/api/somiod/
Body: res-type=application, resource-name=lighting, etc...

2. Create operation:
POST http://<domain:port>/api/somiod/lighting
Body: res-type=container, resource-name=light-bulb, etc...

3. Create operation:
POST http://<domain:port>/api/somiod/lighting/light-bulb
Body: res-type=subscription, resource-name=sub1, endpoint: mqtt://192.168.1.2:1883, etc...

4. subscribes channel light-bulb

5. Create operation:
POST http://<domain:port>/api/somiod
Body: res-type=application, resource-name=switch, etc...

6. Create operation:
POST http://<domain:port>/api/somiod/lighting/light-bulb
Body: res-type=content-instance, resource-name="1", content: *<cmd>on</cmd>*, etc...

7. Fire event:
Publish in endpoint, channel light-bulb;
content = *<cmd>on</cmd>* related with a creation event

8. Event notification:
Including content = *<cmd>on</cmd>* and creation event
And the light bulb software turns it on

Figure 1 - Generic architecture of a sample testing application scenario

Project assessment

Project assessment will rely on the following criteria:

Criterion	% [0-100]
Middleware	**60%**
CRUD+discovery operations for the application resource	15%
CRUD+ discovery operations for the container resource	15%
CRUD+ discovery operations for the content-instance resource	15%
CRUD+ discovery operations for the subscription resource	15%
Testing application - able to test middleware operations (final hardware and final control software, if needed, may be emulated with desktop applications).	**25%**
Project report (among other information, should include a section with **complete** cURL commands for CRUD+Discovery operations for all SOMIOD resources)	**15%**

Guidelines

Groups must have 4 students, which can be from different practical classes. The name and number of the group members must be submitted on the course page (http://ead.ipleiria.pt - Moodle) by the end of November. The project delivery must be performed until the date published in the official assessment calendar. There is a mandatory project discussion (individual practical exam about the project) that will occur later in the semester, also published in the course assessment calendar.
Except for the database, the project must be developed using solely the **technologies used in practical classes**.

Besides the source code of the entire solution, it is also **mandatory** to deliver a **written report** following the published template, which could also include Appendixes. The template for the report is available on the course web page. Remember to tag your source code (using text headers) because, as this project is an evolutionary project, your source code could be selected to be used as a starting point in the next school year.

Delivery

The project delivery must be done on the course web page (http://ead.ipleiria.pt) where students must submit a link to an external ZIP file (or .7z file) with all the project material. The link to the project delivery should use an external cloud service, e.g., OneDrive, Dropbox (shared file), WeTransfer, etc.
Therefore, students must guaranty that all the following material is included in a ZIP file (.7z) when delivering the project, and with the following organization structure:

- Text file (**identification.txt**) with all the information about the team members (number, name, and e-mail) and course name, etc.
- Folder **Project**: all the source code must be included in this folder. The source code of the project must include source files and other resources (files, executables, dll's, etc.)
- Folder **Report**: the written report in PDF/.docx format. Don't forget that at the end of the report (appendix) it is mandatory to identify which features each team member has implemented.
- Folder **Other**: other files required by the project that were used by the team members but were not included in the previous folders.
- Folder **Data**: the SQL data files (.sql) and the database files (the .mdf and .ldf file for a SQL Server database). Therefore, the database must have data to allow the testing of the project. Also, it is mandatory to deliver a database script **(.sql file**) including the **database structure** and **data/database records**. If using a database in the cloud, the necessary credentials to connect to the database should also be provided. If a MySQL database is used, all necessary information and the complete database scheme must also be provided.

References

[1] Mohapatra, S. K., Bhuyan, J. N., Asundi, P., & Singh, A. (2016). A Solution Framework for Managing Internet of Things (IOT). *International journal of Computer Networks & Communications.—2016.—8.—P*, 73-87.