DomoticASW

Distribuited System

Corrado Stortini
corrado.stortini2@studio.unibo.it

Marco Raggini
marco.raggini2@studio.unibo.it

Francesco Carlucci
francesco.carlucci6@studio.unibo.it

Accademic year 2023/2024

Contents

1 Goal/s of the project

Create a smart home system and its own protocol, allowing devices to be added or removed dynamically (without having to stop the system).

The protocol enables smart home devices to describe the actions they support and the data they generate. This means device manufacturers can create compatible devices without requiring changes to the system.

Users interact with the system through web client. The web client doesn't talk directly to the smart devices but communicates through a server installed in the house.

The server is also accessible online, so users can manage their home remotely.

Automations can be set up, triggered either by users or external events.

The client can receive custom push notifications from the server, such as:

- Alerts when a device goes offline
- Notifications when the temperature hits a set threshold
- Updates when the washing machine finishes its cycle
- Results of a scheduled automation

There are two types of users:

- **Admin**: Can add and remove devices from the system and define which devices other users can interact with. (The admin also uses the system, he's not technical staff.)
- **User**: Can only interact with devices they've been given access to by the admin.

The system includes an authentication feature so that only authenticated users can interact with it. Users can register themselves but won't be able to use the system until approved by an admin.

1.1 Use scenarios

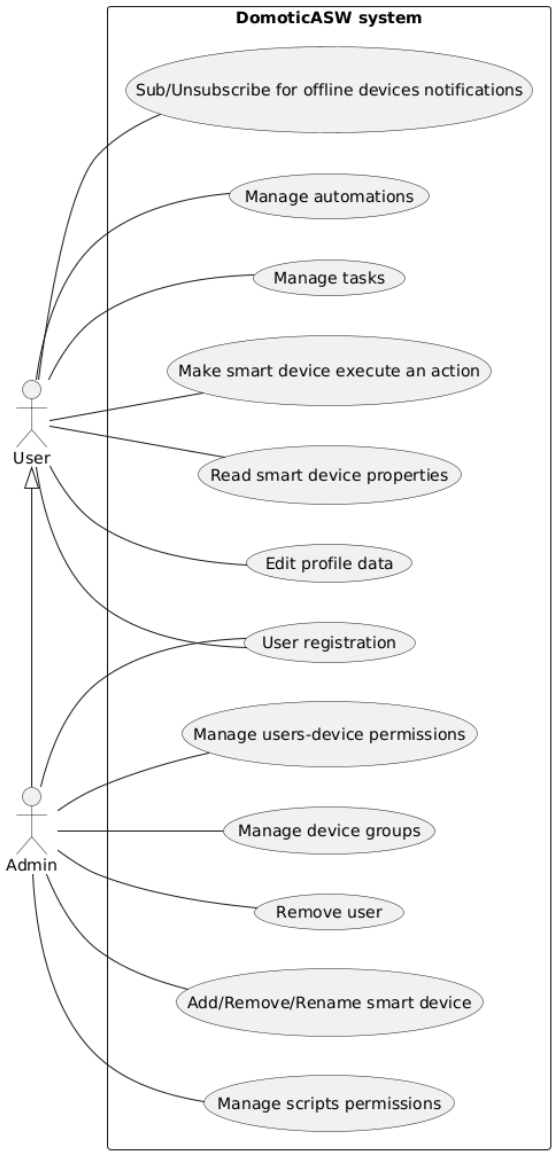


Figure 1: use cases diagram

2 Technologies used

- Stack **MEVN**: requirement of ASW exam
- Other languages or runtime to implement the devices: requirement of SPE exam
- HTTP REST APIs for client-server and server-device communication
- Docker to containerize components (server and devices): requirement of SPE exam
- Docker Compose for easier development and testing: requirement of SPE exam
- Kubernetes for deployment

3 Requirements Analysis

3.1 Functional requirements

As a	I want	so that
Admin	to add, remove or rename smart devices in my smart home	I can extend my domotic system
Admin	to be able to set up user-device permissions and as a consequence tasks can be executed only by users that own the user-device permission for every device included in the task	I can prohibit unintended use of the system
Admin	tasks to have a user whitelist and a user blacklist to override user-device permissions	I can prohibit unintended use of the system
Admin	tasks and automations to have an editlist to specify which user can edit that	I can prohibit unintended use of the system
Admin	to review and accept or decline users registration requests	I can manage who has access to the system
Admin	to be able to remove users from the system	I can manage who has access to the system
Admin	to organize the home devices in groups	it's easier to manage them
User	to see the properties exposed by my smart devices	I can gain knowledge about the state of the house
User	to make smart devices execute an action	I can alter the state of the house
User	to set up tasks to be executed with one click	I can automate the execution of multiple instructions
User	to express conditional logic in my tasks	I can achieve complex behaviour
User	to set up automations to be executed periodically or when specific triggering events happen	my home can do stuff even without me taking care of it
User	to enable and disable automations	I can stop some automations to run for a while
User	to receive a notification when a specific device goes offline	I can fix problems in case they happen
User	to edit my profile data like my nickname, email and password	I can keep my data up to date

3.2 Non functional requirements

- The domotic system must be compatible with devices produced by multiple manufacturers.
- The domotic system should be accessible through the internet so that it can be accessed remotely.
- The domotic system should be accessible through the local network so that internet access is not required when the user is physically in their house.
- The domotic system should be accessible both from desktop and mobile clients.
- Users (and the admin) must be authenticated in order to interact with the system.
- The server should restart automatically in case of a crash.

4 Design

4.1 Domain modelling

4.1.1 Users Management

This bounded context exposes a UsersService which offers all the methods to implement the use cases.

UserService service also includes methods for authentication (login, verifyToken, validateToken). The idea is that every other service will use this one to validate authentication tokens received by the client.

UsersService is responsible for managing the system's users. Essentially, its purpose is to centralize all logic related to authentication, authorization, and user lifecycle management.

Main responsibilities:

- **User lifecycle**
  - Registration of new users.
  - Approval or rejection of requests by the Admin.
- **User data management**
  - Retrieval of personal data.
  - Modification of nickname and password.
  - Deletion of users from the system.
- **Role-based authorization**
  - Distinction between Admin and User.

  - Enforcement of security rules.

- **Authentication management**
  - Credential validation.
  - Generation and verification of JWT tokens.

Security Rules

- The first registered user automatically becomes Admin.
- Regular users can only operate on their own data.
- The Admin has additional privileges: management of users and requests.
- All operations (except login and registration) require a valid token.

4.1.2 Devices Management

This bounded context exposes:

- a **DevicesService**
- a **DeviceGroupsService**
- a **DeviceStatusesService**
- a **DeviceEventsService**

More complex stuff explanation:

- The DeviceStatusesService service will be responsible for keeping the devices DeviceStatus up to date, and allows for subscribers to listen to status change events.
- The DeviceEventsService service will be responsible for receiving DeviceEvents (through the publishEvent method) from the devices, and allows for subscribers to be notified about DeviceEvents.
- The DeviceGroupsService and DeviceGroups are responsible for managing the N-N relationships with Devices.
- The DevicesService is responsible for keeping DeviceGroups up to date in case of device removal.
- The DevicesService offers the method updateDeviceProperty which can be invoked by devices to notify the server about their current state. (It is also possible to subscribe for changes)

TypeConstraints

Since devices will define their own action and properties they must also define what datatypes they are.

Types are defined in the Type enum which is generic on T which reprents the actual datatype that will be used internally.

A TypeConstraint is a constraint over a type which can also have additional constraints over the values, for example: An input which requires an integer from 0 to 100 can be modeled as a subclass of TypeConstraint with Type "IntType" which overrides the validate method implementing that logic (in the diagram we called this IntRange).

DevicePropertys which have a setter will use its TypeConstraint, otherwise they will have their own TypeConstraint.

A setter is a DeviceAction whose execution is expected to set a property with the given input. This allows to create richer user interfaces where properies and actions are bound.

DeviceActions have just one TypeConstraint which constraints the input they can take. Actions that require no input can be implemented by an input of Type "VoidType".

4.1.3 Notifications

This bounded context exposes a NotificationsService which offers all the methods to implement the use cases.

The service subscribes itself to the DeviceStatusesService to be informed when a device goes offline.

To achieve eventual consistency in case of the removal of a user from the system, the next time a device offline notification would be sent to that user the service will remove that subscription from the repository.

Regarding device removal from the system, it is not a bad idea to keep the subscription. Let's say that the DeviceId is actually a hardware identifier, in case that device will be added again to the system, subscription would be valid again.

4.1.4 Permissions

This bounded context exposes a PermissionsService which offers all the methods to implement the use cases.

A more detailed explanation of PermissionsService:

- `registerScriptService`: this method is implemented because of the double dependece between PermissionsService and the ScriptService. It is used only at the start of the program.
- `canExecuteTask`: it is responsible for implementing the expected behaviour (checking user-device permissions and in the case of a presence in the blacklist or the whitelist uses that as decision factor). The method always permits to an admin to execute a task, bypassing every control.
- `canExecuteActionOnDevice`: it checks if the calling user has the permission to execute actions on the specified device. It is implemented by checking the userDevicePermissionRepository. The method permits to an admin to execute an action even if there is no record on the userDevicePermissionRepository.
- `canEdit`: it checks if the user can edit a specified script. It simply checks if the user is an admin or the editList of the script contains the user email.
- `addToWhiteList` and `addToBlackList`: they are responsible of adding and removing permissions of an user. Only an admin can use these methods and if the tasklists of the relative taskId doesn't exist, it will be created. Even if the blacklist or the whitelist doesn't exist they will be created in order to add the user email. An important note is that an admin cannot be added to the blacklist, it is possible to add an admin in the whitelist, but it doesn't produce any positive effect, because an admin can already perform every action.
- The others methods allow to add, remove or search entities. They are coherent with their names and only an admin can perform them.
- Methods that could generate some confusion about their names are getAllUserDevicePermissions and getAllUserDevicePermissionsOfAnUser. The first one will get every permission for every user device; the second one will get every permissions that an user has on devices.

4.1.5 Scripts

This bounded context exposes a ScriptsService which offers all the methods to implement the use cases.

A Script can be either a Task or an Automation the main difference is that automations have a Trigger.

Each script has a sequence of Instructions that has the following behaviour when executed based on the concrete implementation:

- SendNotificationInstruction: sends a notification to a user
- WaitInstruction: pauses the script execution for a given amount of seconds

- StartTaskInstruction: starts another task waiting for its completion
- DeviceActionInstruction: make a device execute the specified action with the given input
- ConstantInstruction:
  - CreateConstantInstruction: defines a constant with a given value
  - CreateDevicePropertyConstantInstruction: defines a constant which will get it's value at runtime from a device property
- IfInstruction: defines a sequence of instructions that are executed only if the condition evaluates to true at runtime. (The condition is based on constants)
- IfElseInstruction: defines two sequences of instructions that are executed if the condition evaluate to true or false respectively.

Conditions must operate on homogeneous types and for each type a fixed set of operators are given.

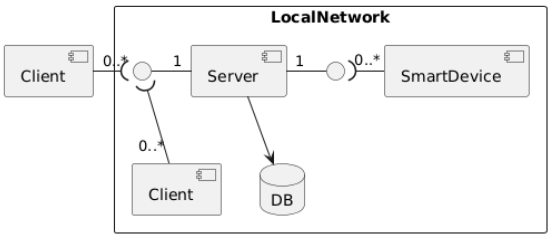
When a script is executed it creates internally an ExecutionEnvironment which is responsible for storing constant values. Tasks need a token to be executed manually by a user, while Automations do not (because they are executed automatically given a period of time or a device event firing). Tasks can also be executed without a token if executed through a Start Task Instruction inside a script.

4.1.6 Builders

Builders can check whether a script syntax is correct, but they cannot do the same with regard to semantics (because they would need to access devices data in the repository).

Scripts semantic correctness is checked by the ScriptsService

4.2 Architecture



The server implementation will follow an hexagonal architecture design.

The separation between domain, ports and adapters is clearly enforced by the folder structure:

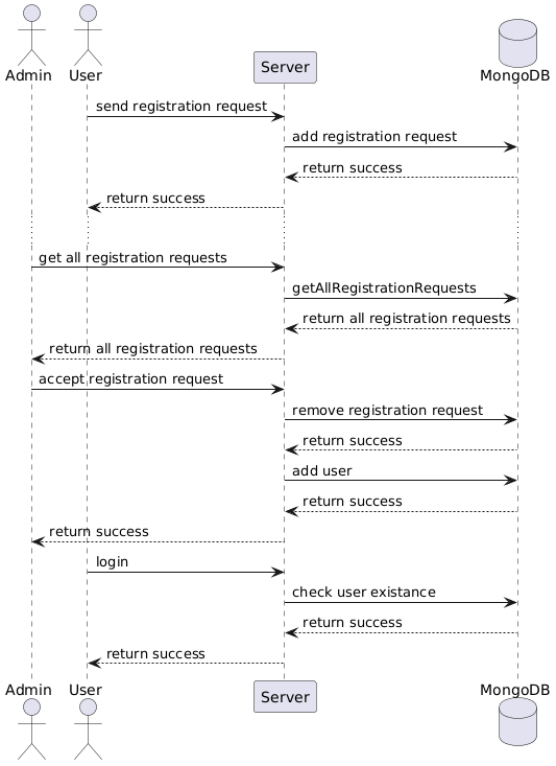
- src/domain
- src/ports
- src/adapters

In the following sequence diagrams, it is taken for granted that the sequence of actions will not return an error; in such a case, an error message describing what happened will simply be sent to the user.

4.2.1 User Registrations

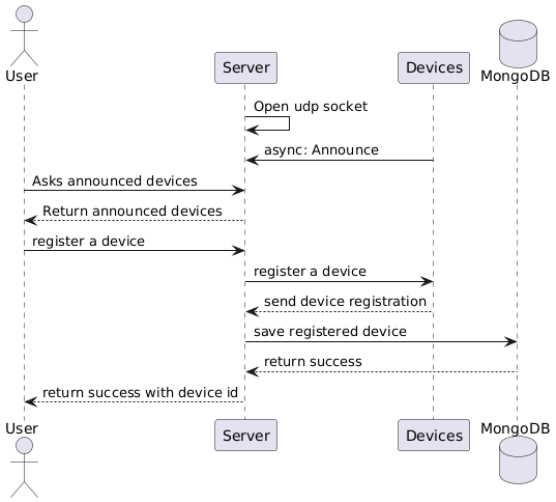
The following sequence diagram shows how a user can send a registration request to the server and then log in when it is approved by the admin.

An exception to be noted is that the first user to register on the server automatically becomes the admin and can log in without the approval of anyone.



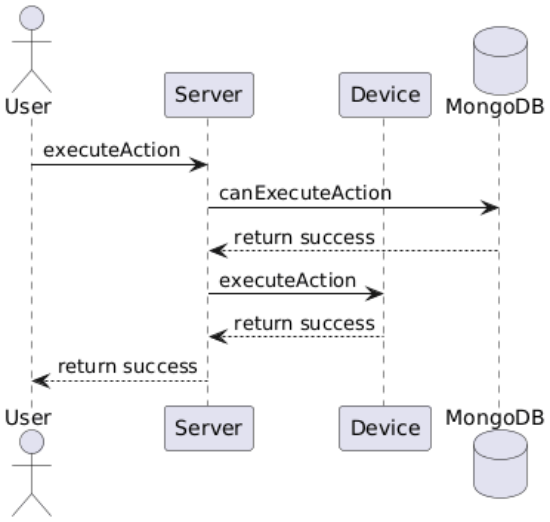
4.2.2 Device registrations

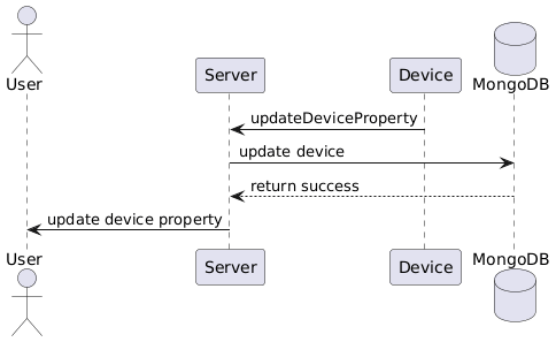
The following sequence diagram shows how a user can add a device to the server (through the "register" message). Note that the "announce" message of the device is sent at a frequency determined by the device.



4.2.3 Device actions and properties

The following two sequence diagrams show what occurs when a registered device changes a property and when a user chooses to execute an action on it. Obviously, a device changes its properties based on its functionality (for a thermometer, it could be the temperature of the room) or the actions of a user (if the user decides to turn on a lamp, for example).

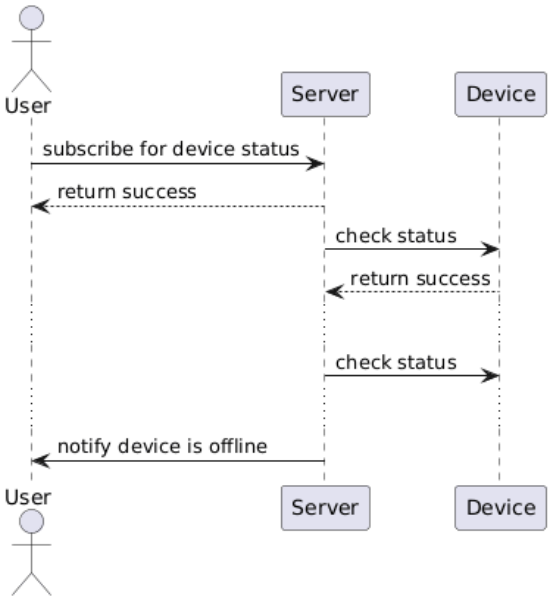




4.2.4 Device offline notifications

In the following diagram is shown how the server manage the "Device is offline" notification to be sent to a user.

The server will send the check-status message to a device with a frequency determined by the server itself and regardless of a user being subscribed to its status change, sending the "Device is offline" notification only to the ones subscribed.



5 Salient implementation details

5.1 Tools

In order to achieve higher software quality and reliability we decided to opt for the following optional tools:

- Typescript (both on the server and the client) to add types in javascript

- Functional programming library Effect-TS (on the server) for our personal preference to use functional programming also in typescript
- Tailwind CSS (on the client) for fast styling of the pages
- DaisyUI components library for Tailwind (on the client) to use pre-defined components in Tailwind

5.2 Devices management

5.2.1 Protocol

It is required that every device while requesting an ip from the LAN DHCP server will also specify a name to bind to that ip. This allows the server to store this name and use it to reach devices even if they change their ip.

Devices announce themselves on a configurable known port by sending broadcast UDP datagrams including basic data (id, name, lan hostname and port) that can be used by the server to register them to the system. The server `DeviceDiscovererUDPAdapter.ts` will keep device announces in memory for a finite amount of time before forgetting them.

The communication protocols expects devices to expose the following http routes on the port that he announced during discovery:

- **POST /register**: The server will contact the discovered device at this route providing the server port that the device will use to communicate with it afterwards.

  The device will respond with a json description of himself (look at DeviceCommunicationProtocolHttpAdapter.ts inline doc for more info).

  From now on the device is registered to the system and is enabled to send updates about its properties to the server.
- **POST /unregister**: The server informs the device that it has been removed from the system.
- **POST /execute/**: The server tells the device to execute a specific action given some input.
- **GET /check-status**: The server will periodically ask the device if it is healthy (any 2xx status code will indicate healthiness).

5.2.2 Real time property update

The DevicesService allows to subscribe for receiving device property updates. One of these subscribers is SocketIOPropertyUpdatesSubscriberAdapter.ts which is a SocketIO server that will will relay the updates to every client that connects to it.

5.2.3 Repositories

All repositories of this bounded context are implemented by extending a generic implementation (look at BaseRepositoryMongoAdapter.ts for more infos)

5.2.4 HTTP Api

Each service has its own http controller which defines routes and http logic (you can look at http adapters).

DTOs are used when the interface to be exposed has to be different from the internal interface of entities. (This is done only due to time constraints, in an ideal scenario it would be better to define DTOs for each entity)

5.3 Notifications Management

The **NotificationProtocol** exposes a **SocketIO** endpoint to which the client can then connect in order to receive the notifications sent by the server (**DeviceOfflineNotifications** or other kind of notifications, like the one sent from the script instructions).

To receive a notification a user must first emits the event "login" sending its email, then the server will send an event "notification" to that user when needed (based on **DeviceOfflineNotificationSubscriptions**, script instructions or script error), sending a message.

5.4 Scripts Management

5.4.1 Script

According to the modelling of the **Script** aggregate, it is an interface that allows users to execute instructions.

Task and **Automation** extends the Script interface in order to have the behaviour described in the modelling part.

When executed, a Script creates an ExecutionEnvironment for that execution (because a script can be started while it is already started, it will create another, fresh one, ExecutionEnvironment every time it is started).

If a Script returns an error after being executed, a notification will be sent to the admin.

5.4.2 Instructions

There are multiple instructions, some which just do some action like waiting, starting other tasks, creating costants, sending a notification or executing device actions, others control the flow of the script, like if or if-else.

- **WaitInstruction**: Stop the script for a specified amount of seconds.
- **SendNotificationInstruction**: Send a notification through the NotificationService to a specified email with a specified message.
- **StartTaskInstruction**: Start another task with the same permissions of the one who started the current task.
- **DeviceActionInstruction**: Execute a specified action on a chosen device.
- **CreateConstantInstruction**: Create a named constant with a value of the specified type.
- **CreateDevicePropertyConstantInstruction**: Create a named constant with a device and a property of that device. At runtime, it will have the value of the specified property.
- **IfInstruction**: Check if it is valid a Condition, if it is true than execute the instructions in the then field. – IfElseInstruction: Check if it is valid a Condition, if it is true than execute the instructions in the then field, otherwise execute the instructions in the else field.

5.4.3 Condition

A Condition is a data structure which contains two ConstantInstructions, a negate field and a ConditionOperator.

It has also an evaluate(ExecutionEnvironment) method, which returns a boolean based on the ConditionOperator, the two ConstantInstructions and the negate field, which, if it is true, switch the boolean returned by the evaluate method (if negate = true and negate should return true, than evaluate returns false)

To be created, the two ConstantInstructions and the ConditionOperator must have the same type.

5.4.4 ConditionOperator

There are multiple `ConditionOperator`s:

- `NumberEOperator`: Equal operator for numbers.
- `NumberGEOperator`: Greater equal operator for numbers.
- `NumberGOperator`: Greater operator for numbers.
- `NumberLEOperator`: Less equal operator for numbers.
- `NumberLOperator`: Less operator for numbers.

- `StringEOperator`: Equal operator for strings.
- `ColorEOperator`: Equal operator for colors.
- `BooleanEOperator`: Equal operator for booleans.

The `evaluate` method of the `ConditionOperator` receives two arguments, the left and right constants, and returns whether the condition is true or false.

5.4.5 ExecutionEnvironment

At runtime, a data structure is needed to store all constants with their respective values; this is the `ExecutionEnvironment`. This is also where the `createDevicePropertyConstantInstruction` stores its runtime values.

Additionally, the `ExecutionEnvironment` contains the token of the user executing the task, if present.

5.4.6 NodeRef

To check the syntax of a script, `NodeRef`s are used. A `NodeRef` contains its `superNode`, which is another `NodeRef`.

There are three types:

- **RootNodeRef**: Refers to the root of the script. Its `superNode` is itself.
- **ThenNodeRef**: Refers to an instruction inside the `then` part of an `if` or `if-else`. Its `superNode` is the node containing the `if`. It additionally stores a reference to the `If` or `IfElseInstruction` that contains it.
- **ElseNodeRef**: Refers to an instruction inside the `else` part of an `if-else`. Similar to `ThenNodeRef`, it stores a reference to the `IfElseInstruction`.

The instruction associated with a `ThenNodeRef` or `ElseNodeRef` is used during script construction to determine where the instruction belongs inside conditional structures.

5.4.7 ConstantRef

A `ConstantRef` contains a `ConstantInstruction` and the `NodeRef` in which the constant is defined.

This is useful because:

- It allows checking whether a constant is used within a scope where it is accessible (similar to variable scoping in programming languages).
- It allows detecting multiple declarations of the same constant within a scope.

In both cases, a syntax error is returned if necessary.

5.4.8 ScriptBuilder

The `ScriptBuilder` is an abstract class that creates a script while validating its syntax.

Build There are two methods used exclusively by the `ScriptsService`:

- `build()`: Creates a script with a randomly generated `ScriptID`.
- `buildWithId(ScriptID)`: Creates a script using the provided `ScriptID`.

Both return an `InvalidScriptError` if:

- the syntax is invalid,
- the script name is empty,
- or an automation with a period-trigger has a zero or negative period.

AddInstructions Each `add[Instruction]` method takes a `NodeRef` and returns a new `ScriptBuilder` instance that contains the newly added instruction and any newly detected `InvalidScriptErrors`.

There are also some `add[Instruction]` methods that return other data structures in addition to the new `ScriptBuilder`:

- `addIf`: Returns a `ThenNodeRef` containing the newly created `IfInstruction`. This allows adding instructions inside the `then` scope of the `if`.
- `addIfElse`: Returns both a `ThenNodeRef` and an `ElseNodeRef`, each containing the newly created `IfElseInstruction`. These allow adding instructions respectively inside the `then` and `else` scopes of the `if-else`.
- `addCreateConstant`: Returns a `ConstantRef` containing the newly created `ConstantInstruction`. This `ConstantRef` can later be used to create `if` or `if-else` instructions.
- `addCreateDevicePropertyConstant`: Works in the same way as `addCreateConstant`, but the created constant is associated with a device property.

5.4.9 ScriptsService

The ScriptsService is the service that manage all the Tasks and Automations.

There are methods to retrieve one or more Tasks/Automations, methods that accept a ScriptBuilder to create or modify a Task/Automation, methods that remove a Task/Automation, a method to start a Task and a method to change the state of an automation between enabled or disabled.

At the start of the server, the ScriptsService will start all the automations created that are enabled, starting to listen for DeviceEvents (for the DeviceEvent triggered Automations) or waiting (for the period triggered Automations).

When removing or editing an automation, the old one will be stopped if there are no errors while doing it.

6 Validation

The development of the server has been done following the TDD metodology.

Unfortunatly, for the Scripts part it was really hard to test everything because there are a lot of different outcomes when it comes to programming (there are Ifs, If-Elses, that can be also nested and with checking of constant declarations etc.), so it was preferred to test them from the GUI interface.

The test of the GUI has been done entirely from the GUI itself. The devices were tested only by the GUI and with means of PostMan. (There were not really requirements about them, so it was preferrable to just test their comunication with the server)

6.1 Passing test

All of the tests are passing correctly and are checked automatically every time a Pull Request is done.

6.2 Amount of tests

There are 34 test suites, with 439 total tests.

6.3 Tests coverage

File	% Stmts	% Branch	% Funcs	% Lines
All files	91.18	92.98	88.81	91.18
adapters	97.56	93.33	100	97.56
BaseRepositoryMongoAdapter.ts	97.56	93.33	100	97.56
adapters/devices-management	98.32	86.36	100	98.32
DeviceGroupRepositoryMongoAdapter.ts	100	100	100	100
DeviceRepositoryMongoAdapter.ts	97.99	84.21	100	97.99
adapters/notifications-management	100	100	100	100
DeviceOfflineNotificationSubscription.ts	100	100	100	100
adapters/permissions-management	83.52	87.09	83.72	83.52
EditListRepositoryMongoAdapter.ts	67.12	85.71	75	67.12
TaskListsRepositoryMongoAdapter.ts	94.49	86.36	92.3	94.49

UserDevicePermissionRepositoryMongoAdapter.ts	95.87	89.47	85.71	95.87
adapters/scripts-management	98.1	97.56	100	98.1
ScriptRepositoryMongoAdapter.ts	98.1	97.56	100	98.1
adapters/users-management	100	100	100	100
RegistrationRequestRepositoryAdapter.ts	100	100	100	100
UserRepositoryAdapter.ts	100	100	100	100
domain/devices-management	97.72	94.05	97.45	97.72
Device.ts	99.57	94.59	100	99.57
DeviceActionsServiceImpl.ts	100	100	100	100
DeviceEventsServiceImpl.ts	100	100	100	100
DeviceGroup.ts	100	100	100	100
DeviceGroupsServiceImpl.ts	99.5	96.49	90.47	99.5
DeviceStatusesServiceImpl.ts	98.26	95.83	100	98.26
DevicesServiceImpl.ts	99.27	97.61	100	99.27
Types.ts	89.85	80.39	96.87	89.85
domain/notifications-management	94.76	96.66	77.77	94.76
DeviceOfflineNotificationSubscription.ts	100	100	100	100
NotificationsServiceImpl.ts	94.19	96.55	76.47	94.19
domain/permissions-management	75.32	96.45	72.91	75.32
EditList.ts	100	100	100	100
PermissionsServiceImpl.ts	71.46	95.86	67.9	71.46
TaskLists.ts	100	100	100	100
UserDevicePermission.ts	100	100	100	100
domain/scripts-management	90.15	89.23	88.33	90.15
Instruction.ts	100	100	100	100
InstructionImpl.ts	99.64	93.75	100	99.64
Operators.ts	100	100	100	100
Refs.ts	100	100	100	100
Script.ts	100	100	100	100
ScriptBuilder.ts	91.33	70.66	100	91.33
ScriptsServiceImpl.ts	75.55	88.88	57.14	75.55
Trigger.ts	100	100	100	100
domain/users-management	91.84	92.5	90	91.84
RegistrationRequest.ts	100	100	100	100
Token.ts	100	100	100	100
User.ts	100	100	100	100
UsersServiceImpl.ts	89.19	90.9	85.18	89.19
ports	100	100	100	100
Repository.ts	100	100	100	100
ports/devices-management	100	100	100	100
Errors.ts	100	100	100	100
Types.ts	100	100	100	100
ports/permissions-management	53.65	100	40	53.65
Errors.ts	53.65	100	40	53.65

ports/scripts-management	100	100	100	100
Errors.ts	100	100	100	100
ports/users-management	88.13	100	85.71	88.13
Errors.ts	88.13	100	85.71	88.13
utils	100	100	100	100
MongoDBErrorCodes.ts	100	100	100	100

7 CAP Theorem

7.1 Partitioning

Our system heavily relies on the server part, which usually would not be a really good choice, but in our scope the server **runs locally** on a computer, without being hosted on internet (and, even if so, a local server needs always to exists, making a remote server a sort of a "proxy" of the real one), removing then all the implications of the net (like being unable to contact the server without internet). The server can still run even if there are not devices available or even if some device is offline/broken.

7.2 Availability

The **Availability** is really well achieved in our system because every time something fails (like a device not reachable by the server of a device going online) the user interested is notified, either by a notification or a popup.

The server find offline devices sending a request of "check-status" every 5 seconds, if they answer it with status code 202 they are healthy, otherwise either they answer with an error (specifying the cause) or don't answer at all (they are offline).

7.3 Consistency

Our system is also really well Consistent because we use MongoDB API for the DB with one replica, and at the application level, each instruction (add, get, getAll, remove, update) ensures that it does only if in the right circumstances.

1. It is possible to add an element just if it is not a duplicate by checking a key and sometimes even a name.
2. It is possible to get an element just if an element with that key already exists.
3. It is always possible to get all the elements (if not present, it just returns an empty list).
4. It is possible to remove an element just if an element with that key already exists.
5. It is possible to edit an element just if an element with that key already exists and sometimes it is checked also the uuniqueness of its (maybe) new name.

In all these cases, the user is notified that something went wrong with a pop-up.

7.4 Scalability

Unfortunatly, the server (also due to the low partitioning and the same reasons above) is not easy to be scaled, but usually being it a server used locally just for the devices inside an house and the users inside an house (about 4/5 for family or 20/30 for agencies), the **scalability** we think is an aspect of a less importance respect to the **availability** and **consistency** (an user doesn't want to think that a device is doing something when it does not, or viceversa).

7.5 Server replicas and fault tolerance

With kubernetes we tried to make **server replicas**, but there were problems were the devices where comunicating updates to random replicas and not just one of them (the one comunicating also with the GUI), we tried with the "sessionAffinity: ClientIP" configuration but it was not working, so at last we decided to stuck with only one replica of the server.

We think that something that could have been really good to do for fault tolerance is deciding which replica is the "main" one and which are the ones to be used only in case of a failure, maybe in the application level.

8 Deployment Instructions

For the deployment of our application we decided to use Kubernetes.

To run the system, go to the server repository, then here there are some instructions to be executed.

1. To make the changes you do on the server persistents execute the following command:

   ```
   kubectl apply -f ./k8s/persistence
   ```

2. To start the server and the DB execute the following command:

   ```
   kubectl apply -f ./k8s/deployment
   ```

3. To start the devices (these are just to test the server without having real devices):

   ```
   kubectl apply -f ./k8s/devices
   ```

To stop the server and/or the devices use one or both of the following commands:

```
kubectl delete -f ./k8s/deployment
kubectl delete -f ./k8s/devices
```

To reset the DB use the following command:

```
kubectl delete -f ./k8s/persistence
```

9 Usage Examples

The GUI is a website made in Vue + Html + Css.

We have prepared a full demo to allow you to test it. In this demo there are already users, devices and tasks/automations ready to use. To run it, this is the command:

```
docker compose up -d
```

Once the Docker containers are all up and running, open your browser and go to `http://localhost` to view the website.

1. **Login** with an existing user/ register with a new user (you have to wait the approval of an admin in this last case to make the login)
2. In the **Device** page (the first to appear) you can view the devices and interact with them if you have permissions to do so.
3. You can then go to other pages from the tabs on the top bar of the application: **Tasks**, **Automations**, **Notifications**, **Settings**, and **Admin** only if you are an admin.
    a) The **Admin** tab is a topdown menu where you can go to different administration pages like: Users, Devices, Devices Groups, User Device Permissions and Scripts permissions
4. In the **Notifications** page you can see all the notifications sent to you, and can possibly delete them (they are not stored in the server, so they are only available to see in that session)
5. In the **Settings** page you can modify the nickname and the password (the email cannot be modified)
6. In the **Tasks** page you can execute an already existing task (if you have the right permissions to do so), create a new task or edit an existing one (if you have the right permissions to do so).
    a) If you want to create a task, click on the add button on bottom of the page. If you instead want to edit an existing one, you should click on the row of that task (this let you go on read-only mode) and then click to edit in the top bar (edit-mode).

b) Set a name for the task.

c) Put all the instructions you want to put inside the task.

d) You can move instructions up/down by means of the arrow buttons at their left, or remove them with the cross button at their right.

e) Save to actually create/edit the task.

7. In the **Automations** page you can enable/disable automations (if you have the right permissions to do so), create a new automation or edit an existing one (if you have the right permissions to do so).

   a) The process of creating/editing an automation is almost the same of the tasks, but with one difference.

   b) You must create a trigger for the automation with the button under the name of the automation, either period trigger or device event trigger. The **period trigger** indicates when and how often the automation should fire. The **device event trigger** indicates which device event will fire the automation.

   c) You can then modify the parameters of the triggers like you would do with an instruction.

8. In the **User** administration page you can remove some users from the server or accept/remove the requests of new users.

9. In the **Device** administration page you can modify the name of already registered devices, remove them from the server, or add new devices.

   a) To add new devices, click on the bottom button of the page.

   b) Click the "plus" button on the right of the devices you want to add.

10. In the **Device Groups** administration page you can modify the name of the device groups, create new/modify groups.

    a) To add/remove devices to a group, click on the row of that group.

    b) You now see all the devices on the server, to add a device click on the right button ("plus" button) of the devices under the "Device not in group" title.

    c) To remove a device, click on the "cross" button at their right.

11. In the **User Device Permissions** Administration page you can edit the permissions of the users on the devices.

    a) To modify the permission of a user, click on the row of that user.

    b) Click on the "cross" button to the right of a device to remove it.

    c) Click on the "plus" button to the right of a device to add it.

12. In the **Scripts Permissions** page you can edit the scripts permissions of the users.

a) Click on a task or an automation to edit the permissions that users have on that task/automation.

b) For a task, there are three lists, Whitelist, Blacklist and Editlist, for an automation there is only the Editlist one.

c) To add a user to a list, click on the "plus" button to their right.

d) To remove a user from a list, click on the "cross" button to their right.

In almost all of the pages there is an "info" button that explains exactly what you can do in that page (like what instructions do and why).

10 Conclusions

10.1 Future Works

We think in the future some works that can be done are:

1. Make server replicas works with the logic we described above.
2. Add more types of data that devices can manage, like streams (for videocameras or music players, and so on so forth).
3. Notify also when a device comes back online (We think this is the only thing of availability not being present, but it should be really easy to add).

10.2 What did we learned

Being a really big project, we learned really lot of things, like:

1. Co-working with teammates.
2. Thinking about Distribuited aspects (primary Availability and Consistency).
3. Using Docker and Docker Compose for the development and testing of the application.
4. Deploying an application with Kubernetes.
5. Creating emulations of real world IoT devices.