pharaon

PILOTS FOR HEALTHY AND ACTIVE AGEING

Grant Agreement: 857188

Second Pharaon Open Call - Technical information

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No 857188.

Contents

Figures

Tables

Acronyms & Abbreviations

Term	Description
SSO	Single sign-on
OIDC	OpenID connect
CI/CD	Continuous integration and delivery
UI	User interface
BPMN	Business process modelling notation
JWT	JSON Web token
NGSIv2	Next Generation Service Interfaces version 2 (Fiware context broker API standard)
PKCE	Proof key for code exchange

1 Introduction to Pharaon Hub

1.1 Overview

This document describes how to integrate technical components in order to comply with Pharaon integration standards. Components are integrated with the help of a set of integration standards and tools, which we call the Pharaon Hub. It is built on top of widely used interoperability standards: OpenAPI and OpenID Connect (OIDC). The Pharaon Hub integration tools support use of these standards as much as possible.

This document will be updated if and when new changes are introduced to the Pharaon Hub.

1.2 Pharaon reference architecture

The Pharaon reference architecture describes the placement of various technical components and facilities within the Pharaon system. It covers five layers: collaboration / processes, application, service, platform, and device / network. Additionally it covers management, interoperability, and security and privacy aspects of each layer.

The Pharaon Hub, with its reliance on OpenAPI and OIDC, concentrates on service interoperability, but also covers any of the other layers and aspects as needed. Figure 1 shows the generic reference architecture overview defined in the first phase of the project, overlaid with the current coverage. This includes integration tools, CI/CD facilities, and the developer's manual and resource catalogue.

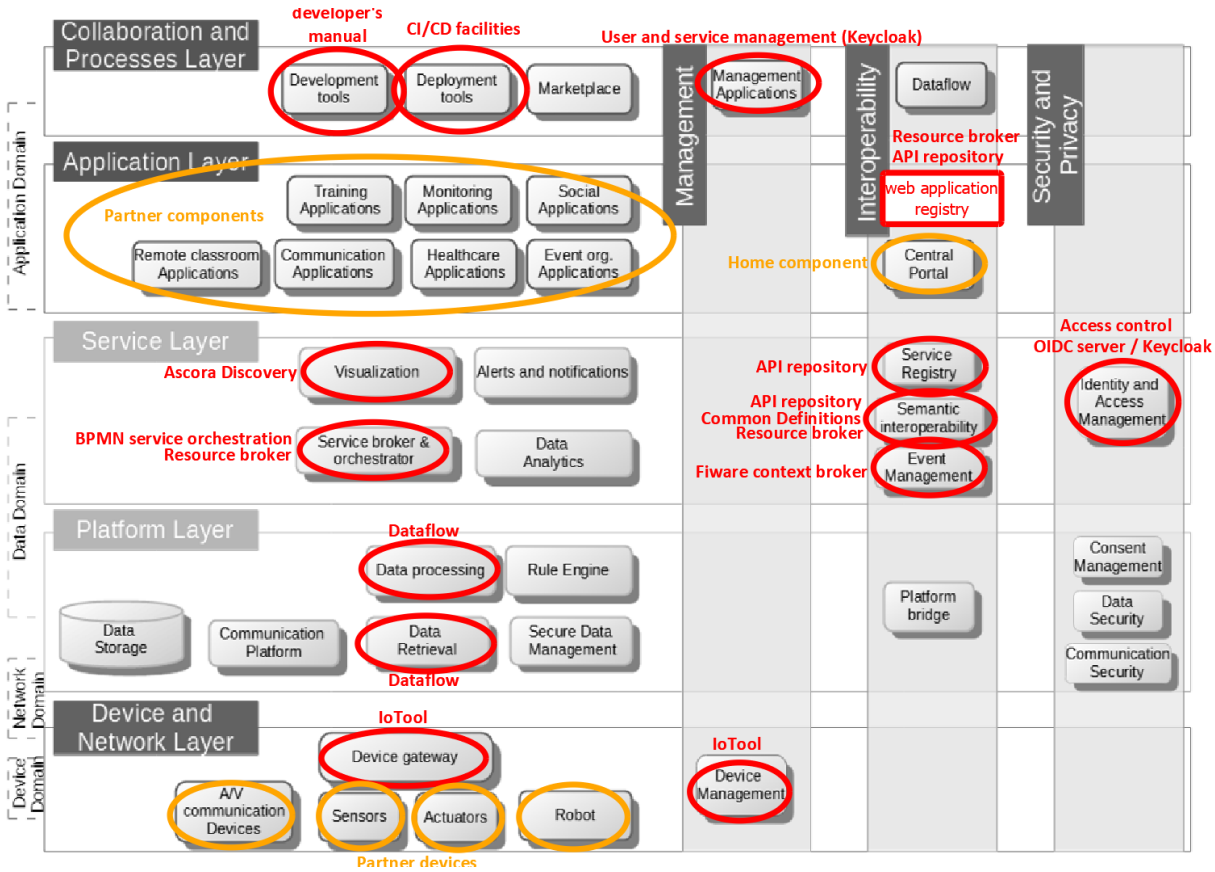


Figure 1 Pharaon reference architecture and its coverage by current Pharaon Hub components.
Red: functionality covered by Hub integration tools.
Orange: functionality covered by Pharaon technologies and OC2 applicant.

1.3 Pharaon Hub tools and facilities

In this section we describe the Pharaon Hub facilities in more detail. We currently have the following facilities (see Figure 0.2):

- API repository. This is a database that stores API metadata, including a link to the OpenAPI spec of each API. In Pharaon we also define Web applications in OpenAPI, to make them findable in the same way as APIs. The repository is managed through a Web portal. It has a basic API for accessing the data in the database. APIs can be annotated with metadata using the OpenAPI extension scheme, via a set of guidelines we formulated for Pharaon (see section 2.4). For example, a function returning a number can annotate the return value with the SNOMED (https://www.snomed.org/) code for heart rate, indicating the meaning of the value. Additionally, an API can be annotated with organisational, location, and language information, so that for example geographical, legal, and language constraints are explicitly formulated. Besides storing APIs, the API repository also has:
    - A common definitions facility for defining common data structures
    - A search function for finding APIs and services by content or metadata. This can be used to connect service users to service providers without hard-coding, and implement plugin schemes.
- Service orchestration. The service orchestration tool uses business process modelling notation (BPMN) to specify processes. BPMN schemas can be edited using the included visual

programming interface. They are executed using Camunda. Schema steps can call API endpoints specified in the API repository. An API is available for starting Camunda processes.

- Fiware context broker. The context broker manages a set of entities (for example, rooms). Each entity has properties (like sensor values). It provides an API endpoint for storing, retrieving, and subscribing to data. Fiware also enables use of other standard components like an IoT bridge that enables reading out particular types of devices.
- Dataflow. Provides a background service that can collect and aggregate data from various sources. Via a visual programming user interface you can specify one or more dataflows that automatically tracks data from particular data sources and stores it in a target database.
- CI/CD facilities. This includes installations of Jenkins, SonarQube, Harbor, Prometheus and Grafana.
- API gateway and Web integration facilities. In order to help with integration, we provide a custom built Pharaon API gateway, and a software library with Web components to help with integration.
- General developer resources, including general developer guidelines, integration guidelines, and usage of third party API gateways.

In our approach, everything is connected via an OIDC server. This server also manages user data, including things like real name and preferred language. Our choice of OIDC server, Keycloak, provides login, logout, and user management user interfaces, and a user management API. The basic OIDC setup is that all user interfaces perform SSO via the OIDC server, and all API calls include an OIDC access token that is verified by the callee via the OIDC server. Additionally we have a scheme for inter-user access control, implemented via standard Keycloak features.

An overview of the setup is given in Figure 2. It shows the Pharaon applicant platform and Pharaon technologies (orange), the Pharaon Hub integration tools (green) and the base services (blue). Some examples of interactions are given. The purple lines indicate an example process flow of the service orchestration. Technology 1 starts a particular BPNM process, which then calls an API endpoint in Technology 2, and returns the result. Technology 2 passes its auth token to the BPNM process, which passes it along subsequent API calls. The dotted gray lines indicate SSO and token verification with the OIDC server.

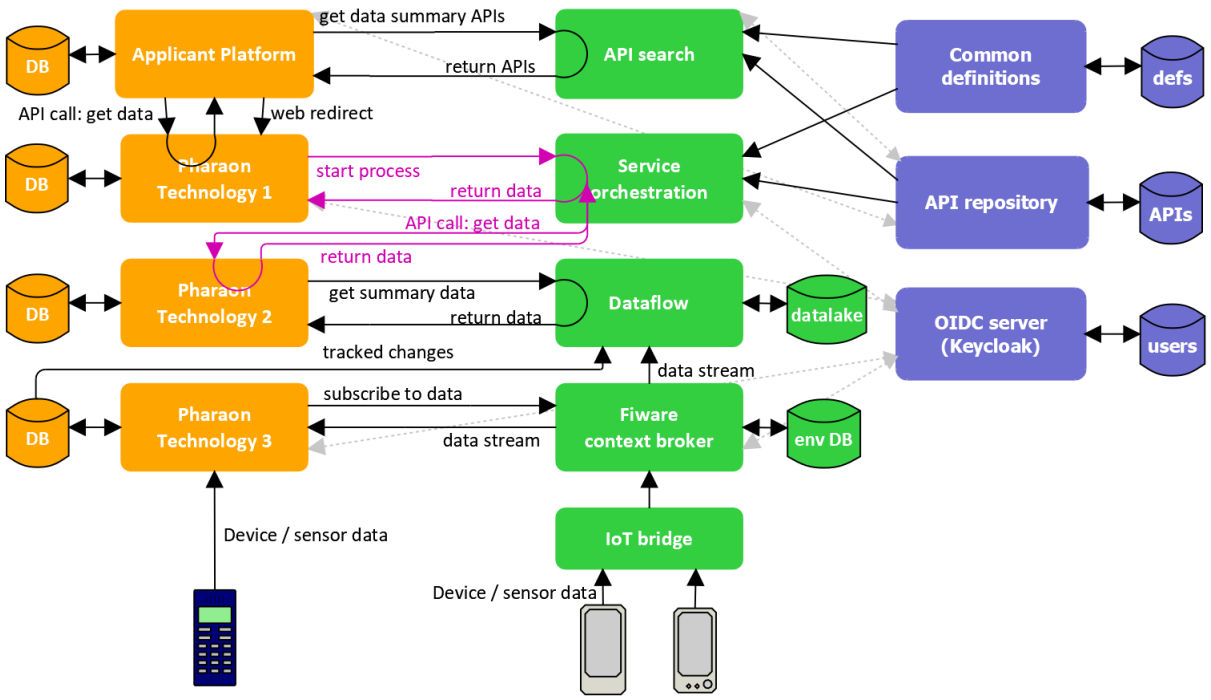


Figure 2. Pharaon Hub facilities overview.

1.4 Semantics and metadata

In Pharaon we have a data-centric approach, which means data is designed to outlast particular data providers or users, by making the data and definitions persistent and adding sufficient meaning to the data. In the Pharaon Hub, shared data definitions can be semantically annotated and stored in the common definitions facility. The definitions are JSON schemas for use with OpenAPI accessible via the $ref reference construct. In addition to data, our approach also enables APIs and services to be annotated with semantics.

We prescribe a specific way to annotate JSON schemas and OpenAPI definitions, enabling specification of things like semantic data types, endpoint semantics, and API contextual metadata. Our primary approach is to add computer readable metadata to OpenAPI, complementing the already available human readable metadata, via the use of ontologies and coding systems. We use a simple ontology paradigm based only on concepts. We define no relationships between concepts at this point. Each concept is encoded by a unique “canonical” URL. Preferably, the URL points to a human readable description of the concept. In this way, the concept definition can double as a documentation reference, and conversely, any stable, canonicalized documentation URL can be used to indicate a semantic concept. In this scheme, machine readability is simply recognising that each particular URL represents a distinct concept, enabling semantically enriched search. If a class hierarchy is needed, it can be encoded in the URL path, making it possible to use a path wildcard to represent a family of classes.

A concept ontology can also be defined using a wiki (Hepp et al., 2007) . The wiki authoring interface provides a low barrier of entry, so it can be maintained by the entire stakeholder community, rather than just by ontology specialists. Community wikis are available, like Wikipedia, that provide access to

a complete and evolving, yet relatively stable set of concepts. Some wikis enable referring to specific versions of pages via permalinks. This is ideal for ontology descriptions, as it enables them to have version control. Some wikis also allow pages to be organised in subfolders, enabling the creation of the aforementioned class hierarchy. Recommended ontologies and coding systems are found in Table 1, though any system that uses URL coding can be used.

Table 1 Recommended URL- based coding systems

Ontology/coding system	**Domain**	**URL format / example**
ContSys/ISO 13940	Healthcare	https://contsys.org/concept/medical_device
Snomed	Medical	http://snomed.info/sct/{code}/version/{timestamp}[1]
Loinc	Medical	https://loinc.org/41950-7/
Wikipedia	General	https://en.wikipedia.org/wiki/ISO_8601
Pharaon concept wiki	Technical	https://gitlab.com/pharaongroup/concepts/-/wikis/resource/webpage/dashboard

The Pharaon concept wiki defines Pharaon-specific technical concepts. Following the OpenAPI structure, it distinguishes three different semantic base classes: services, resources, and data. A service refers to an entire OpenAPI definition, defining a coherent collection of URL patterns, like particular functions and Web application entry points. An example would be a database access standard. A resource refers to a single HTTP method in the OpenAPI definition, which can be either an API function or a Webpage. An example would be a database read function for specifically getting data within a given time range. Data refers to JSON schemas, which can be used to describe function parameters or return values. Examples would be the aforementioned Temperature or User values. The tree of concept classes is shown below.

```
├──data – top level for data elements and structures
├──service – top level for API and Web app specifications
│    ├──webapp
│    └──API
└──resource – top level for a single OpenAPI endpoint
     ├──webpage
     └──API function
```

Our approach enables any part of an OpenAPI definition to be annotated with concept metadata, via the keyword x-def. We provide a number of additional keywords for specific metadata, like component ID, organization, country, language, and display information. More details are found in section 2.6.

[1] The Snomed URL scheme is documented here: http://snomed.org/uri

2 How to integrate with the Pharaon Hub

2.1 Overview of integration requirements

In addition to integration with particular technologies available for OC2 integration, we require applicants to integrate according to the following set of technical requirements:

- **OpenID Connect** (OIDC) has to be implemented. We will provide a function library and API gateways to help with this. More details are found in section 2.2. An overview:
    - All application user interfaces should use authorization code flow for single sign-on. All integrated end user APIs should be able to use the obtained token for verification and identifying basic user information (unique ID, real name, preferred language, etc, all of which can be done with the userinfo endpoint).
      If token validation fails, your API endpoint should produce a 403 error.
    - Client credentials flow should be used for machine to machine communication between two applications. Integrated machine to machine APIs should be able to verify the token and distinguish it from the authorization code token.
    - Use our specific access control method to determine if one user has access to another. We currently identify the following two scenarios (:
        - Group membership. HCPs belonging to a group have access to all patients in the group.
        - One-to-one access: a user can have access to one or more specific other users.

          For simplicity, we do not currently provide more fine-grained access control, such as access to specific resources of a user.

      In order to make OICD integration easier, we provide support for using particular API gateways that can be used to manage authorization, in particular the ICE API gateway (developed in--house) and the Kong platform (third-party software).
- Use **OpenAPI** to define APIs of your own components or call APIs in other Pharaon components.
- **Register your APIs in the API registry**, so they can be found by the hub integration tools, in particular service composition and service search. We provide a scheme for annotating APIs with metadata to make them more searchable, so they are usable for service discovery and plugin schemes. See section 2.2 for more details. Note that not all data structures need to be fully specified for the OpenAPI tools to function.
- Use the **H**ub **service search** to obtain other Pharaon components' APIs. The minimal requirement is to search by component ID ("service discovery" style) to avoid hard-coding URLs.
- For Web-based applications, use any Web style integration facilities we provide.

We recommend the following requirements to applicants, which, if implemented, would positively impact the application evaluation:

- Use the service composition tool, in case data is integrated from multiple API functions
- Fiware context broker and related tools

- Use an API gateway for software that is difficult to integrate directly with Keycloak/OIDC, or for federating external services.
  We provide two solutions: the ICE API gateway and Kong, but they can use another gateway if it fits their use case better. See section 2.3 for more details.
- Follow the Continuous Integration and Continuous Delivery (CI/CD) process and the Quality Assurance (QA) approach defined within the PHArA-ON project, based on the DevOps principles. The PHArA-ON project offers an integration and testing infrastructure that can be used to automate (as much as possible) the different DevOps's phases (i.e., build, package, stage, test) defining and running CI/CD pipelines. The CI/CD tools available within the PHArA-ON integration and testing infrastructure are: the Jenkins CI/CD server used to define and execute pipelines, Sonarqube to scan code for code analysis and detect vulnerability, Harbor as the registry of docker images. Moreover other additional services are available such as Keycloak for identity and access management, Prometheus and Grafana to collect and visualise monitoring data from the applications, Postfix and Dovecot to allow applications to send and receive emails. A GitLab repository is available to store code that will be checked during the pipelines execution. If the components cannot be integrated using project-level tools (e.g., no open source code and/or no binary artefacts available) private integration environments or "hybrid" pipelines (using private and project-level tools) are allowed to follow the PHArA-ON release process and guarantee the same quality levels. We also recommend performing testing activities during the integration phases in order to release high-quality software. The PHArA-ON approach is used on a multi-level testing considering both functional (e.g., unit tests, integration tests, UI tests) and not-functional (e.g., load, performance, monitoring) dimensions than can be performed in an automated way (via the CI/D pipelines defined above) or manually if the automation is not feasible.
- Follow the developer's handbook - available to registered developers through private Pharaon Gitlab at https://gitlab.com/pharaongroup/developers-handbook), contains getting started guides, Pharaon technologies details including integration instructions, project-wide guidelines, best practices, focused survey and tools analysis, recommended open-source tools to serve many different technical activities of the Pharaon partners who are adapting, integrating, testing, and validating their technologies within the Pharaon pilots.

2.2 Authentication and access control

Single sign-on (SSO) is performed via Keycloak OIDC. In our approach, OIDC tokens are used for direct exchange between technologies (e.g. via API endpoints), and technologies are responsible for their own (data) security. Also, we use Keycloak to store global personal data, like real name and preferred language.

We require that user interfaces (Web applications and apps) use the OIDC authorization code flow to obtain a token associated with the currently logged in end user. This is the most common auth flow, and is suitable for end users using browser technology. If the user is not logged in yet, Keycloak shows a login screen. See section 2.2.1 for more details.

For server-to-server authorization, we recommend using the client credentials flow with Keycloak. This flow consists of a single call, returning a JWT that can be passed to another server, which can verify it

using Keycloak's public JWT key. See the appendix for an explanation of the client credentials flow. See section 2.2.2 for more details.

In both cases, the token is a JSON Web token (JWT) that contains some useful basic info, like the user ID and group membership information. The user ID is in UUID format, and can be used as a stable ID within your own technology. Further details on what will be in the JWT is still to be determined.

2.2.1 Authorization code flow

OIDC handles end user SSO via the authorization code flow. Details are described in this section. Four OIDC endpoints are particularly important:

- Authorize: is just a URL, must be loaded in browser. Mobile apps or PC applications can redirect to a browser, and can redirect back to the app using a custom URL scheme.
    - Auth server shows login screen if not logged in, and optionally asks the user for consent
    - Auth server will then redirect to a fixed client return URL, with a code as parameter
- Token: call this with client ID, client secret, and code, to get a token. Call must be server-to-server, otherwise add PKCE for extra security.
- Userinfo: call this to verify a token and get user ID and other info, like name, email, access role, preferred language, and other fields.
- Logout: load in browser, logs user out of auth server.

You should use the following basic approach.

- Auth flow starts when loading a client Web page for the first time, or starting a client app. Web page loads should verify the obtained token using the userinfo endpoint.
- Tokens expire after a number of minutes or when user logs out, so restart the auth flow if a token is invalid.

Alternatively, the OIDC refresh token can be used to get a new token (more code needed but more efficient).

- When one client calls the API of another, it passes its own token.
- OIDC Clients should verify the caller's token via the userinfo endpoint, for every API call. The auth server then provides the associated user ID and access role.
- Keycloak admin and API can be used to edit personal data.
- For Web apps, there are two distinct cases: server-side and client-side. A Web application is client-side when it calls OIDC endpoints that are normally server-to-server from within the browser. In that case, secrets in the browser code are exposed to the end user, and additional security measures are needed to secure the application, in particular Proof Key for Code Exchange (PKCE) verification.

Figure 3 specifies a detailed example process flow. It illustrates login, authorization code flow, and getting data via an API of another component to show a graph to the end user. The sequence diagram includes full details, like browser redirects and token and browser session cookie exchange.

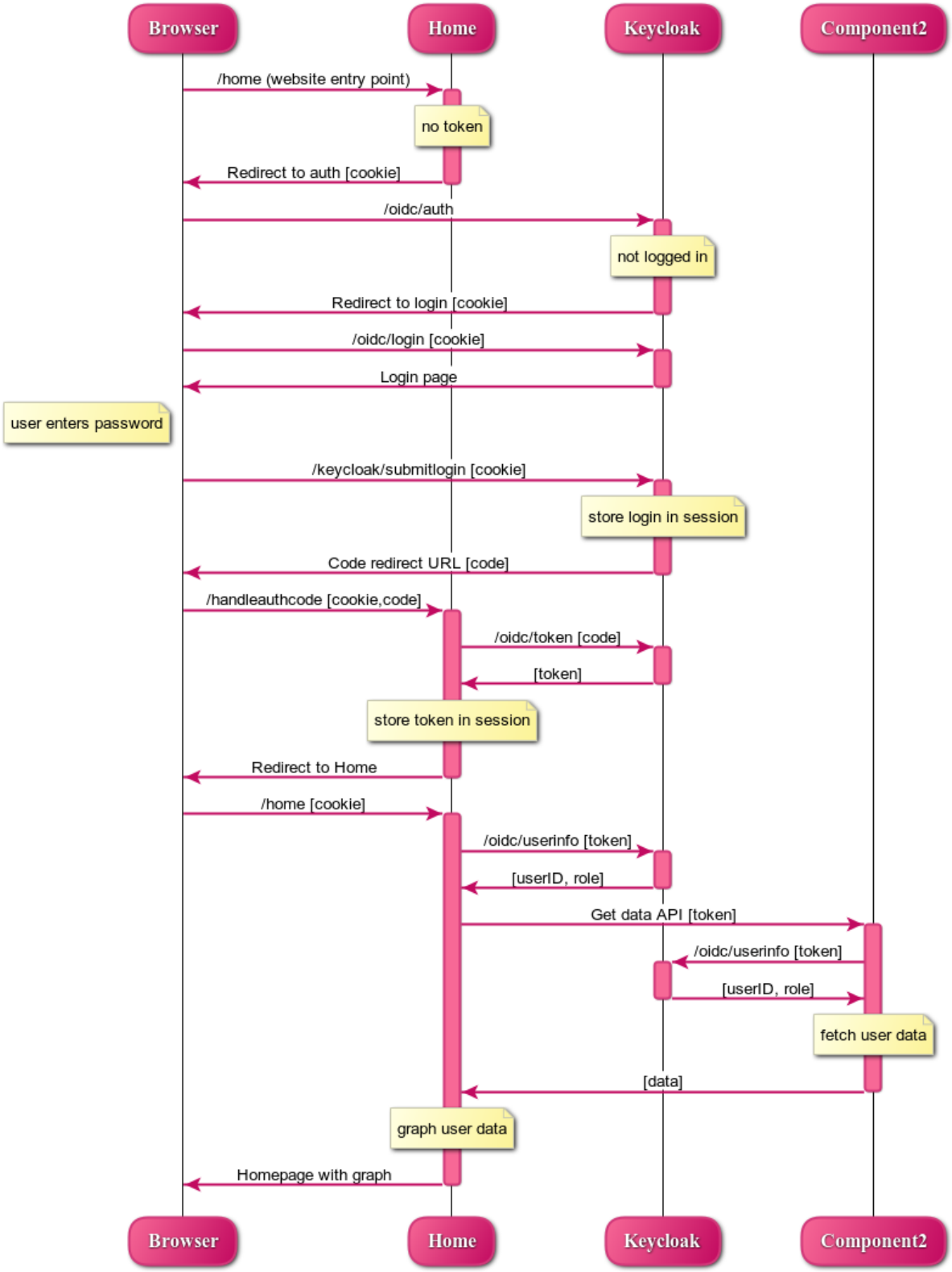
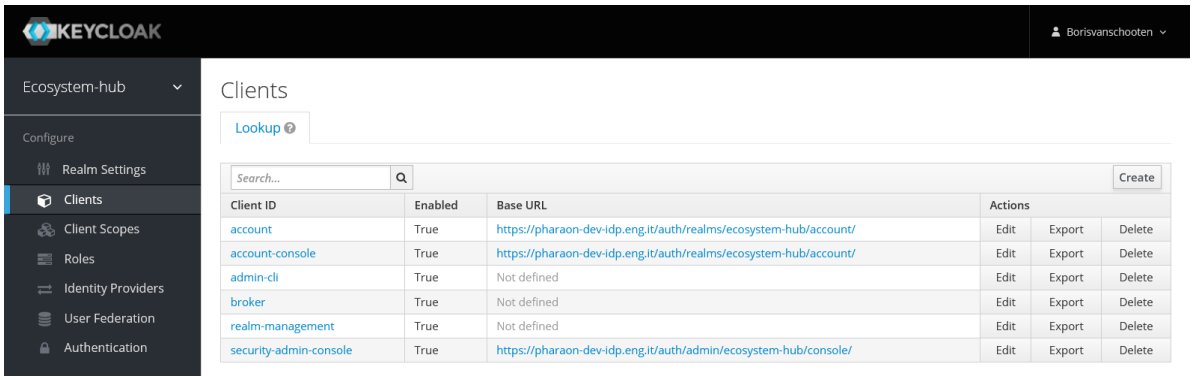
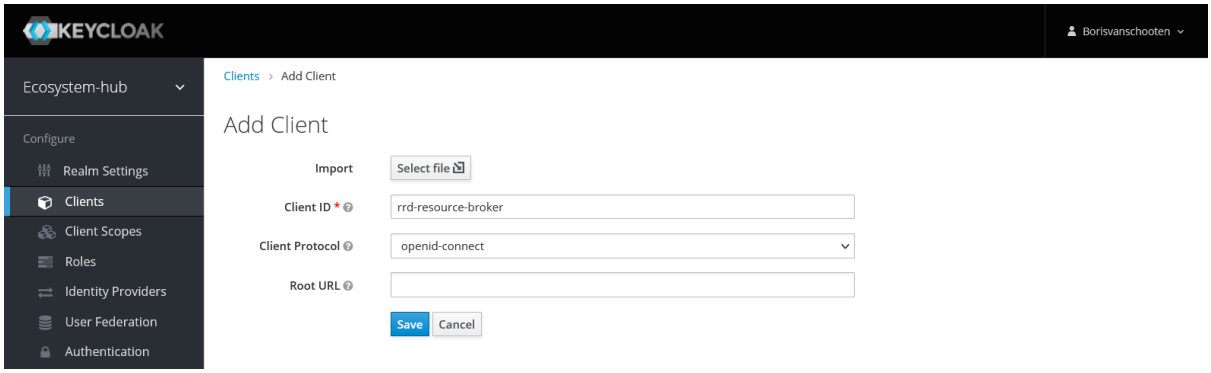


Figure 3. Sequence diagram of example OIDC scenario. [cookie] indicates session cookie; [token] indicates OIDC token; [role] indicates user access role.

Keycloak can be configured to work with your software as follows. Log into Keycloak with your realm admin account and select Clients on the left. Now, create a new OIDC client with the Create button on the right. This client represents your software component.

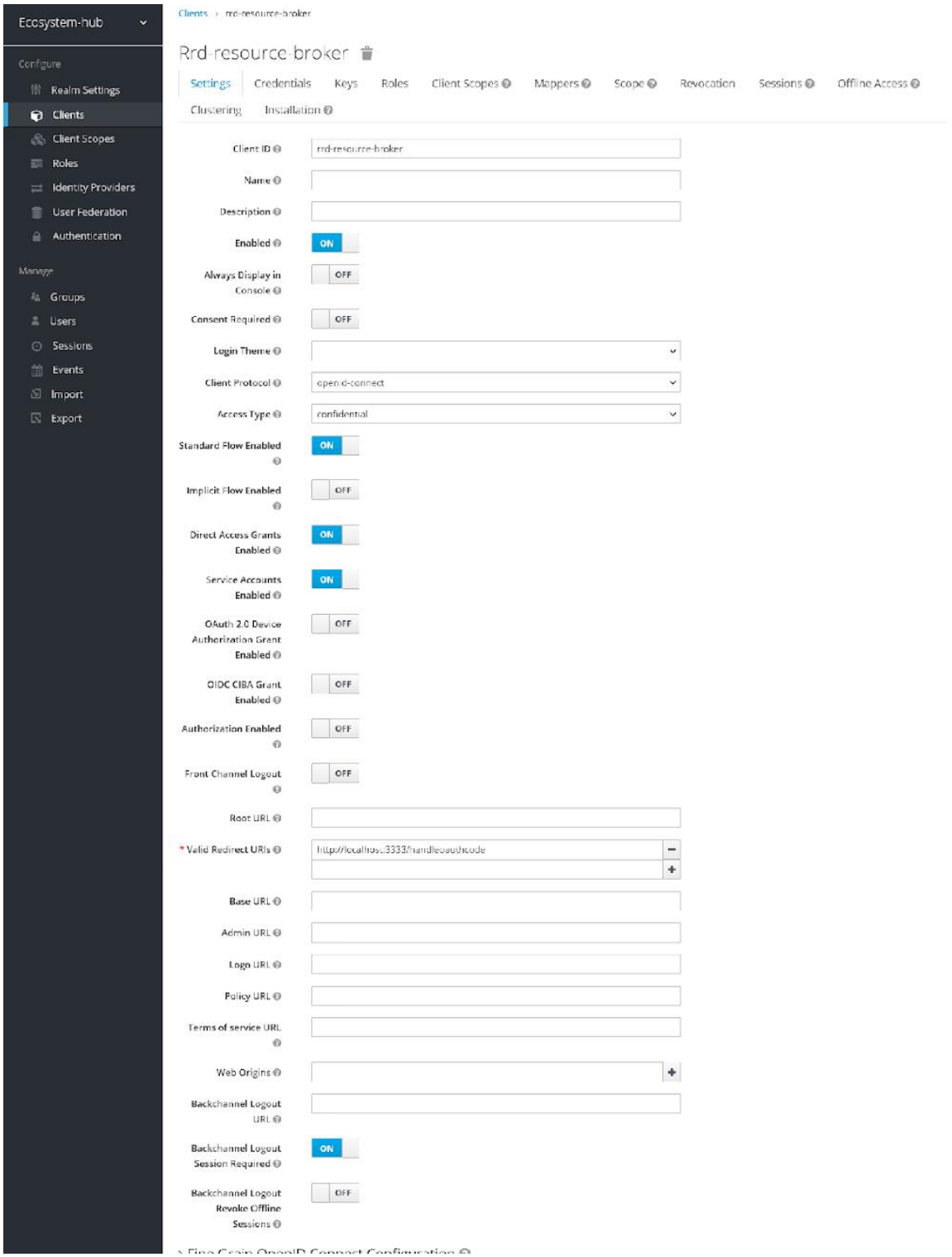


In the add client form, fill in your desired client name. For client protocol, select openid-connect. You don't have to fill in the root URL.

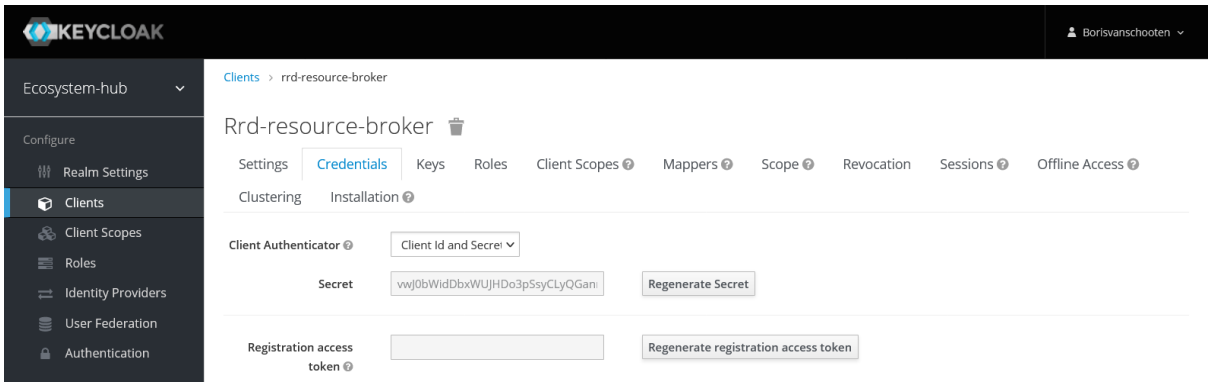


Now, you get a bigger form where you can configure the client.

- Make sure that Standard Flow is enabled for Authorization Code flow.
- If your Web app is server-side, select access type = confidential, which provides extra safety via a client-specific secret. If your Web app is client-side, the secret is not useful for security, so you can select access type = public.
- You will also have to fill in the Valid Redirect URIs field. Here, you fill in the URL you use in your client to handle the auth code returned by Keycloak (the /handleauthcode step in Figure 3).
- You can leave the other fields on default. Now, save the changes.



If you selected access type = confidential, there should now be a Credentials tab. In this tab is the client secret that you have to supply in the first step of authorization code flow. Also, the Service Accounts Enabled option appears. Enable this to enable the client credentials flow.



If your software component is a client-side Web app, you can provide extra security via PKCE verification, available via advanced options. For more info about setting up PKCE, see for example: https://www.appsdeveloperblog.com/pkce-verification-in-authorization-code-grant/.

Keycloak is now configured. The authorization code flow can be started by redirecting the browser to the auth endpoint. Note that this URL has to be loaded in a browser, giving Keycloak the opportunity to show a login screen if needed.

https://[KeycloakServer]/auth/realms/[YourRealmName]/protocol/openid-connect/auth?client_id=[ClientId]&response_type=code

Note that for some versions of Keycloak, the /auth at the beginning has to be omitted. Optional parameters are state, scope, and redirect_uri. If no redirect URI is supplied, Keycloak will redirect the browser to the first valid redirect URI you have configured. If Keycloak accepts your request, the redirect looks like this:

http://[YourServer]/handleoauthcode/?session_state=[UUID-style-hash]&code=[AuthorizationCode]

This assumes that your server's authorization code redirect handler is found at "/handleoauthcode". If you supplied a state parameter, the state value will be echoed back to you in this callback, so you can check if this callback actually corresponds to the original call. This is not needed when using a session cookie to store browser session state.

Now, you can get an access token using the supplied AuthorizationCode. Note this is a server-to-server POST call, with parameters supplied in the POST body. We can do this call with Curl:

*curl --location --request POST 'http://[KeycloakServer]/auth/realms/[YourRealmName]/protocol/openid-connect/token' *
*--header 'Content-Type: application/x-www-form-urlencoded' *
*--data-urlencode 'grant_type=authorization_code' *
*--data-urlencode 'client_id=[ClientId]' *
*--data-urlencode 'client_secret=[ClientSecret]' *
--data-urlencode 'code=[AuthorizationCode]'

If the credentials are valid, the response body contains token info in the form of a JSON string:

{
"access_token": "[AccessToken]",
"expires_in": [ExpiryTimeInSeconds],

"refresh_token": [RefreshToken]
"refresh_expires_in": [RefreshTokenExpiryTimeInSeconds],
"token_type": "bearer",
[…]
}

The most important thing is the access_token. You can store it in your browser session. Access can now be verified server-to-server by calling the userinfo endpoint. This is a GET call that needs a header field *"Authorization: Bearer [AccessToken]"*:

https://[KeycloakServer]/auth/realms/[YourRealmName]/protocol/openid-connect/userinfo

If the token is valid, you get a 200 response with a JSON body containing user information:

{
"sub": "[KeycloakUserId]",
[…]
}

If the token is invalid or expired, you get a 40x error response. The simplest thing to do is to restart the authorization code flow. Keycloak will re-check if the user is still logged in (as set by the login expiry time settings), and if so, give a new token immediately. A more efficient method is to use the refresh token to get a new access token, as it requires only one server-to-server call.

You should verify the access token every time you load a webpage in your Web app. This ensures your Web app will become inaccessible whenever the user logs out in any browser tab or window, and gets a new token when the token expires. Additionally, the token should be used to call APIs in another software component. Within Pharaon, the standard we use for passing a token to an API on behalf of an end user is by using the same header field used for userinfo authorization, that is, "*Authorization: Bearer [AccessToken]"*. The other component can then check your token for validity by calling userinfo.

Finally, if you want Keycloak to log the user out, load the following URL in the browser:

https://[KeycloakServer]/auth/realms/[YourRealmName]/protocol/openid-connect/logout

It has no parameters and can easily be put on a logout button in your Web app.

2.2.2 Client credentials flow

The client credentials flow can be used for server-to-server communication that is not on behalf of a particular end user. This flow is much simpler, as it involves only a single call to the token endpoint. First, you have to make sure you configured a client in Keycloak that corresponds to your software component. Ensure that the client's Service Accounts Enabled option is on. Now, you can do the following call to get the token:

curl --location --request POST 'http://[KeycloakServer]/auth/realms/[YourRealmName]
/protocol/openid-connect/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=client_credentials' \
--data-urlencode 'client_id=[ClientId]' \
--data-urlencode 'client_secret=[ClientSecret]'

If your credentials are correct, you will receive a JSON structure which contains a field access_token containing a JWT:

{
access_token: '[JWT_Token]',
expires_in: 300,
refresh_expires_in: 0,
token_type: 'Bearer',
'not-before-policy': 0,
scope: 'email profile'
}

A server that receives this JWT can decode and verify it without any call to Keycloak. You can use this online tool to do both:

- https://dinochiesa.github.io/jwt/

The decoded header and payload should look something like this:

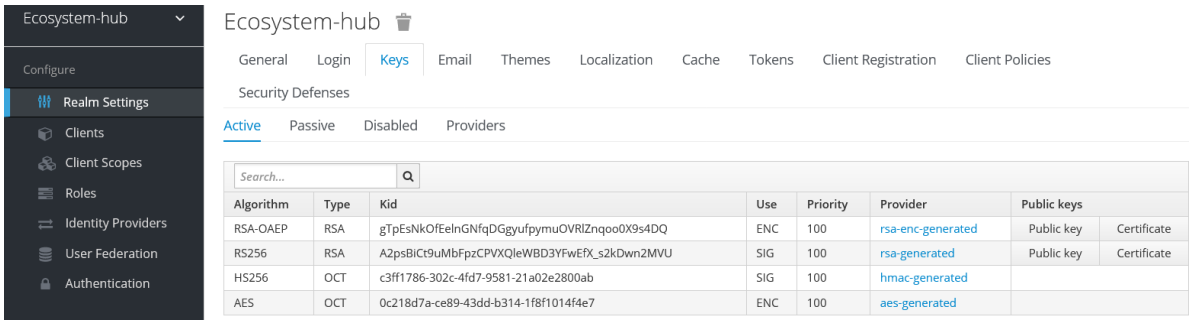
Header:

{
"alg": "RS256",
"typ": "JWT",
"kid": "A2psBiCt9uMbFpzCPVXQleWBD3YFwEfX_s2kDwn2MVU"
}

Payload:

{
"exp": 1671784742,
"iat": 1671784442,
"jti": "35431617-662f-4469-a3d1-66ae074e1678",
"iss": "https://pharaon-dev-idp.eng.it/auth/realms/ecosystem-hub",
"aud": "account",
"sub": "0edb13ab-9df0-4e52-a44c-7acc661b25da",
"typ": "Bearer",
"azp": "rrd-resource-broker",
[…]
}

You can ignore most fields. The exp field indicates the Unix time stamp that the token expires, which you should check. With help of the Keycloak public RS256 key, you can verify if the token was generated by Keycloak. In the online JWT tool, you can fill in the public key at the bottom right. Click the checkmark in the middle to verify the JWT against the key. You will find the public key under Realm settings -> Keys. Click on the RS256 column's Public key button to get the key.



2.3 API gateways

2.3.1 ICE API Gateway

ICE API Gateway is a lightweight API that helps connect multiple services together, without having to worry about authentication.

For example, let's assume that service A wants to connect to service B, and the latter requires a Bearer token. You can use ICE API Gateway to store authentication details (Auth Url, clientId, secret, etc) and then point service A to call ICE API Gateway. When called from service A, it will query the Auth server to grab a token, will append it as a header to the call, and will query service B.

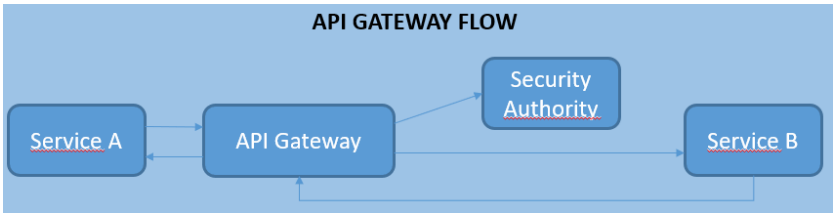


Figure 4. ICE API gateway use case

2.3.2 Kong

A use case of API Gateway solution to allow authentication and authorization can be implemented using Kong API Gateway as shown in the figure.

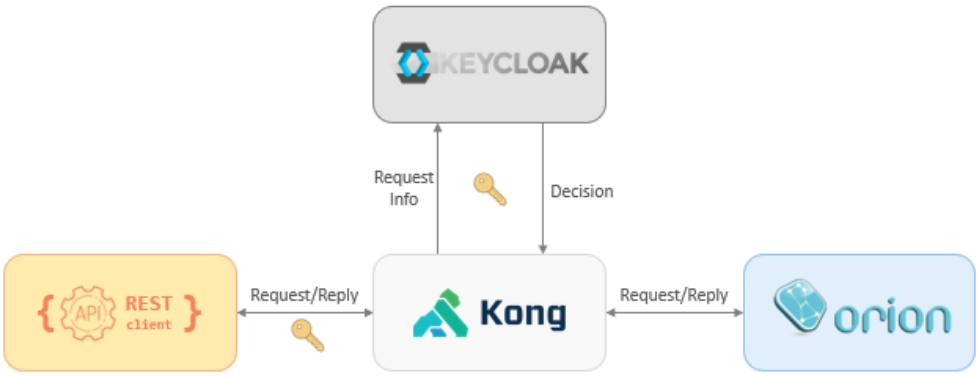


Figure 5. Kong serving as gateway for Fiware Orion

This is a typical architecture to protect a backend service (in this case FIWARE Orion, but you can use any REST APIs server) through Identity Management (i.e. Keycloak). The workflow is simple; a client REST makes a query by using a token (in the header), and the Kong API Gateway evaluates the token

(asking in the Keycloak server) and makes a decision - according to this decision, Kong allows or denies the request.

Also, the configuration in Keycloak is easy. First, it's necessary to create some roles to assign different users. Second, create a resource (identifying a real path) and assign a policy, permission, and role. This represents a resource-based configuration.

For example, let's suppose to create a userA user and assign him a roleA role; and let's suppose to create permission for the resource on /pathA path. After login and retrieving the token for userA, he can access on /pathA path according to the resource-based configuration but he cannot access on /pathB path.

2.4 OpenAPI Semantic Metadata reference

OpenAPI already supports human-readable semantic annotations, like the summary, description and example fields. Our approach adds similar, but computer-readable, annotations next to these, using the keyword **x-def**. The value of an x-def field should be a "canonicalized" URL, referring to a particular concept in an ontology or wiki, or code in a particular coding system. This approach enables annotating APIs, API functions, webapps, and webpages, as well as annotate JSON schemas. For example, we can specify that a returned numeric value is a Temperature in Celsius, or a particular key-value data structure is a Pharaon user account.

Besides x-def, we identify several other annotations for data fields and structures:

- x-unit: specifies a Unified Code for Units of measure/UCUM string for a numerical value.
- x-currency: specifies a currency code for a numerical value
- x-format: specifies the data format of a String value through a canonicalized URL that describes the format

Resource and service semantics

The search facility enables Pharaon technologies to search within the existing library of services via the API repository. It enables search by content, enabling any Pharaon software to find particular services by searching for a particular API format along with particular semantic annotations. For example, we can look for an API with a specific time range query function that outputs Temperature values in Celsius.

In addition to APIs, the broker can also handle Web application entry points. A Web entry point is syntactically identical to an API function: a GET or POST request to a URL with particular parameters, usually returning a text/html formatted response body. Encoding Web entry points in this way enables semantic interoperability at the application layer, that is, facilitating user interface integration via a "plugin"scheme. An example of a Web entry point would be a dashboard page, showing a data overview of a specific technology, or a configuration page, enabling a user to configure the technology. A central platform UI can then use the resource broker to look up all dashboard or configuration pages in the system, and provide links to them on its Home or Settings page, or even embed them in a single page using iframes or similar scheme. For each Web entry point, we can also define metadata, like title and icon, that can be used to display the link.

At the resource level, we provide the following additional semantic / metadata annotations:

- x-conformsto: specifies a URL, where a partial or complete OpenAPI definition, can be found. This specifies that a particular specification syntactically conforms to this definition.
- x-display: specifies metadata for displaying a link to a webpage, in particular title, description, and icon. This should be specified at the function level (e.g. under “get”).

At the service level, we also provide a standardized way to specify context, namely the following context fields, that can be specified inside the *info* field:

- x-servicetype: identifier indicating the type of service. Can be: api or website.
- x-componentid: a reverse domain name notation denoting a specific component from a specific organisation (e.g. nl.rrd.paco). This enables getting resouces from a specific component.
- x-instanceid: a free form string, indicating a particular instance of a component
- x-country: two letter country code, useful for local services or country-specific legal restrictions
- x-location: free-form string, indicating location for local services
- x-languages: array of language codes for any resources that involves natural language, especially Web applications, so we can match them to the user’s language.
- x-department: a department identifier. Specifies that the resources are for use within a particular (healthcare) organization or department. Each healthcare department involved in Pharaon is assigned a unique department code.

2.4.1 Examples

Below we specify more detailed examples of semantically annotated resources and queries to find them. A resource query specifies a partial OpenAPI structure. Basically, an OpenAPI definition matches the query when the definition contains at least the same elements as the query. There are some special cases that cater for OpenAPI semantics, for example, if a resource has a required parameter that is not specified in the query, it will not match.

The simplest use case for the resource broker is finding an API of a particular known software component by specifying its component ID. Here, the resource broker serves the same function as a typical microservice registry: to connect existing known services to each other without hard coding URLs. This can be done with the following query. The green parts are the semantic annotations:

```
{
        “info”: {
                “x-componentid”: “nl.rrd.paco”
        }
}
```

A similar use case is finding components that provide a particular API standard. For example, consider we are looking for a component that supports NGSIv2 (the Fiware context broker API). We can specify the entire NGSIv2 OpenAPI specification in our query, or even just the parts we need. This would however specify an API that is *syntactically* identical to NGSIv2. Instead we can also simply specify a x-def link to NGSIv2 at the service level. The query looks something like this.

```
{
        “swagger”: “2.0”,
```

```
    “info”: {
        “x-def”: [“https://fiware.github.io/specifications/ngsiv2/stable/”],
        “x-conformsto”: “https://raw.githubusercontent.com/Fiware/specifications/master/OpenAPI/ngsiv2/ngsiv2-openapi.json”
    }
}
```

Note that we do not specify any part of the API itself, which means we assume that returned APIs will conform to the format specified in NGSIv2. To make machine verification possible, we add x-conformsto, which points to the NGSIv2 syntactic OpenAPI specification. Note that the URLs in this example are just URLs from the current Fiware docs. Though they are not meant to be used as concept ontology URLs, it is fine to use them as long as the URLs are stable, and using them is agreed on within the Pharaon community. Because the URLs could change over time, we could alternatively add NGSIv2 to the Pharaon concept wiki, and refer to the right docs from there.

Another simple use case is to search for particular Web entry points, such as the aforementioned dashboard pages. The following query searches for a GET resource that shows a dashboard page:

```
{
    "openapi": "3.0.0",
    "servers": [{
        "url": "$url"
    }],
    "paths": {
        "$path": {
            "get": {
                "x-def": ["https://gitlab.com/pharaongroup/concepts/-/wikis/resource/webpage/dashboard"]
            }
        }
    }
}
```

Here, we refer to a definition in the Pharaon concept wiki. In this example, we also introduce a new construct here, called variable capture. The $url and $path directives will capture parts of each returned result into the variables *url* and *path*, which are returned along with it. Now, we can construct the resource URL by simply concatenating the captured url and path.

In the example below, we specify a webpage that enables the user to enter their weight. Note that the human-readable semantic annotations that are already part of the OpenAPI standard are marked blue.

```
{
    "openapi": "3.0.0",
    "info": {
        "description": "Weight component webpages",
        "version": "1.0",
        "title": "Weight component webpages",
        "x-languages": [“nl_nl”, "en_gb"]
    },
    […]
```

```
        "paths": {
                "/enterweight.html": {
                        "get": {
                                "x-def": [ "https://gitlab.com/pharaongroup/concepts/-/wikis/resource/webpage/data-entry" ],
                                "x-display": {
                                        "icon": "http://sr.photos1.fotosearch.com/bthumb/CSP/CSP991/k12478283.jpg",
                                        "title": "Enter weight",
                                        "description": "Enter your weight manually",
                                },
                                "parameters": [{
                                        "x-def": ["https://gitlab.com/pharaongroup/concepts/-/wikis/data/redirect-URL"],
                                        "name": "redirect_url",
                                        "in": "query",
                                        "required": true,
                                        "schema": {
                                                "type": "string"
                                        }
                                }],
                                "responses": {
                                        "200": { "description": "ok" }
                                }
                        }
                }
        }
}
```

The *x-languages* annotation enables queries to specify the required languages. The *x-display* annotation provides some metadata that enables an application component to display the link. Additionally, we specify that the webpage has a parameter that represents a redirect URL, that can be used when the data entry is finished.

The entered weight data is handled by the component that provides this webpage, and is typically stored in the component's local database. Retrieving the value requires a separate companion data retrieval function. The semantic definition does not specify the type of data that is entered. This could be achieved by defining a subclass weight (resource/webpage/data-entry/weight). But as it is, it can for example be used by a central portal to provide an overview of data entry pages in the whole Pharaon system.

Below is a more complex example, showing a fragment of the RRD eHealth platform OpenAPI specification with semantic annotations.

```
{
        "swagger": "2.0",
        "info": {
                "description": "R2D2 eHealth platform API",
                "version": "1.0",
                // Stable URL where you can find the docs, which also serves as semantic identifier in queries.
```

```
            // So a simple broker query { “info”: { “x-kindof”: "https://www.rrdhost.nl/r2d2/docs/" }
            // will return this API.
            "x-def": ["https://www.rrdhost.nl/r2d2/docs/"],
            [ … ]
    },
    [ … ]
    "paths": {
            "/project/{project}/table/{table}": {
                    "get": {
                            "summary": "getRecords",
                            "x-def": "https://gitlab.com/pharaongroup/concepts/-/wikis/resource/API-function/standard-time-range-query",
                            "operationId": "getRecordsUsingGET",
                            "produces": [“application/json"],
                            "parameters": [{
                                    "name": "end",
                                    "in": "query",
                                    "description": "end time exclusive",
                                    "x-def": ["https://en.wikipedia.org/wiki/ISO_8601"],
                                    "required": false,
                                    "type": "string"
                            }, {
                            [ … ]
                            }, {
                                    "name": "table",
                                    "in": "path",
                                    "description": "table",
                                    "x-def": ["https://gitlab.com/pharaongroup/concepts/-/wikis/data/database-table-identifier "],
                                    "x-values": {
                                            "fitbitsteps": {
                                                    "x-def": "https://loinc.org/41950-7/",
                                                    "description": "Fitbit step data"
                                            }
                                    },
                                    "required": true,
                                    "type": "string"
                            }, {
                                    "name": "user",
                                    "in": "query",
                                    "description": "user ID",
                                    "x-def": ["https://gitlab.com/pharaongroup/concepts/-/wikis/data/user-ID"],
                                    "required": false,
                                    "type": "string"
                            }],
```

[…]

The fragment specifies a query on a database table. User, start (not shown), and end times are specified as query parameters. We specify the semantics of the parameters via Pharaon concept references. Using the x-values annotation, we can specify the meaning of particular values of a “enum” type value. Here, we specify that a particular table, called fitbitsteps, returns step data, using the appropriate Loinc code.

3 Pharaon technologies

This section describes the Pharaon pilot technologies that are to be integrated with the platform built by the applicants. Some are tentative, which means it is not certain if they will be included.

3.1 AmiCare (CETEM)

AmICare is a non-invasive system that provides peace of mind to relatives and carers. It is made of out-of-sight textile sensor with could support and smartphone app. Its smartphone app can be configured and personalized by each carer or app user to trigger alarms based on certain time and action rules, such as: "if not in bed by midnight", "if in the coach for more than 3 hours", "if away from bed for more than 1 hours between 0:00am and 6am".

3.1.1 Hardware

Amicare hardware consists of the following parts and components:

- External pressure sensors to detect occupancy in the bed or armchair.
- Environment and movement sensors installed in Amicare.
- External sensor to detect when the fridge is opened and closed.

Figure 6. Amicare, pressure sensor and aperture sensor.

3.1.2 User interface

Amicare Web App.

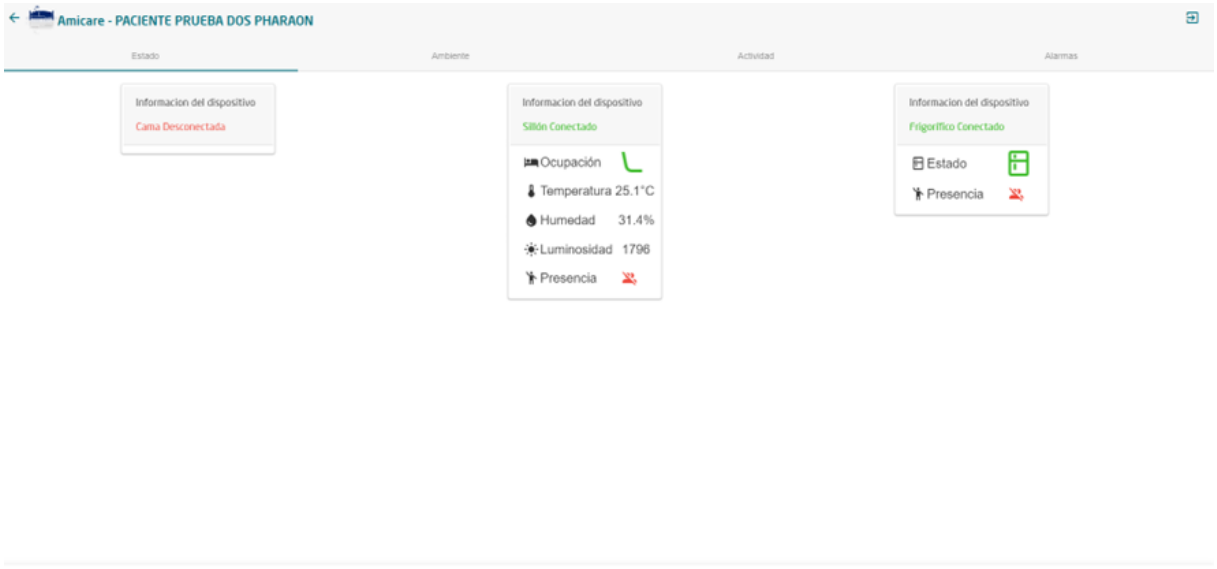


Figure 7. Actuated status of installed devices (main screen).

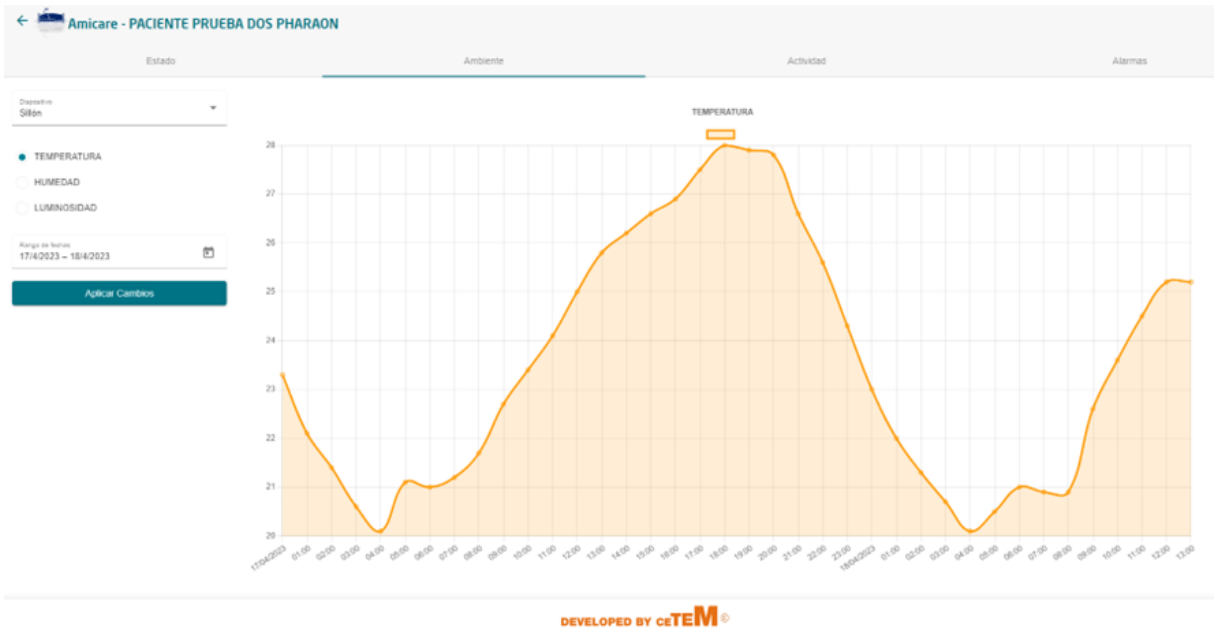


Figure 8. Amicare Web - Temperature data.

3.1.3 APIs

A public SwaggerUI is available at this URL:

- http://amicare-swagger-bucket2.s3-website.eu-central-1.amazonaws.com/

Explanation of the functions:

- Amicare_token: Authentication to obtain data from APIs
- Amicare Enviroments: By selecting the user and the date range (Timestamp ms), we obtain the temperature, humidity and luminosity data.

- Amicare Events: By selecting the user and the date range (Timestamp ms), we obtain the user's occupancy and movement data.
- Amicare Fridge: by selecting the user and the date range (timestamp ms), we obtain the opening and closing data of the fridge and the user's movement in the kitchen.

3.2 uGrid (MIWenergia)

The uGRID software aims to digitize energy consumption, providing to the final consumer more information about their demand of electrical energy. The purpose is to achieve the maximum possible energy efficiency and to have under control the electricity consumption by setting alerts and generating reports.

3.2.1 Hardware

uGrid requires installing smart power devices in the home. This includes energy consumption measurement devices and other smart home devices like smart lamps and switches.

3.2.2 User interface

uGrid has a Web app that works on small and large screens.

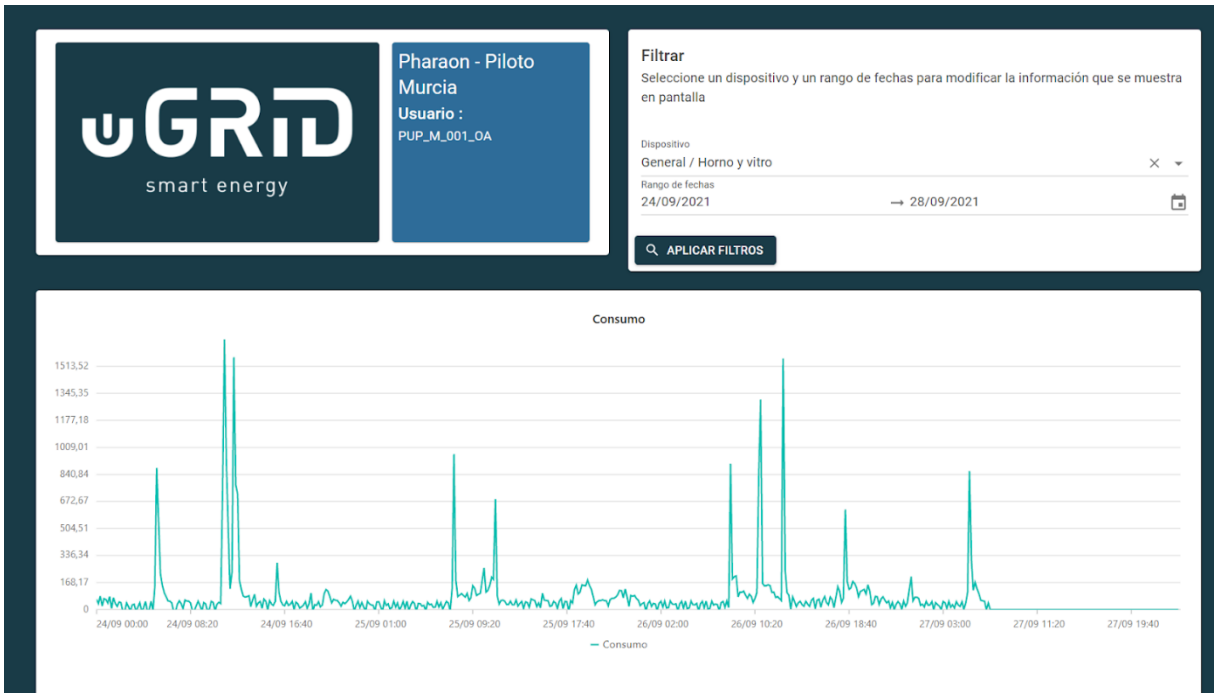


Figure 7. Web app showing energy consumption.

3.2.3 APIs

uGrid provides an API for accessing the collected energy data. A public SwaggerUI is available at this URL:

- https://api-pharaon.miwenergia.com/swagger/index.html

3.3 Discovery (Ascora)

Discovery is a web-based visualization component for data-driven information. Data from other technologies is aggregated, processed and then presented in the form of individual diagrams. Discovery was originally developed in the EU H2020 project MMAA and was further developed in the BMBF projects AwareMe and PDExergames with regard to the special requirements in the eHealth area (role models, security aspects, data types, etc).

3.3.1 Hardware

Discovery is Web-based, and will run on phones, tablets, and PCs. No special hardware is required.

3.3.2 User interface

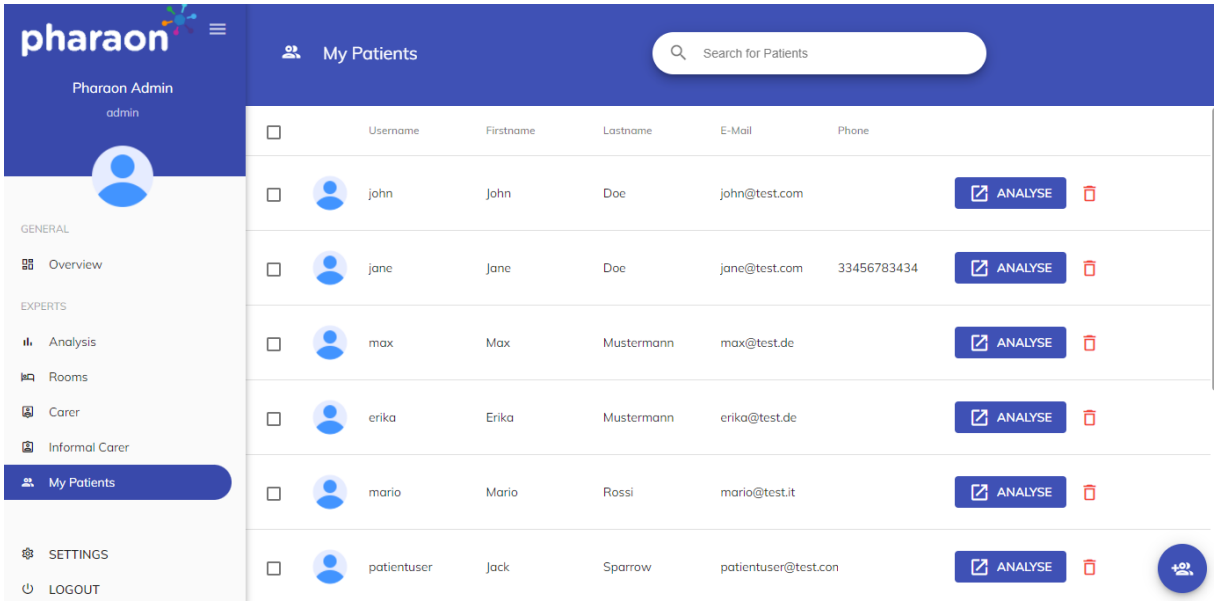


Figure 7. Overview screen, showing the main menu and users.

Figure 8. Example of a graph screen.

3.3.3 APIs

An API is available for entering data to be shown in the Discovery dashboards. A public SwaggerUI instance is available here:

- https://backend-pharaon.ascora.eu/swagger/index.html

3.4 PACO (RRD)

PACO is a program that aims at the promotion of healthy eating behaviour in elderly people, while simultaneously reducing loneliness. Two virtual coaches guide the participants through the different modules of PACO. One of the coaches is Herman, a chef, who guides users in the cooking related modules, namely the 'eating book' and the 'recipe book'. The other coach is Ellen, an elderly woman, who is the guide for the 'goal book' and the 'stories' module. PACO also enables the user to formulate health goals.

3.4.1 Hardware

PACO is Web-based, and is designed for tablets and PCs. No special hardware is required.

3.4.2 User interface

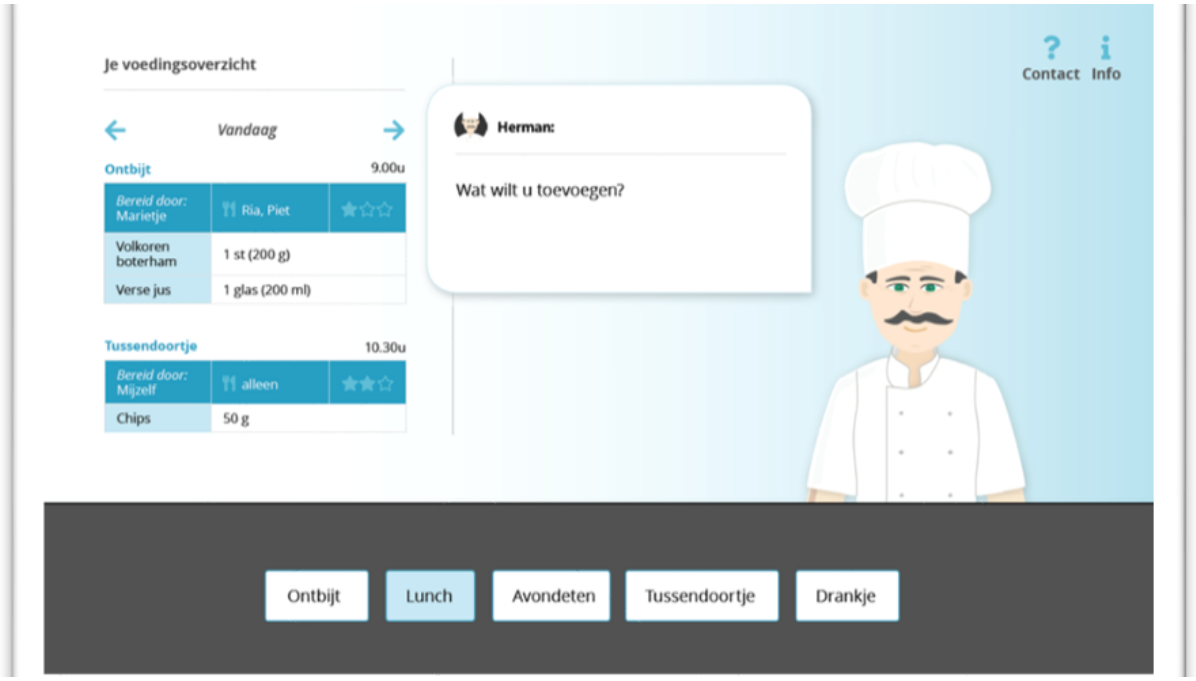


Figure 9. Food diary entry screen. User can enter ingredients, amount, and with whom the meal was eaten.

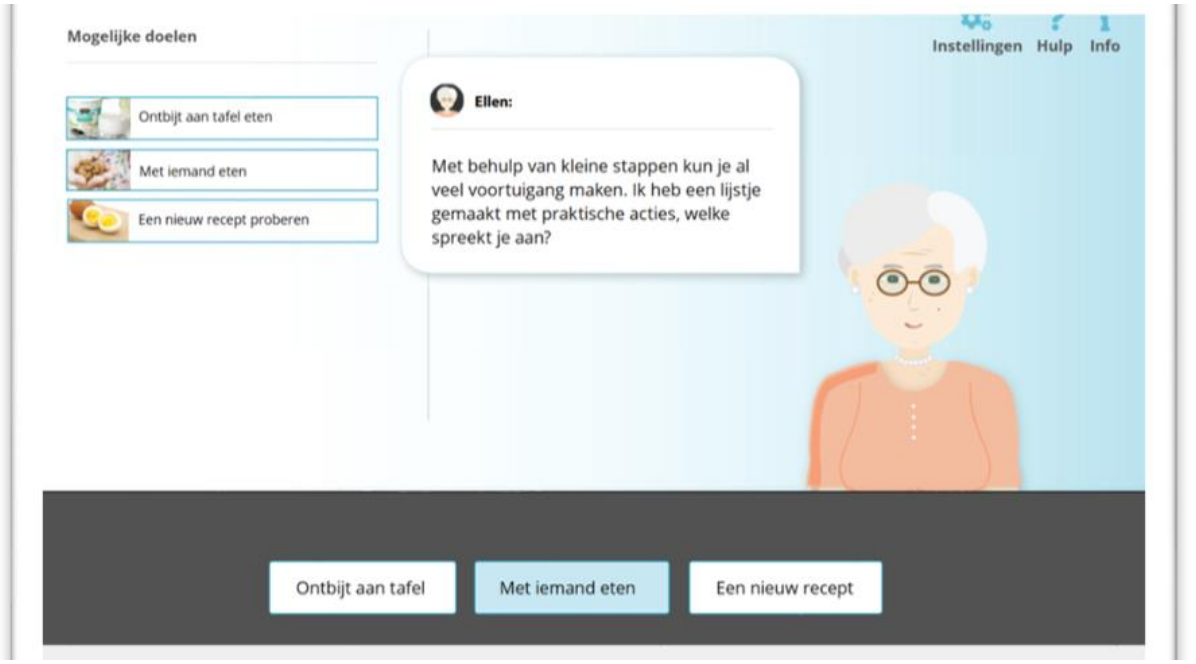


Figure 10. Health goals dialogue. The user is coached into specifying personal health-related goals.

3.4.3 APIs

PACO provides an API for reading the configuration, current program progress, meals entered by the user, and the user's health goals.

A public SwaggerUI instance is available here:

- https://www.rrdhost.nl/servlets/paco/r2d2project/swagger-ui/

3.5 RRD eHealth platform app (RRD)

The RRD eHealth platform includes an app, that enables monitoring steps, heart rate, sleep via a connected sensor. It also enables periodic questionnaires to be administered. A step goal can also be set, which can be adapted automatically to the user's actual step count. It also has a coaching feature, where user can be led through coaching dialogues, that can be developed using a low-code visual programming tool. The collected data can be accessed via the platform API.

3.5.1 Hardware

The app runs on Android phones and tablets, with a beta IOS version also available. Measuring steps, heartrate, and sleep, is done via an external device. Several device options are available: Fitbit, Mox, and Garmin.

3.5.2 User interface

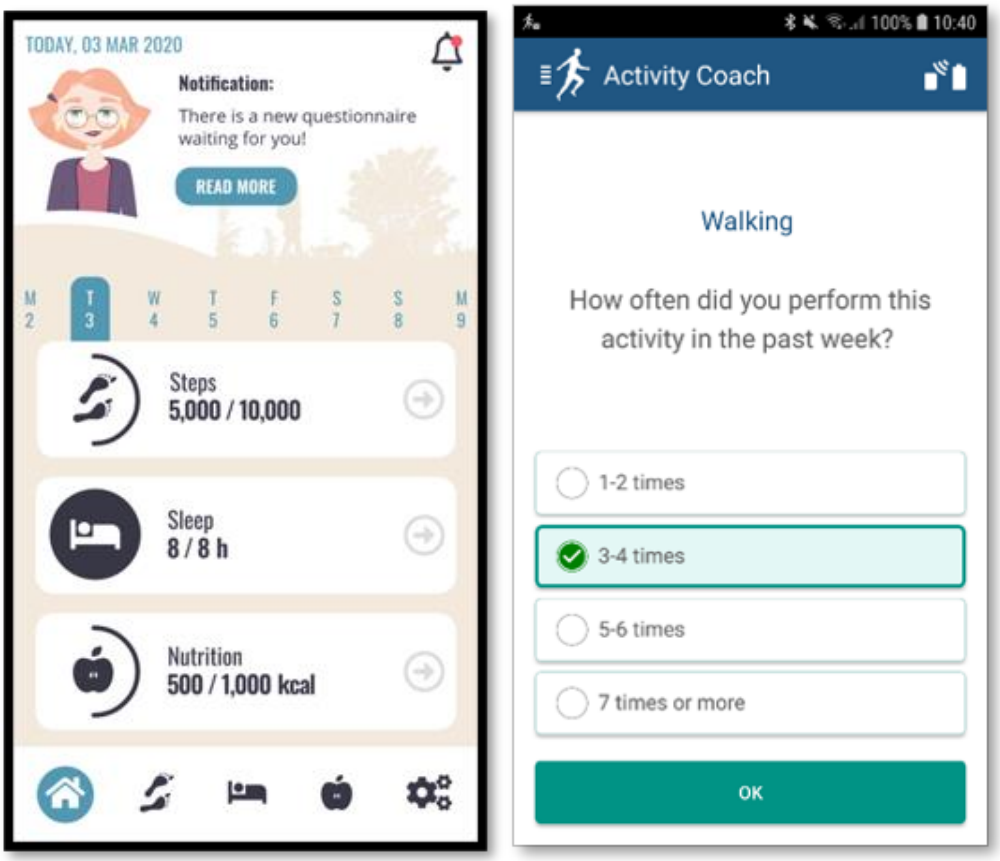


Figure 11. Left: home screen, showing a dashboard with links to the different screens. Right: example questionnaire.

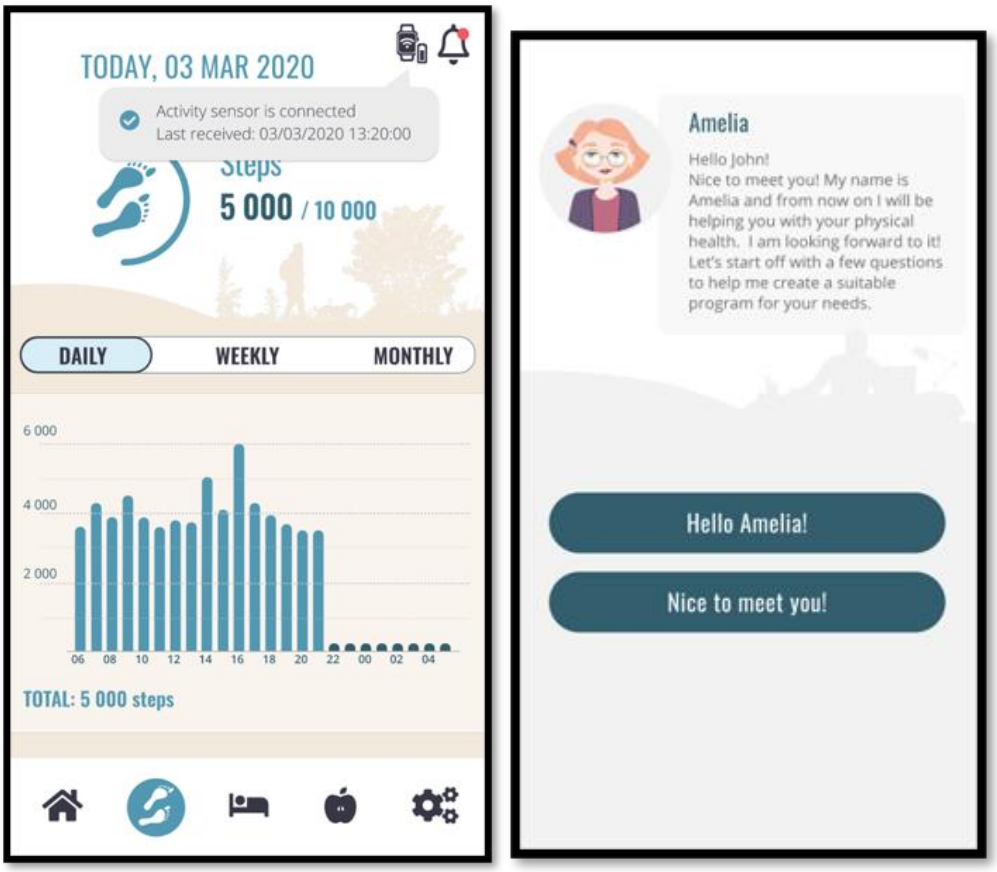


Figure 12. Left: step count graph. Right: coaching dialogue with multiple choice question.

3.5.3 APIs

The eHealth platform has an extensive API, which includes reading the collected data. A public SwaggerUI is available here:

- https://www.rrdhost.nl/servlets/paco/r2d2/swagger-ui/

3.6 Sentab TV and App (Sentab, tentative)

Sentab is an interactive environment linking older adults to their families and peers. The system aims to address loneliness aspects, provide health monitoring capabilities, and link socially. The suite of applications is designed to work on TV (Sentab TV box), web and apps. The functionality includes video calling, media sharing, newsfeeds, integrated health devices (blood pressure, activity monitor), Sentab wellbeing Index etc.

3.6.1 Hardware

Sentab TV works on a smart TV device (Sentab TV box). Sentab App works on a smartphone.

3.6.2 User interface

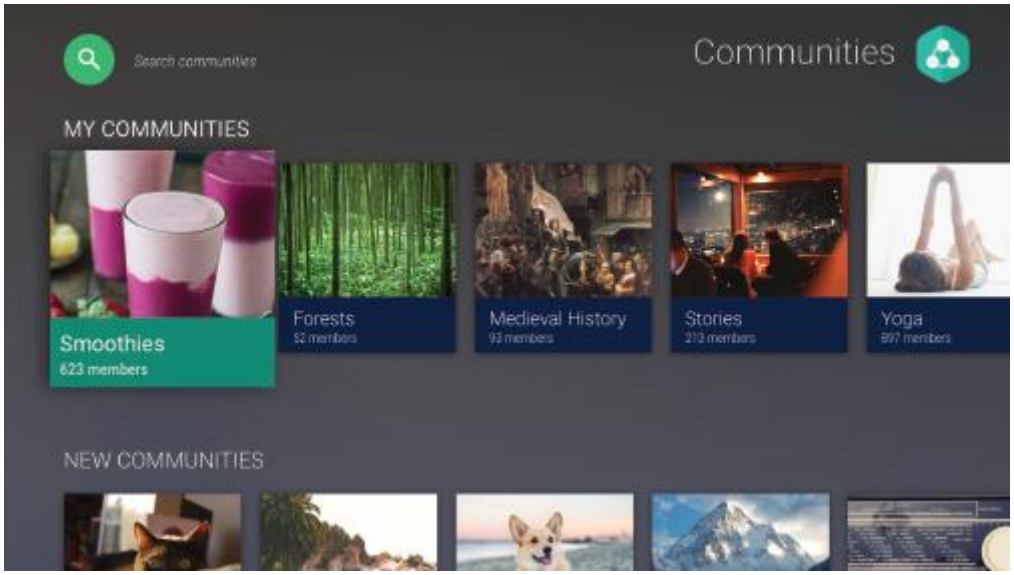


Figure 13. Communities screen

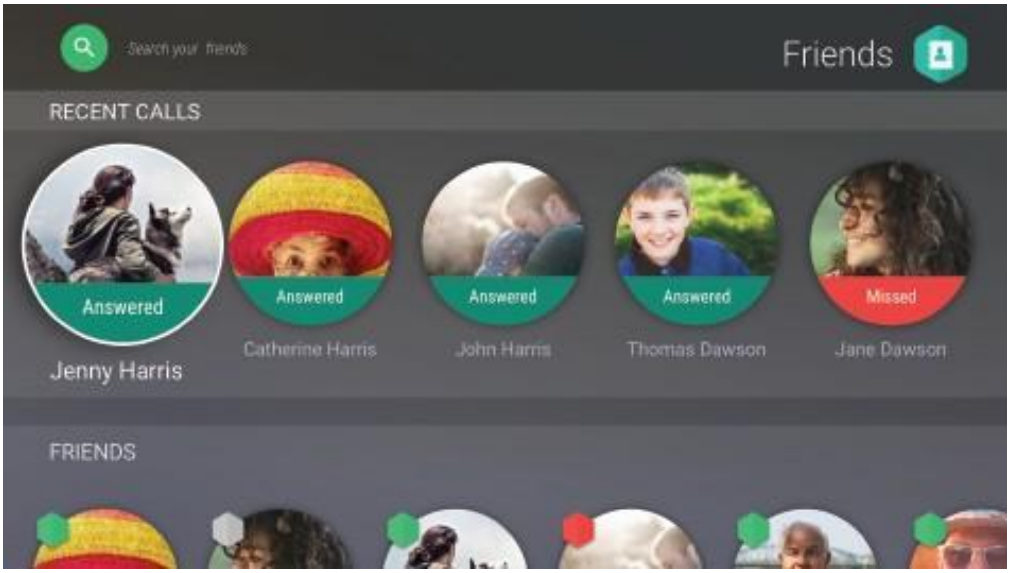


Figure 14. Friends and calls screen

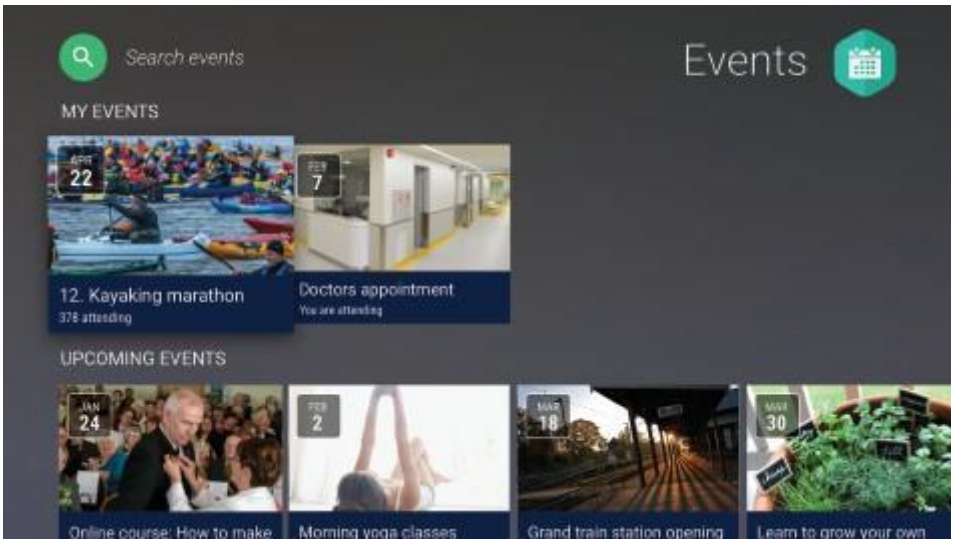


Figure 15. Social events screen

3.6.3 APIs

Details are not available at this moment.

3.7 A.I Improved Social robotics (Co-Robotics)

The MoveR-L robot is a general-purpose robot that can navigate autonomously indoors. It can carry out a telepresence service between patient and caregiver or nurse, that is, by carrying out an indoor video call service for example. Through an AI software, the robot itself can navigate autonomously coming up to the patient and start the service via a touch screen, or autonomously if required.

The robot is provided with a shared control system that allows the user to directly control the robot movements and function at any time, but also to let the robot autonomously perform all the activity. This function improves the usability of the machine, keeping it flexible and reliable.

Furthermore, the provided MoveR-L robot can safely approach the end-users leveraging safety laser scanners to avoid obstacles and hurting people. The combination of this safety function with the artificial vision services allows this robot to closely approach the user in a proper effective and comfortable manner to improve acceptability and usability of the Pharaon services. Indeed, when the robot is asked to deliver a service, it uses the embedded cameras to detect humans and when a human shape is detected, the robot approach procedure starts. In particular the robot slowly moves frontally to the user, to provide a facilitated access to the embedded touch screen. Once a safe but comfortable distance is reached, the touch screen shows the pharaon selected services. The images processed by the robot cameras are privacy compliant since only a low-definition depth data stream is processed and no image is stored.

The robotic platform is a lightweight ROS based AGV for advanced social interaction. The robot shape fits close indoor environments thanks to a small structure.

The robot is integrated with the Fiware context broker. Robot data, such as position, speed, battery status, usage, safety blocks and any alerts to be sent to a web interface, are available through Fiware. Via a Web-based user interface, it is possible to have real-time visual feedback on the status and sensors of the robot.

We will integrate the robot with the FIWARE user interface designed by ENG. The robot will connect autonomously to the security layer that allows us to interface with Orion, a C ++ implementation of the NGSIv2 REST API binding developed as a part of the FIWARE platform and storing sensor information on the MongoDB database. The security layer will allow the user to register as a caregiver or as a patient in order to display all the necessary information directly on the interface in Monitoring-IT.

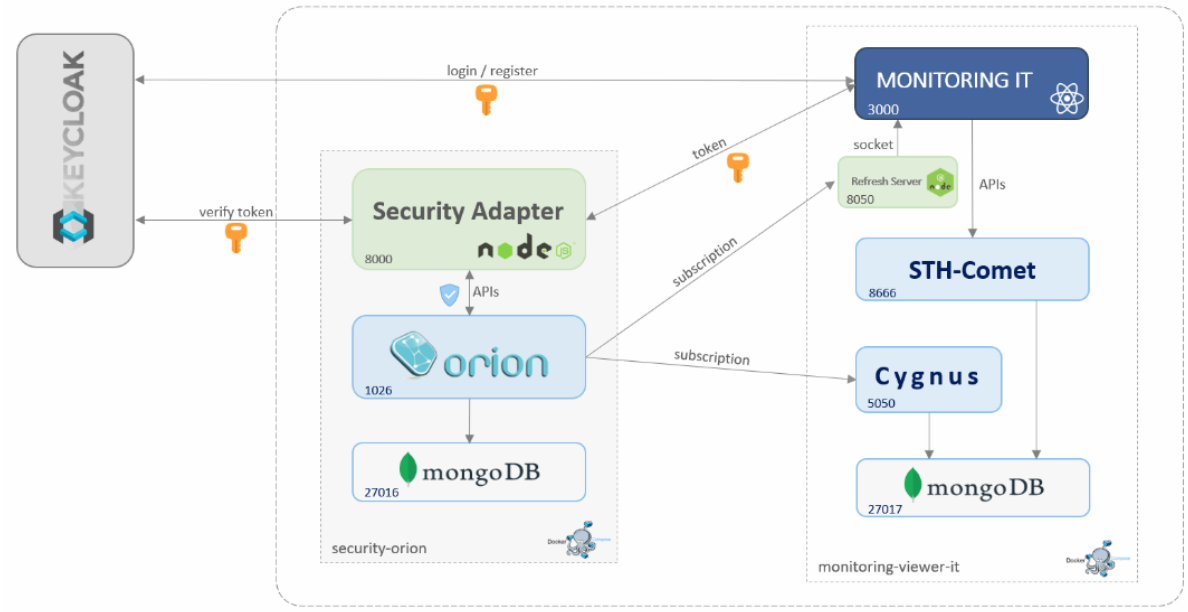


Figure 16. Pharaon Monitoring-IT

Note that at this time, limited copies of the robot hardware are available. One robot is available and the providing of further copies requires 8 weeks.

Figure 17. Robot prototype

3.7.1 Hardware

The robot leverages the two (rear plus frontal) laser scanners to map and locate itself in the environment, in a privacy compliant path planning and obstacle avoidance function. These sensors scan 360° all the relative distances between the robot and the environment. No user/face image or audio is taken while the robot autonomously moves. The user's safety is guaranteed by the laser scanner that triggers a software and hardware motor stop if an obstacle enters close in the safety distance of the robot. All the laser data is not stored, but only temporally used by the Bayesian based navigation algorithm for the autonomous navigation (Monte Carlo based probabilistic filter).

The use of infrared and depth camera images to detect users, allow the robot to autonomously look around for the user. The low-resolution and blurred images in the infrared are only processed and not stored.

An embedded PC is used for running algorithms and it can be used in indoor rooms only, with flat floors and Wi-Fi connection coverage over the entire operational area. Moreover the device recharges by means of a wall plug charging system.

The robot is provided with a touch screen and can be controlled by touch screen or smartphone using the Co-Robotics app as described below. The integration with Pharaon by using FIWARE interface is under development, this integrated interface will provide a single sign-on function and the data useful technology implementation including the robot position, alerts, service timestamps and battery level.

3.7.2 User interface

To control the robot first of all the user (formal caregiver) needs to register via username and password in the app link https://corobotics.igloo.ooo/signup (Figure 18).

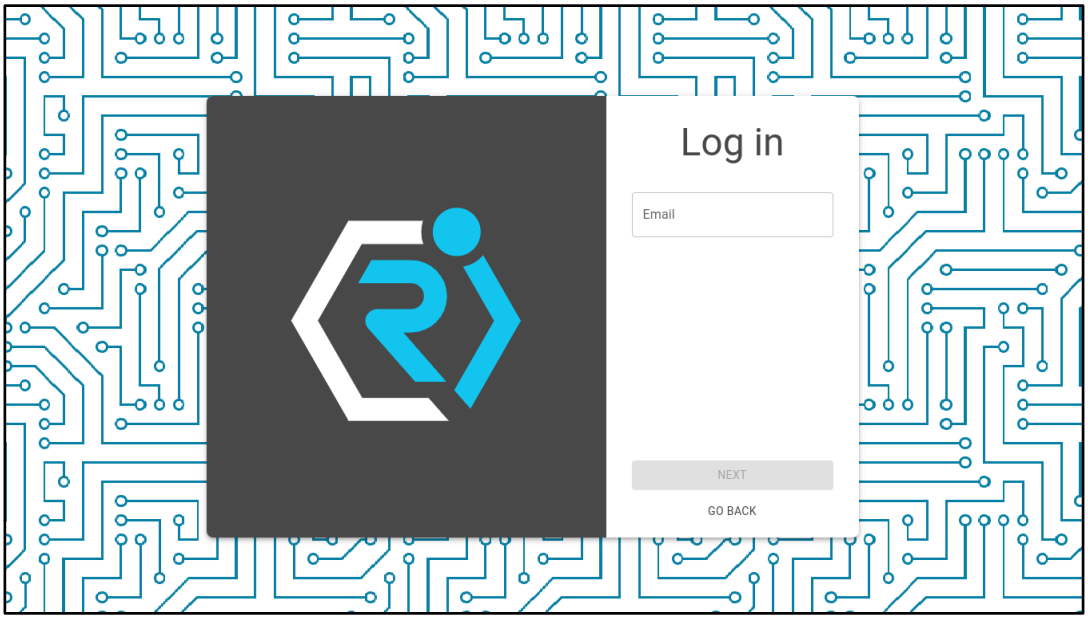


Figure 18: AI improved telepresence robot mover - Log in in the robot app.

Once logged in, one or more robot collections can be created. Within each collection it is possible to load manually or through QR-code all robots we want (Figure 19). Indeed, each robot comes with an associated unique identification number visualised as a QR code.

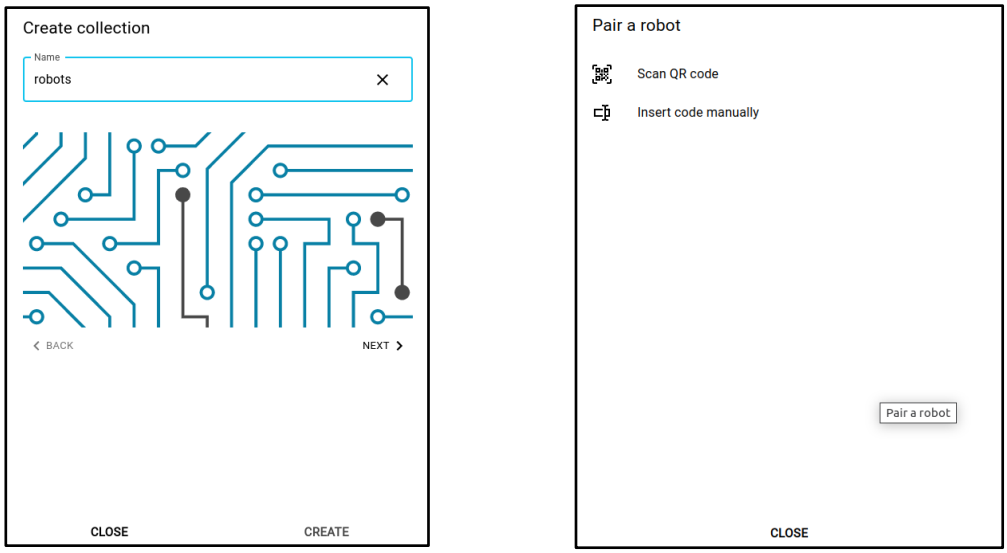


Figure 19: AI improved telepresence robot mover - Robots collection and pair new robot.

If the association has gone well, the robot control page will open instantly. Inside it is possible to teleoperate the robot through the Joystick variable, generate a new map, or commando it to reach the targets selected during the "New Program" phase, if a program has been selected (Figure 20).

To *Create a New Map* turn on the robot at charge point, i.e. where the robot charging station has been set up. To generate a new map the user clicks on the "Create new map" button and follows the instructions that the robot will list vocally.

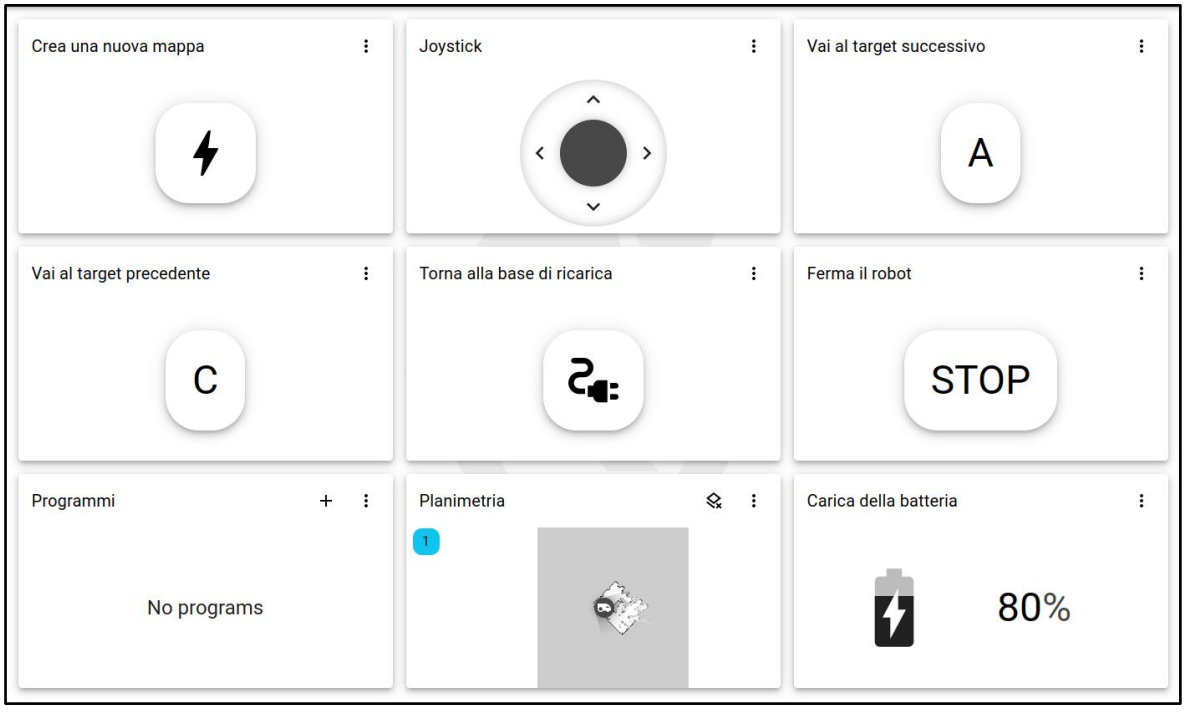


Figure 20: AI improved telepresence robot mover - App's home page.

Once the robot is ready, it will be possible to guide it in the environment to be mapped, following the procedure reported in the user manual and once the lap is finished bring the robot back to the starting point, then you can press on the button “Save Map”. The robot will save the map and only after the robot has said “Map saved correctly” can we be sure of the success of the procedure. Otherwise repeat the procedure.

Immediately afterwards, the map just made appear in the “Planimetry” field, so the user sees the success or otherwise of the operation.

To *Create a New Program* the user clicks on the “+” icon in the “Programs” card (Figure 21), only after creating the map. Now the user can insert the necessary data:

- Program name;
- Instruction name;
- Target name;
- Coordinates value and the angle in degree. You can enter them manually or graphically by clicking on the map at the desired point and drag the pointer in the desired direction (Figure 22).

In addition, it will be possible to manage any actions to be performed and the speed with which to reach that particular target.

Once the map has been generated and the program or programs associated with it have been created, it is possible to *control the robot,* just select the desired program (Note: in the selection of the program the robot will always have to start from the charging station, or from the zero point of the map).

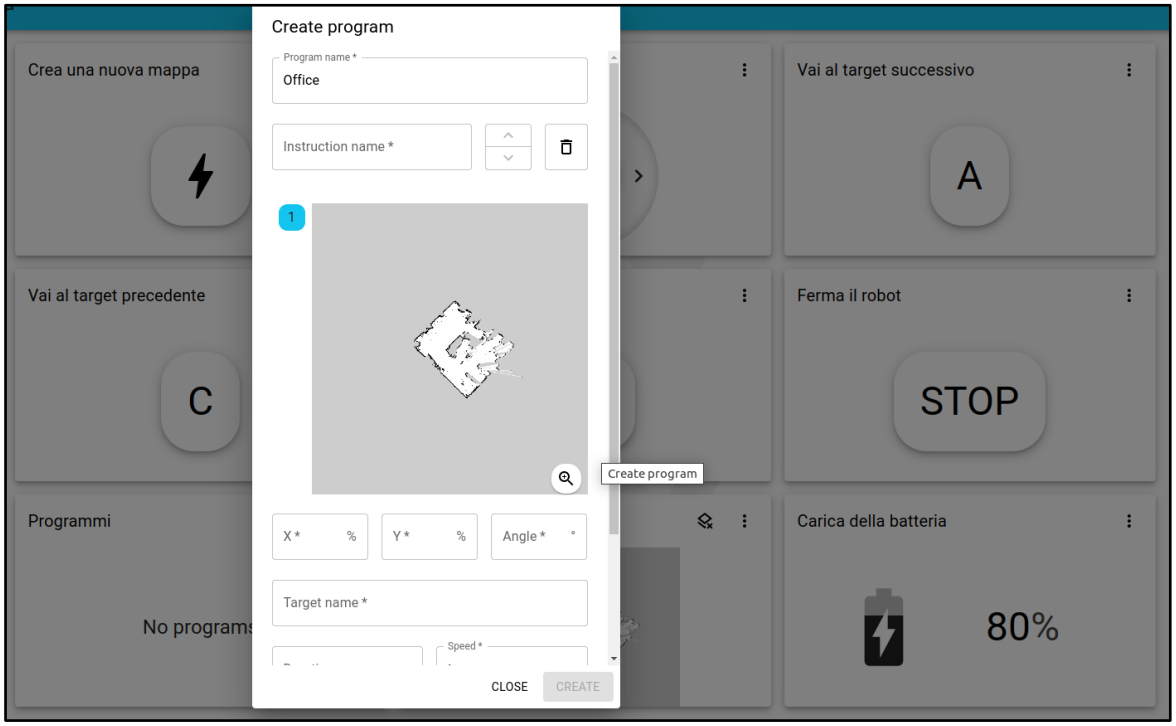


Figure 21: AI improved telepresence robot mover - How to create a new program.

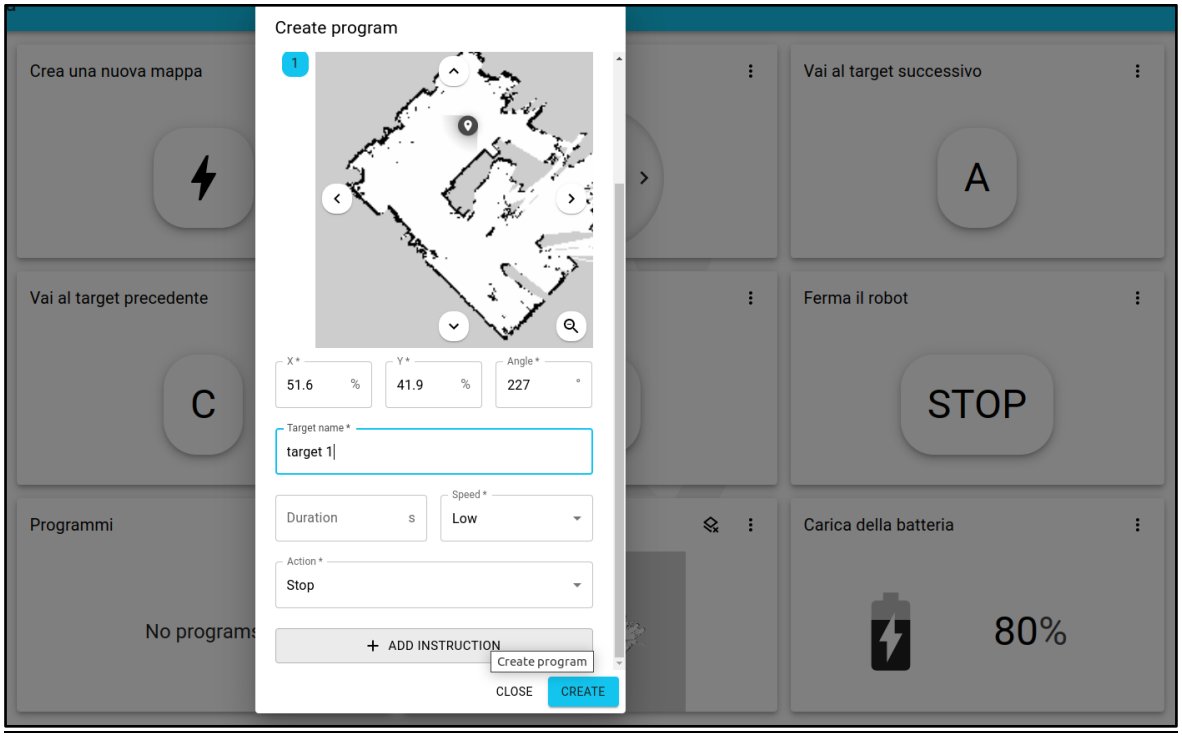


Figure 22: AI improved telepresence robot mover - Target selection.

Thus the robot receiving the command, will load the program to be executed and only after the robot has said “Ready to go”. It will be possible to give it the following commands:

- Prev/Next: reach the previous or the next target;
- Stop: stop the robot;
- Charge: send it back to the charging station.

3.7.3 APIs

Details are not available at this moment. It is under development.

3.8 IoTool (SenLab)

IoTool (https://iotool.io) is an IoT platform that can collect sensor data from different sources, like FitBit, environmental sensors, etc. and control actuators based on triggers and recipes. It also supports virtual sensors / algorithms processing data based on real sensors, like SUM, MAX, MIN etc. Currently we support approximately 150 sensors and other sensors can be added by SenLab or external developers in case that the specific sensor communication protocol description or API is available. Sensors can communicate with the IoTool platform using different protocols and interfaces, namely:

- An Android smartphone app (all internal Android sensors and actuators, BT, BLE, WiFi connected devices). The app is freely available on Google Play, same for the API.

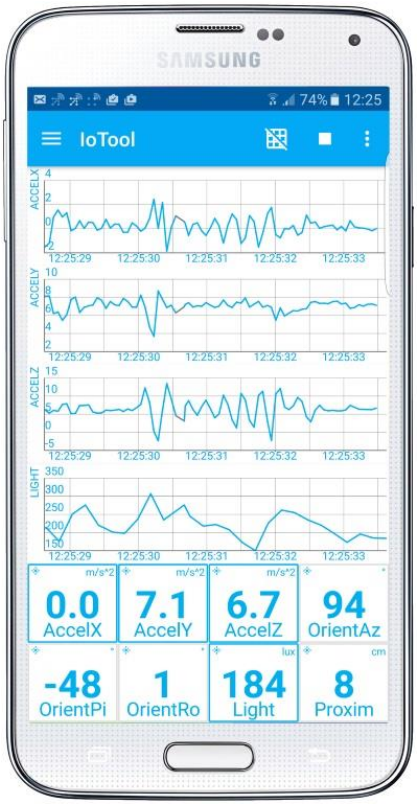


Figure 17. IoTool frontend / Android

- A direct connection (WiFi, ETH) to the IoTool cloud. For example, a lot of environmental sensors / actuators use it, like Shelly devices and some AirQ monitors.
- An indirect collection - in this case sensor sends data to its own cloud and IoTool cloud is collecting data using API from it.

Supported sensors / actuators:

- Medical/wellness, like FitBit, Pine64, Sensoria, Zephyr technologies with ECG, Core, various wristbands with HR, steps, activity recognition, fall detection, location and other sensors.
- Environmental, like Shelly (all devices supported like PIR, gas leak detection, temperature, humidity, switches, light bulbs), QingPing, AirQ, Awair, Ruu-vi.
- SmartSocks, like Sensoria.
- IMUs, like Sense, mBientLab.
- Other.

As mentioned before, other devices can be integrated, usually in a few hours, all depending on the device documentation quality. More about IoTool extensions can be found on https://iotool.io/extensions.

IoTool cloud also incorporates the admin part (users, devices description, connectors), visualisations (Grafana with Prometheus), monitoring and logs (Grafana with Prometheus and Loki), MQTT clients and broker, routing engine based on node.js and node-red, importers, exporters and connectors to other platforms (like ThingsBoard, IBM Watson, Microsoft Azure, Google).

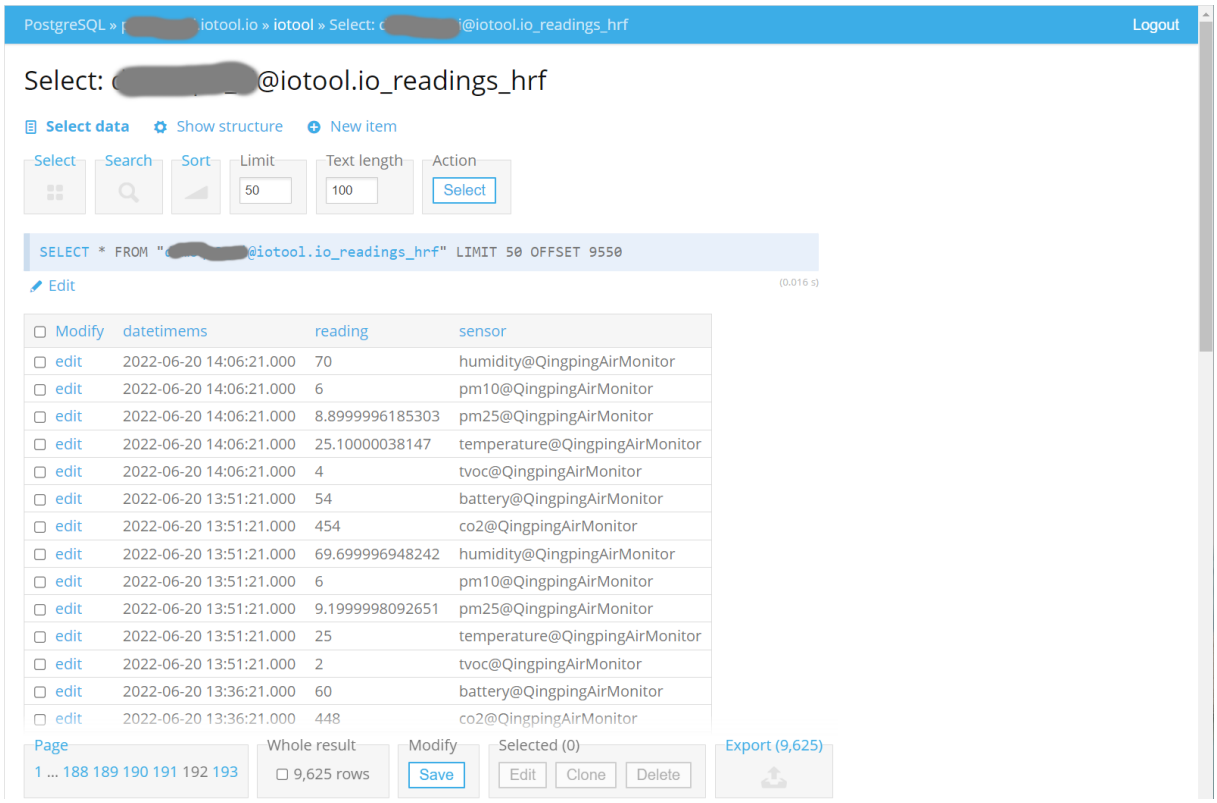


Figure 18. IoTool Cloud admin

IoTool will be integrated with the Fiware context broker, so all sensor data will be available via Fiware.

IoTool also includes a workflow system (on the client and cloud side) where rules, triggers and connectors can be defined to perform actions on certain sensor and actuator data, manage data flow and create specific rules. It is based on node-red and it can be easily adjusted to the user case needs.

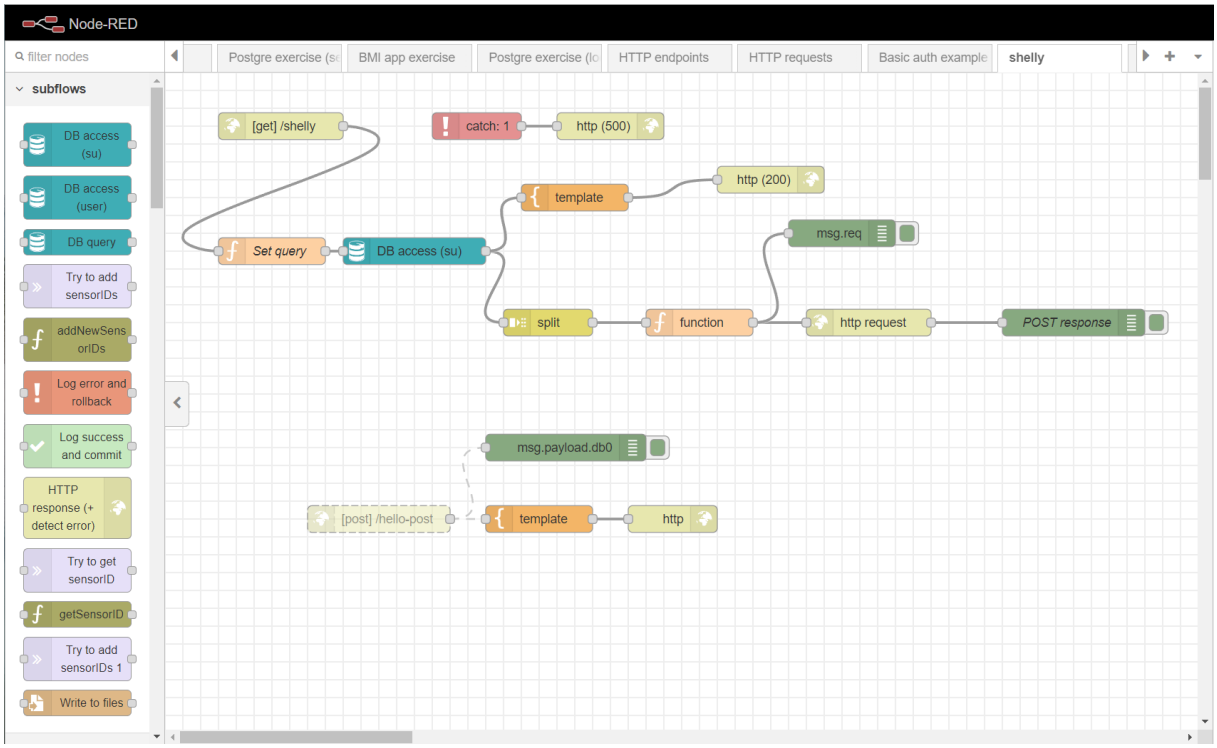


Figure 18. IoTool node-red rules and connectors