



H2020-ICT-2018-2-825040



Rational decomposition and orchestration for serverless computing

Deliverable 5.3 Technology Library

Version: 1.2

Publication Date: 30-June-2020

Disclaimer:

The RADON project is co-funded by the European Commission under the Horizon 2020 Framework Programme. This document reflects only authors' views. EC is not liable for any use that may be done of the information contained therein.

Deliverable Card

Deliverable	5.2
Title:	Template Library
Editor(s):	Matija Cankar (XLAB)
Contributor(s):	Michael Wurster (UST), Matija Cankar (XLB), Anže Luzar (XLB), Neža Vehovar (XLB), Pelle Jakovits (UTR), Hans Georg Næsheim (PRQ)
Reviewers:	Damian A. Tamburri (TJD), Domenico Presenza (ENG)
Type:	Report
Version:	1.2
Date:	30.6.2020
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No

Executive summary

This deliverable introduces the technology library core concepts for handling the reusable content in DevOps and the necessary abstractions defined in RADON and designed according to the user requirements. As Such, the technology library comprises two parts. One is the function artefact manager called Function Hub, which serves the function packets organised by versions. The second is IaC template management serving TOSCA module templates and TOSCA service templates. The latter is covered by the Template Library services, such as RADON particles repository, and Template Library Publishing service. The technology library addresses all the reusable content to develop a RADON application.

Both the aforementioned parts are presented with the main objectives, concepts and examples of how to use them through standard interfaces, such as REST API or CLI. The results presented in this deliverable are still in an alpha version which means that the final form and the final content of the technology libraries, will be presented in the consecutive deliverable.

Glossary

RADON Module	A TOSCA entity type. This is one representative from the set of node types or policy types, etc.
Template Library	An umbrella term for managing (storing) the templates (entity or service).
RADON Particles	A public repository example of Template Library
TPS	Template (Library) Publishing Service
Application blueprint	TOSCA service template, application composed with TOSCA Modules.

Table of contents

Introduction	7
Deliverable objectives	7
Overview of main achievements	7
Structure of the document	8
Technology library	9
Template library - IaC content manager	9
Concepts of the template library	10
Individual users - local storage	11
Open communities and academic users - versioning system, public repository	12
Advanced and enterprise template management - publishing service	12
RADON Particles - Template library public repository	13
RADON Template library publishing service (TPS)	14
Objectives and access	14
Architecture	15
Template library publishing service CLI	16
REST API	17
REST API design	17
Service access and security	19
REST API usage	21
Implementation	25
Template library content	26
The organisation of the content	26
FaaS abstraction layers	26
AWS modules	26
Azure modules	27
GCP modules	27
OpenFaaS	28
Data Pipelines	28
Function Hub	30
Purpose	30
Architecture	30
API	31
Content	32

Examples	33
Conclusions	34
Current requirement fulfilment status and future work.	35
References	36
Appendix A	37

1. Introduction

In recent decades, application development has been changed drastically. The true power and effectiveness of the developer is not only in the language that she uses, but also in the framework and related tools, which provide an environment with a toolset and templates that minimize the time to develop. This time to achieve a result, for an MVP or final application, can be crucial for making business decisions and consequently the choice of technology.

As the executive summary of this document summarizes, this deliverable is dedicated to the initial versions of the template library and function hub, which are live catalogues of FaaS and IaC content. Each new piece into those two catalogues contributes to the RADON framework usefulness and maturity. The content is a foundation for new applications and provides an abstraction of microservice application development approach, giving developers an opportunity to focus on relations of specific services instead of the services itself.

1.1. Deliverable objectives

The main objective of the deliverable is to present the current status of the work and research done in the management of IaC and FaaS content. For both a significant push was made based on the requirements from the potential users and also service integration experts.

Objectives of this deliverable are:

- Present the general concepts of RADON Technology libraries, focusing on:
 - Template library
 - Function Hub
- For each tool provide a particular example and describe:
 - The user approaches to the tools.
 - The concepts used to develop the tools.
 - Currently available services based on the development.
 - The current content stored in the services.
 - Presentation of interfaces, APIs, examples of use.
- Overview of achieved requirements Y2.

1.2. Overview of main achievements

- Designing the RADON Template library context - with constant tracking of WP6 (use-case) requirements, and following the WP4 progress on designing the best practices of entity type development resulted in requirement and functionality updates of template library in WP2 and finally, this whole process results in the balanced results described in this deliverable.
- The Alpha version of Template library publishing service (TPS) following the requirements from other WPs, end-users and experiences gained from RADON particles repository.

- Providing a TPS API to the crucial services to allow integration of the tool inline with the integration plan designed in WP2.
- Deploying TPS service on a publicly available endpoint providing the availability of the services to the rest of the consortium.

1.3. Structure of the document

The document continues with a short intro of Technology library and explains the reasons for the way that the templates therein were developed. The deliverable intentionally does not get into the same level of detail presenting the Function Hub, as this service is already presented in detail in previous deliverables such as [RADD6.1]. Section 3 presents RADON particles and section 4 focuses on TPS. Section 5 makes an overview of the Template Library content and introduces abstraction layer concepts. Section 6 is dedicated to Function hub, while section 7 concludes the document.

2. Technology library

In the RADON terminology, the technology library is an umbrella term for the management of application definitions, configurations - both known as IaC artefacts and TOSCA service templates - and application (function) code ([Figure 1](#)). The application code artefacts are managed by a specialised service called Function hub, while the module templates and service templates are managed by Template library.

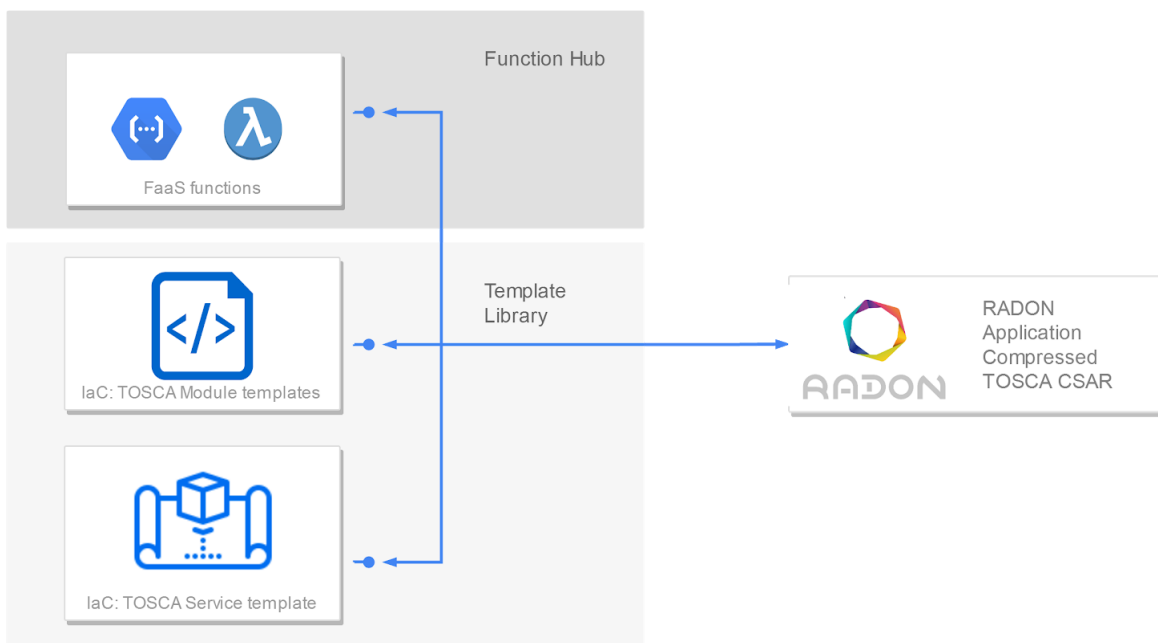


Figure 1 - Technology library components

2.1. Template library - IaC content manager

A set of RADON technology library deliverables will provide the overviews and updates of Template Library releases. RADON Template library is a centrepiece of storing, sharing and publishing the following content:

- *TOSCA artefacts, module templates as entity types.*
- *Implementations* that provide the functionality to the TOSCA artefacts and can be expressed in one of the orchestration/automation languages. In the case of RADON, ansible playbooks are used.
- A FaaS abstraction layer(s), which consists of groups/sets of TOSCA templates including the implementations to support a specific technology or provider. The abstraction layer

allows users to develop applications using reusable templates with no or significantly lower effort of playbook writing.

- TOSCA service templates, frequently Blueprints

Beside mentioned content, a library can also have user and user group possibilities to provide the possibilities of sharing the content in a managed way.

2.2. Concepts of the template library

The DevOps engineers follow the process of creating the application blueprints and using them to deploy application artefacts. One initial question arises, where to store the content describing the application. The TOSCA models and blueprints are managed as IaC (infrastructure as a code), therefore approaches used for software development are appropriate.

Developers create content incrementally in small steps that provide them with the ability to start with a small runnable result, which can face unit tests. Through this test-driven approach, developers improve small applications until they reach the desired set of functionalities, by solo development or teamwork. The final result is shared to the targeted users. A similar path of the development was taken into consideration while we planned the tools for IaC content management. The result of the miniature development research and collaboration among WP4 and WP5 unveiled us that users work with three different content storages during the development - *local storage*, *versioning system* and *publishing service*. Note that the publishing service allows to target different user approaches and requirements, and contributes to the maturity of the content. In this paragraph we propose three different approaches to the use of template library and corresponding concepts presented in [Figure 2](#).

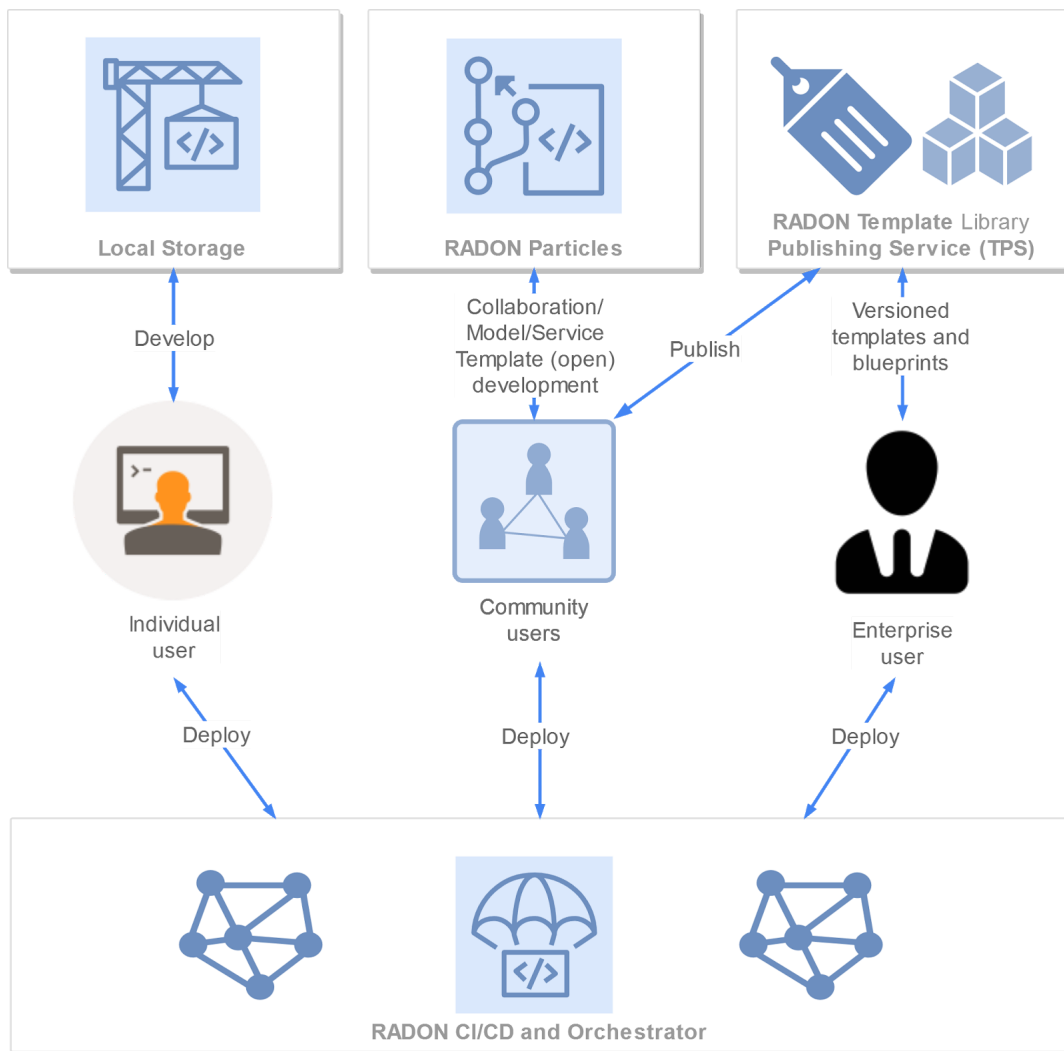


Figure 2 - Template library concepts

2.2.1. Individual users - local storage

Software developers, testers and small office administrators usually have their own stored environment with all their tools and scripts that make their life easier. This is usually a maintained directory structure stored offline or list of dependent software, backed up on repositories or local backup systems. Although this approach does not provide sharing of the environment in a good way, it is very common when starting to use a new technology or use it in the initial stage of the development. For this kind of use, the sufficient template library storage can be a folder of TOSCA model templates, their implementations and TOSCA service templates. Even with this approach users can use orchestrators as Opera, make changes on blueprints and deploy complex applications. However, this approach seems a bit primitive and not in the focus of RADON, but it cannot be totally neglected as there are user's groups exploiting this approach: a) developers in initial stages, b) testers while performing manual tests for triaging the issues, c) administrators of small businesses d) home users, e.g. smart home enthusiast organising their FaaS applications.

2.2.2. Open communities and academic users - versioning system, public repository

Working in teams is natural and this requires sharing TOSCA models and blueprints between developers and DevOps engineers. In this case, an organised folder of blueprints in an online repository can be shared with relevant stakeholders. One example of this kind of library is [RADON Particles](#)¹, published on the GitHub repository and presented in detail in section 3. This open repository of TOSCA module templates allows users to share, contribute to and use templates. This segmented approach is important for module developers that can access all the history of the IaC development and tie it with unit/integration test workflows.

2.2.3. Advanced and enterprise template management - publishing service

Advanced users, enterprises and business-oriented actors require a more advanced TOSCA module and TOSCA service templates that are packed, versioned and annotated with metadata. The users work with abstracted catalogues, and they should not need to face the code complexity or use repository management commands to get the TOSCA service templates. Still, the solution needs to be able to manage the content with similar functionalities as before. Complex applications based on TOSCA models can result in strict dependencies that tightly couple specific groups of modules and software. Updated TOSCA modules can result in new issues, as a consequence, those that can make related applications defective. The template library publishing server simplifies the complexity of managing multiple versions of TOSCA artefacts and does not permit changes of the published content. This approach persuades users to organise each modification in a new version. Adding identity management, private content store and sharing capabilities unveil new potentials of the publishing service. This sets the basis for the business model of on-key development and selling the TOSCA models and TOSCA service templates.

In the following chapters, we focus on the last two presented concepts, presenting RADON Particles, a repository-based Template Library, and the Template (Library) Publishing Service, which is a publishing service variant of the Template Library.

¹ <https://github.com/radon-h2020/radon-particles>

3. RADON Particles - Template library public repository

The RADON Particles is a public repository and, essentially, serves two main use cases: (i) it is an example of a public Template Library repository and (ii) the RADON consortium uses this repository to publish all developed TOSCA entity types (including Ansible Playbooks) and RADON Models that were used during lab validation to provide a comprehensive modeling baseline to the community and a set of demo applications that can be used as examples. The RADON Particles follow the organizational layout and structure required by the RADON Models (cf. D4.4 RADON Models II). It is maintained by the consortium and publicly available on GitHub:

<https://github.com/radon-h2020/radon-particles>

In general, the RADON Particles contain TOSCA entity types and templates to deploy and manage RADON applications. It provides reusable TOSCA types of application runtimes, computing resources, and FaaS platforms in the form of abstract as well as executable TOSCA node types. The repository also comprises RADON's FaaS abstraction layer that provides TOSCA definitions to design particular FaaS application components for AWS, Azure, Google, and OpenFaaS.

The RADON Particles demonstrates how RADON users can use public or private repositories to maintain their custom TOSCA entity types and RADON Models. The consortium uses a Feature-Branch-Workflow using GitHub that essentially shows that approved RADON Models are merged into the master branch, while things under development reside in separate branches or forks for the repository. It is used during the project to develop and approve reusable TOSCA type definitions that can be publicly shared with everyone in the community.

Further, the RADON Particles demonstrate how convenient it is to develop and test new RADON Models as well as TOSCA entity types using the RADON IDE and the Graphical Modeling Tool (GMT). The RADON IDE, for example, is used to create new TOSCA node types and Ansible Playbooks to provide executable modeling entities. Following the proposed Feature-Branch-Workflow, project partners are able to publish executable modeling entities to the community in an open-source fashion. Moreover, these created TOSCA modeling entities that reside publicly in RADON Particles are used by the GMT as modeling baseline. When GMT starts, it gets a copy of the current RADON Particles master to provide a set of reusable modeling entities and demo applications to bootstrap RADON users. Through the GMT, RADON users compose new RADON Models in the form of TOSCA service templates. Further, the GMT is used to package and export a TOSCA Cloud Service Archive (CSAR) required by the RADON Orchestrator for deployment. In future, the GMT will also be able to publish final TOSCA service template and entity types to the RADON Template Publishing Service, which is introduced and described in the next section.

4. RADON Template library publishing service (TPS)

Despite tight coupling of template library and orchestrator proposed from the TOSCA standard[TOS19]², in RADON we decided to decouple it and provide it as a standalone service. This makes the template library more lightweight, specialised and open to be freely integrated with any TOSCA orchestrator by the choice of the user or any other tool operating with TOSCA templates and implementations.

4.1. Objectives and access

The main objective of the Template Publishing Service (TPS) is to store and manage the content. In the nutshell the main objectives of the Template library publishing service should suffice the following requirements:

Advanced content management:

- Reusable content management from storing, versioning and dispatch. The content can be:
 - RADON module templates (TOSCA artefacts),
 - RADON module implementations (Ansible playbooks behind TOSCA artefacts),
 - Application blueprints (a service template created from TOSCA artefacts).

Integrability:

- Providing an API for integration with other tools and services.

Hide complexity:

- User does not need to know git or the language of module implementations. It is enough to be able to download templates and configure/use them.
- A CLI approach, no need to know the API.

Business ready:

- Possibility to integrate the solution with licensing/subscription server.

The presented user requirements and the best practices of content management and publishing from similar services [[Python Package Index](https://pypi.org/)³][[Ansible](https://www.ansible.com/)⁴] lead us to the development of the service focused on publishing IaC content of TOSCA entity templates and service templates. The main service is available online⁵ where there are three endpoints serving

- **Sphinx docs** (<https://template-library-radon.xlab.si/docs/>). This endpoint provides the general documentation and description of TPS together with user manual and examples.

² Find Section 13.1 CSAR Onboarding in [TOS19]

³ <https://pypi.org/>

⁴ <https://www.ansible.com/>

⁵ <https://template-library-radon.xlab.si>

The same documentation can also be found when accessing Template library on template-library-radon.xlab.si.

- **REST API:** <https://template-library-radon.xlab.si/api/>). This endpoint is the REST API server, where API requests are processed.
- **Swagger UI:** <https://template-library-radon.xlab.si/swagger/>) This endpoint provides detailed API documentation with WEB access for issuing the API commands to the Template Library publishing service. This service helps developers to integrate publishing services and allows them to test each command through a web browser.

The services together combine a solution called RADON Template Library publishing service (RADON TPS or shortly TPS). The TPS will be presented in detail in the following paragraphs.

4.2. Architecture

Conceptual design of the TPS is shown on Figure X. On the left side of the figure are applications that use API to manipulate the service, namely Winery (RADON GMT), Web interfaces as SwaggerUI or dedicated page and CLI, which is a client application for TPS. On the right side of [Figure 3](#), there is template library data storage, which basically consists of a relational database (used for storing the metadata) and file or object database, which stores data in the form of template files or templates (this also includes ZIP or CSAR files). The main ingredient enabling access to this data is the Template Library REST API which provides several API endpoints through which the service or users can publish and retrieve the desired data.

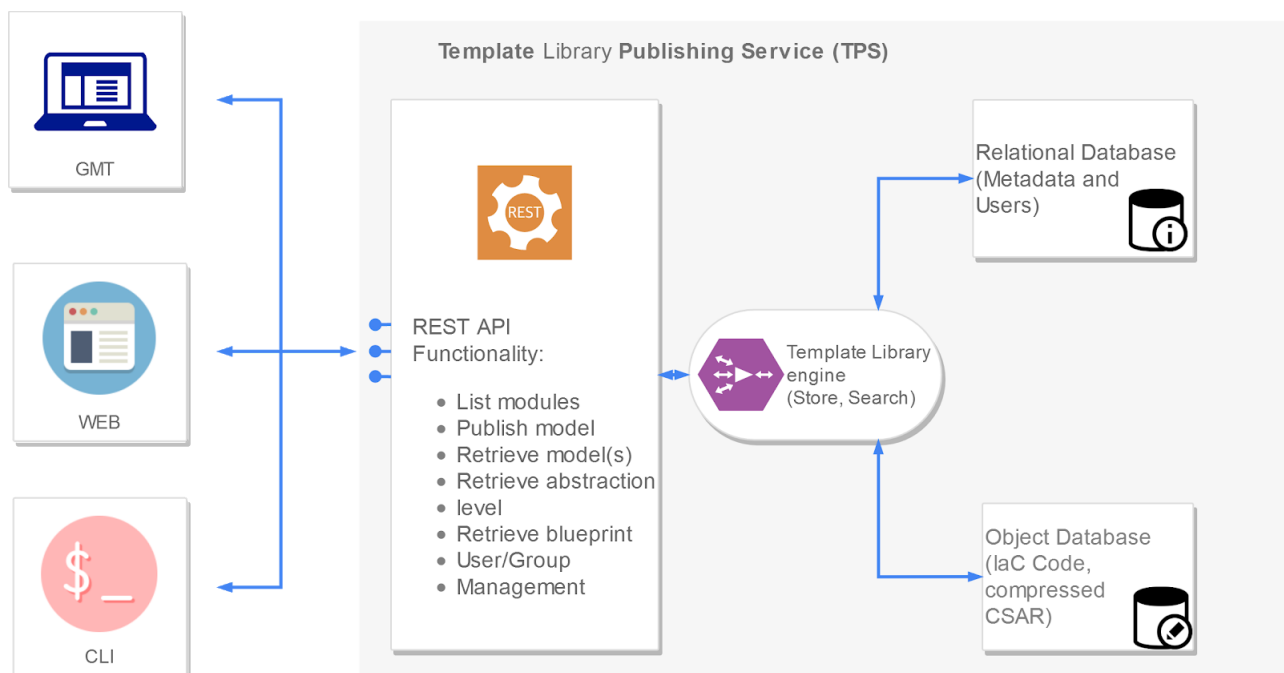


Figure 3 - Conceptual design of TPS

4.3. Template library publishing service CLI

This section explains the usage of Template Library REST Command Line Interface (CLI) which is a client tool for managing templates, blueprints and their versions. [Figure 4](#) is showing all the command-line options for this CLI tool.

```

$ xopera-template-library
usage: xopera-template-library [-h] [-v]
                               {service-template,entity-template,setup,login,logout,create-model,create-blueprint}
                               ...

positional arguments:
  {service-template,entity-template,setup,login,logout,create-model,create-blueprint}
  service-template      Edit a service template.
  entity-template       Edit an entity template.
  setup                 Setup client variables.
  login                 Login to your account.
  logout                Logout of your account.
  create-model          Initialize a model directory and files.
  create-blueprint      Initialize a blueprint directory and files.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Increase output verbosity
  
```

Figure 4 - Template library CLI tool parameters

The client library is called [xopera-template-library](#)⁶ and is distributed through the Python pip package, published on Python Package Index (PyPI). The only thing that needs to be done by the user is to install the package into a virtual environment with `pip install xopera-template-library`. This allows the user to manage the TPS from the command line.

There are several subcommands that allow users to interact with the TPS. In the beginning, the `setup` command should be used in order to configure the API endpoint. Currently, the user account can only be created in Swagger UI or with `curl` with special register credentials. User registration was not included in CLI because of the security reasons. To register a user, the user needs to follow the API instructions from section [Service access and security](#). This temporary step is a workaround to be used until RADON IAM is not integrated with the whole system.

With `create-model` and `create-blueprint` commands, all directories and files that are needed to upload a template, are generated. After generating a model or a blueprint template, the user can update it with his own inputs. Templates can be uploaded using option `save` and downloaded with option `get`. Users can also list public and private templates according to name and view template version information. The template management is split into two commands, `service-template` and `entity-template`, which are the most complex ones. Both have the same set of sub-options that are displayed on [Figure 5](#).

⁶ <https://pypi.org/project/xopera-template-library/>

```

$ xopera-template-library entity-template -h
usage: xopera-template-library entity-template [-h]
                                             {save,get,list,version} ...

positional arguments:
  {save,get,list,version}
  save                  Save a template to database.
  get                   Get a template from database.
  list                  List templates from database.
  version               Version options for templates.

optional arguments:
  -h, --help            show this help message and exit
  
```

Figure 5 - Sub-options for template management

Through CLI users can save templates. User needs to provide a unique name of a template, choose a public or private option (only the user that added a private template can manage it) and version of the template (there can't be two same versions of one template). A certain template can be downloaded from the database by its version id. Users will also need to define whether the template is public or private and the path where the downloaded template directory will be saved.

When users have problems with the TPS CLI the debug mode can be used. This option is enabled with `-v` or `-verbose` global option which will provide more detailed outputs for the executed commands. The concrete usage scenario for TPS Command Line Interface can be found among the examples⁷ that reside in TPS documentation.

4.4. REST API

This section explains the usage of TPS REST API. This REST API is written using Kotlin (KTOR) and provides several endpoints to access data from storage. REST API server is currently accessible on template-library-radon.xlab.si/api/⁸. For testing the REST API an OpenAPI specification is provided and can be publicly accessed through Swagger UI web page on template-library-radon.xlab.si/swagger/. This service will help you to interact with API easily since all the API endpoints can be executed and reviewed from here.

4.4.1. REST API design

TPS REST API was designed in a way that it would be unambiguous and intuitive to use. Therefore its content has been organized into several REST endpoints that can be called and thereby invoked by sending common HTTP requests like GET, POST, PUT and DELETE. To properly arrange the content so that it would be readable for users the related endpoints were grouped together so that it would be evident that they operate on the same database entities. Apart from providing good user experience and linking the related endpoints those endpoint groups (that are further explained in

⁷ <https://template-library-radon.xlab.si/examples.html>

⁸ template-library-radon.xlab.si/api/

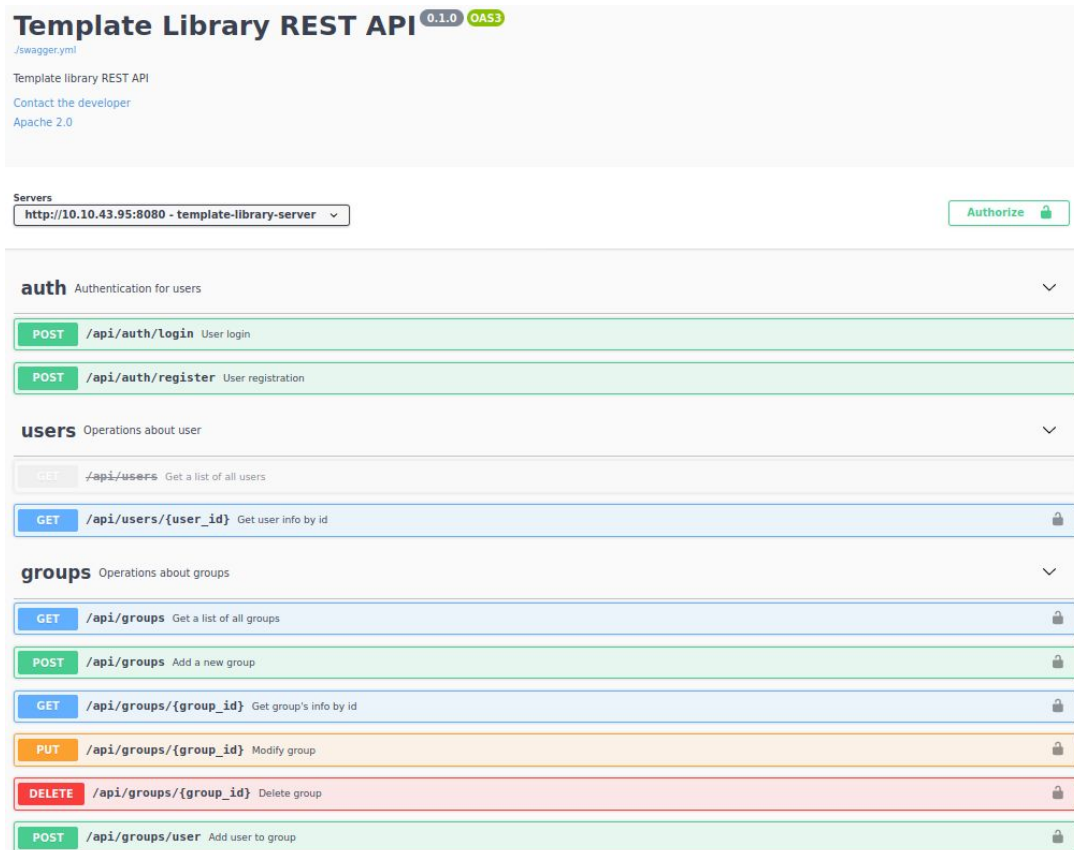
[Table 1](#)) offer different operations and manipulations with the data. Each of them is focused on a different entity and allows storing the data and then retrieving it back. The most important of them all is the auth endpoint group that implements the IAM of the TPS.

Table 1. REST API main endpoints

Endpoint group	Purpose and description
auth	Authentication and authorization for users
users	Operations about users
groups	Operations about groups
template_types	Operations for TOSCA template types
templates	Access to TOSCA templates
versions	Access to versions of templates
export	Operations for exporting data

The requests mostly include JSON objects for transmission of the data. With some requests linked to files, multipart objects can be used.

To achieve the desired state and to provide quality, the TPS REST API was designed using the Swagger OpenAPI specification and for the coding part, OpenAPI generator was used to properly define the entities that are a part of the solution. One small part of this design from Swagger UI editor is shown in [Figure 6](#).



Template Library REST API 0.1.0 OAS3

./swagger.yml

Template library REST API
 Contact the developer
 Apache 2.0

Servers

http://10.10.43.95:8080 - template-library-server Authorize

auth Authentication for users

- POST /api/auth/login User login
- POST /api/auth/register User registration

users Operations about user

- GET /api/users Get a list of all users
- GET /api/users/{user_id} Get user info by id

groups Operations about groups

- GET /api/groups Get a list of all groups
- POST /api/groups Add a new group
- GET /api/groups/{group_id} Get group's info by id
- PUT /api/groups/{group_id} Modify group
- DELETE /api/groups/{group_id} Delete group
- POST /api/groups/user Add user to group

Figure 6 - Template Library REST API design

4.4.2. Service access and security

TPS is available only to registered users. In the future, this registration will be made through a RADON user registration process with Keycloak, but for the sake of testing and early availability of the service in the alpha version of RADON tools, RADON consortium partners can add themselves to the user list through SwaggerUI⁹. Users can register to the TPS via REST API with providing the pre-shared HTTP Basic Auth credentials (username and password) while calling the user registration endpoint. This current workaround is required temporarily for securing the service before RADON identity and authorization management (IAM) will be fully operational. When users obtain their usernames and passwords they can manipulate the TPS through API or CLI.

The process of registration is presented in [Figure 7](#), where basic auth credentials are added in advance and in [Figure 8](#) where credentials are added in prompt after the request is executed. Of course, there is also another possibility by --user switch if curl from the command line is being used.

⁹ <https://template-library-radon.xlab.si/swagger>

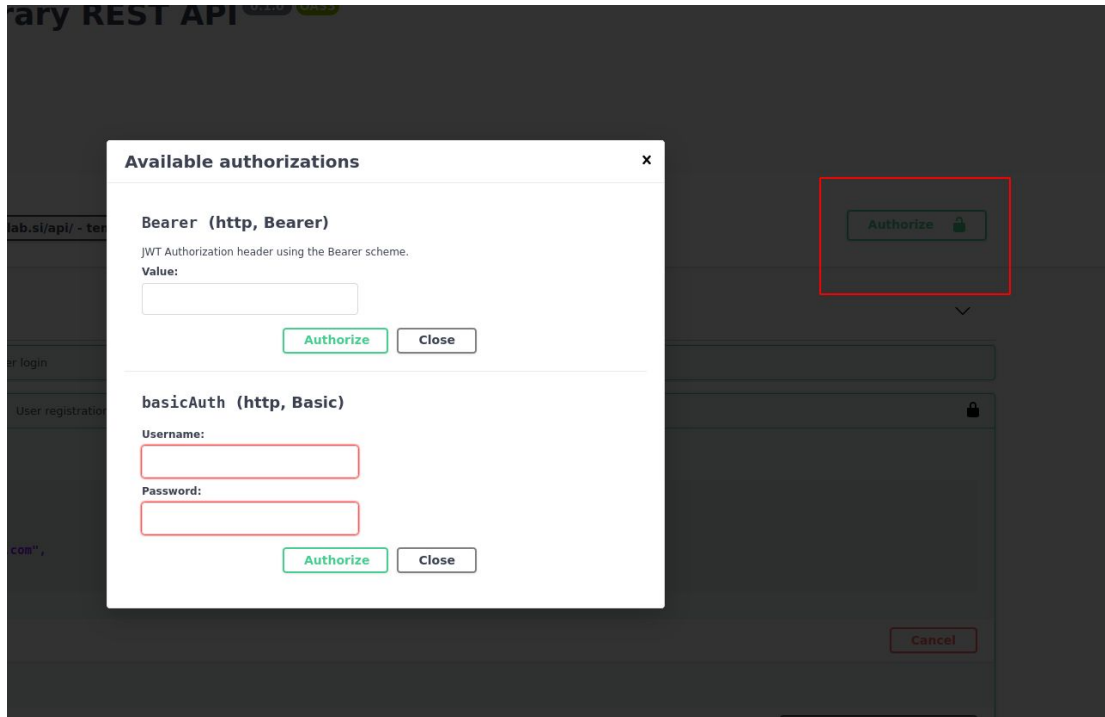


Figure 7 - Basic HTTP Authorization via Swagger UI

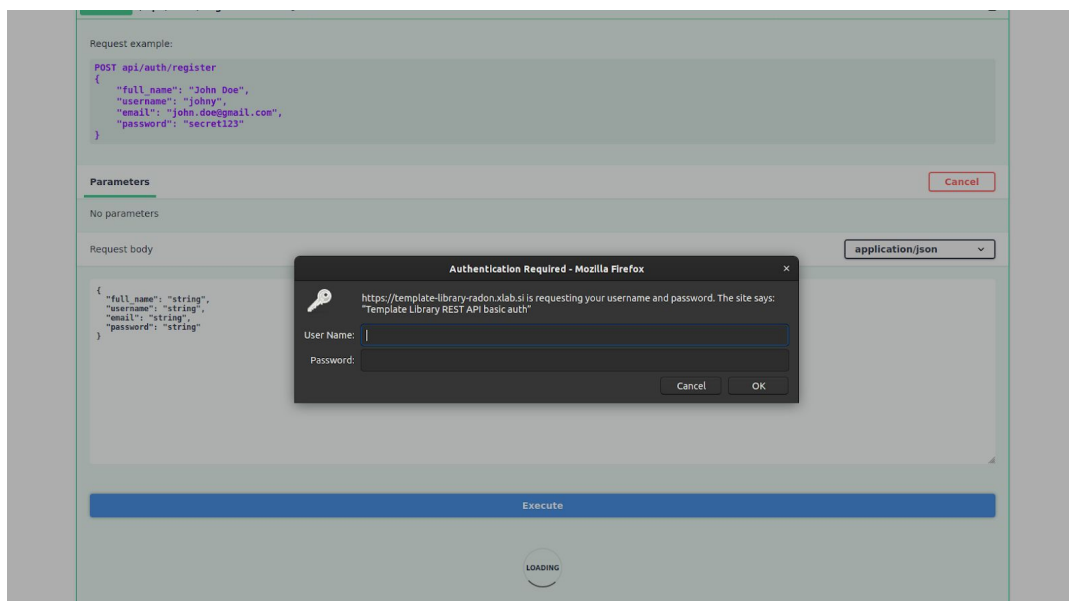


Figure 8 - HTTP Basic Auth via the prompted window

4.4.3. REST API usage

User registration

To be able to use the REST API first a new user needs to be created. This can be done using `api/auth/register` endpoint (POST request on template-library-radon.xlab.si/api/api/auth/register¹⁰). This endpoint is secured and locked by HTTP Basic Auth username and password that have to be provided if one wants to continue with user registration. This can be achieved by taking different ways.

Swagger UI web interface can be used to register a new user manually. Users can click on the green Authorize button and provide basic auth credentials by entering them in the form. Credentials can be also entered later on when trying to register a new user (on `/api/auth/register` endpoint) through a window prompted after the request is executed. Similarly, curl can be used from the command line (like `curl --user username:password`).

When one registers a new user the attributes specified in [Table 2](#) need to be specified in the request:

Table 2. Request body JSON key names for user registration

Parameter	Description
full_name	User's full name (first and last name)
username	User's unique username
email	User's email
password	User's new password (will be securely encrypted with PBKDF2WithHmacSHA512 hashing function)

After a new user has been created, she needs to login using `api/auth/login` endpoint using username and password. After login (SwaggerWeb or CLI) user will receive his Bearer JWT access token which is then used to authorize users to get access to other API locked endpoints. This Bearer token should be put to the Authentication header of the consecutive request. The token is currently valid for 10 hours and after that period user will have to login again and use a new unique token.

For current maintenance and tests, there is a check if the user was added correctly by executing GET request on `/api/users` endpoint in Swagger UI or by curl command. This endpoint is not locked and is marked as deprecated (greyed out) in Swagger UI because it is used only for development and testing.

¹⁰ template-library-radon.xlab.si/api/api/auth/register

Through API group endpoints, registered users can be arranged into groups, where they can share private templates required for a specific DevOps task or inside the team. The TOSCA templates are organised by type. The TOSCA types used in Template Library are shown in table [Table 3](#) and can also be fetched through `api/template_types` endpoint. Among TOSCA types the `csar` and other options are available for managing TOSCA service templates and allowing future compatibility.

Table 3. TOSCA types used in Template library

Parameter	Description
data	TOSCA data type
artifact	TOSCA artifact type
capability	TOSCA capability type
requirement	TOSCA requirement type
relationship	TOSCA relationship type
interface	TOSCA interface type
node	TOSCA node type
group	TOSCA group type
policy	TOSCA policy type
csar	Compressed Cloud Service Archive (CSAR)
other	Other definitions

The template module can be linked to the implementations as Ansible playbooks, Chef recipes, Puppet scripts and others. The implementation code is used by orchestrator and instructs automation tools to actuate commands on targeted providers and instances.

For example, let us prepare an AWS S3 bucket module with a very simple TOSCA template that includes just the node type for the bucket, see [Figure 9](#). The template links to two implementations of Ansible playbooks which are also part of the module. The first one is create operation - `create.yml`, [Figure 10](#) - and the second playbook is delete operation (`delete.yml` - [Figure 11](#)) which is used for *undeployment*.

```

...
tosca_definitions_version: tosca_simple_yaml_1_3

node_types:
  radon.nodes.s3_bucket:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
      bucket_name:
        type: string
        description: The name of the bucket
      aws_region:
        type: string
        description: AWS region
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        inputs:
          bucket_name: {default: { get_property: [SELF, bucket_name] }, type: string }
          aws_region: {default: { get_property: [SELF, aws_region] }, type: string }
        operations:
          create: playbooks/create.yml
          delete: playbooks/delete.yml
...

```

Figure 9 - AWS S3 bucket TOSCA template example

```

...
- hosts: all
  gather_facts: no
  tasks:
    - name: "Create AWS S3 bucket {{ bucket_name }}"
      s3_bucket:
        name: "{{ bucket_name }}"
        state: present
        region: "{{ aws_region }}"
...

```

Figure 10 - Ansible playbook for the creation of AWS S3 bucket

```

...
- hosts: all
  gather_facts: no
  tasks:
    - name: "Remove S3 bucket {{ bucket_name }}"
      s3_bucket:
        name: "{{ bucket_name }}"
        region: "{{ aws_region }}"
        state: absent
        force: yes
...

```

Figure 11 - Ansible playbook for the creation of AWS S3 bucket

These three files along with the metadata form a TOSCA module. This new module can be later uploaded to the Template Library, but first a place for the template, using `api/templates` endpoint and the information shown in [Table 4](#), needs to be created:

Table 4. Request body JSON key names for adding a new template

Parameter	Description
<code>shorthand_name</code>	Short module name (e.g. <code>aws_bucket</code>)
<code>type_uri</code>	Long module name (e.g. <code>radon.nodes.aws_bucket</code>)
<code>template_type_id</code>	Template type identifier
<code>public_access</code>	Specify if module should be private/public (private templates are only visible in groups)

The actual template files (TOSCA YAML files) and implementation files (Ansible playbooks) are added later creating a new version of your module. To add the template to the existing group `api/groups/template` API endpoint can be used. This step enables group members to access private templates inside the group. Public templates are globally accessible while private templates without group membership are solely visible to their creator.

Version endpoints add new IaC components to the templates. Each version of the module is completely unique and immutable and therefore previous versions cannot be changed. This idea of everlasting module versions originates from Python Package Index (PyPI) which was the inspiration when designing the Template Library solution. In PyPI every Python package version is fixed. Even if some older versions of the package are dormant and are not being used anymore they are still present so users can go back and maybe revitalize some forgotten features. To add your new version of the template the POST request on `api/version/` should be used and the request attributes in [Table 5](#) have to be provided:

Table 5. Request body JSON key names for adding a new template

Parameter	Description
<code>template_id</code>	Id of your created template
<code>version</code>	Unique template version
<code>template_file</code>	TOSCA YAML template file
<code>implementation_file</code>	Array of Ansible playbooks as YAML files

Newly published versions can be downloaded using `api/versions/files` endpoint.

Tools like RADON GMT require downloading the whole set modules and templates. Exporting all the TOSCA templates and implementations can be done using `api/export` REST API endpoint which will prepare a compressed file with all templates and implementations sorted by their versions. Similarly, you can use it to retrieve the data from `/api/export/tosca_types` where template data is organized by TOSCA types or `/api/export/groups` which organizes data by groups.

4.5. Implementation

Similar to modern applications, this TPS is designed as a microservice application containing multiple independent container services. The set of services is comprised of (i) relational database for storing the metadata and managing users; (ii) object database for storing TOSCA templates, Ansible playbooks and CSARs; (iii) REST API written using KTOR in Kotlin (iv) OpenAPI specification to describe REST API endpoints; (v) documentation service and (vi) service traffic proxy. Each microservice runs in a separate container which makes the application more robust and scalable.

The implementation of the service was achieved by using different technologies. Sphinx documentation tool is used to render our RST documentation files with a Sphinx Documentation server. The Cloud Native Edge Router called Traefik is used to adapt the solution to the outer world. Traefik reverse proxy gets deployed as a docker container and it connects to other TPS microservices by mounting the docker sockets and enabling setting via docker labels on other containers. Data management is covered by relational and object store databases, to get the best of both options. A relational database is used for organising and managing users and metadata, while object storage is used for saving the versions of templates, like TOSCA YAML definitions and their implementations. The TPS service is packaged as a TOSCA service template (in YAML profile v1.3), powered by Ansible and deployable by xOpera orchestrator. We also use GitLab Continuous Integration (CI) & Continuous Delivery (CD) to build, test, deploy and backup the solution.

5. Template library content

5.1. The organisation of the content

The content organisation was already mentioned in deliverables D4.4 and D4.2, where RADON particles were introduced. Adding the RADON TPS adds new parallel storage, but the organisation of TOSCA entity types remains the same. The TOSCA service templates are accessed through TPS in the same way than entity types. One difference from the previous versions of the template library is that TPS will focus on grouping entity types by domain to form *abstraction layers*. Each abstraction layer has its own topics and in the RADON project, we decided to provide FaaS abstraction layers forming entity type implementations covering major FaaS providers. With abstraction layers, users can use and configure only TOSCA templates, which brings them a unique interface to the management FaaS on different providers.

5.2. FaaS abstraction layers

TPS is in early alpha version and already contains an initial set of TOSCA modules and blueprints. TOSCA modules are currently organised by providers, which is a way to form FaaS abstraction levels. From a simple command in TPS CLI a user can get access to a current list of AWS and OpenFaaS templates.

```

$ xopera-template-library service-template list | grep -e Name
shorthandName: AwsBucket
shorthandName: AwsLambda
shorthandName: AwsBucketNotification
shorthandName: AwsRole
shorthandName: AwsApiGateway
shorthandName: AzureContainer
shorthandName: AzureContainerNotification
shorthandName: AzureFunction
shorthandName: MinIOBucket
shorthandName: OpenFaaSFunction
shorthandName: OpenFaaSFunctionBuild
shorthandName: DemoBlueprintAws
  
```

Figure 12 - List of current TOSCA templates available in RADON TPS.

The detailed description of the TOSCA blueprints in TPS will be described in the following sections

5.2.1. AWS modules

AWS modules cover the necessary definitions to create and deploy FaaS applications on AWS Lambda. The demo example of such an application is available on GitHub¹¹. Within this use case

¹¹ <https://github.com/radon-h2020/demo-tosca-blueprint-aws-lambda>

modules for Amazon Web Services are provided. Published versions of the modules are available in demo TOSCA AWS blueprint Lambda repository and the related code and prepared CSAR is accessible in RADON particles repository¹². The modules were also published to the public Template library instance where we included the modules shown in the [Table 6](#):

Table 6. AWS modules that were published to the Template library

TOSCA template	Description
AWS role	Creates a new AWS role
AWS S3 bucket	Creates a new AWS S3 bucket
AWS lambda	Uses a zip file with function and deploys it to AWS Lambda
AWS bucket notification	Creates bucket notification for triggering the lambda
AWS API Gateway	Prepares Swagger YAML file and deploys a new API Gateway

5.2.2. Azure modules

Here we developed TOSCA modules Microsoft Azure cloud which is accessible in the repository on GitHub: <https://github.com/radon-h2020/demo-tosca-blueprint-azure-function> where the image-resize blueprint resides. The modules listed in the [Table 7](#) and their implementations were published to the TPS:

Table 7. Azure modules that were published to the Template library

TOSCA template	Description
Azure container	Creates storage account and necessary container on specified Azure storage account
Azure function	Creates FunctionApp and deploys a new function to Azure portal using Azure CLI
Azure container notification	Creates an event subscription trigger for function

5.2.3. GCP modules

Google Cloud Platform (GCP) modules were not chosen to be available for the alpha version of the RADON and therefore they are currently in the development backlog. These modules will be fully

¹² <https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.legacy.blueprints/ImageResize>

implemented within the thumbnail generator application use case and published to the Template library within the next few months.

5.2.4. OpenFaaS

The OpenFaaS provider is usually a self-hosted set of instances. The platform can be easily designed as a TOSCA service that can be deployed as a docker container on your IaaS provider. OpenFaaS only provides function management, therefore MinIO object storage was used for storing the important application data. The image-resize functionality was adapted to OpenFaaS platform and the solution consists of the templates in [Table 8](#). The repository with the application modules is used in this prepared application and is available on GitHub: <https://github.com/radon-h2020/demo-tosca-blueprint-openfaas>. The [Table 8](#) shows the OpenFaaS TOSCA modules that were supplied to the TPS.

Table 8. OpenFaaS modules that were published to the Template library

TOSCA template	Description
Docker	Installs docker on a target machine
OpenFaaS	Sets up OpenFaaS in a separate docker container on a VM
MinIO	Sets up docker container for MinIO object storage
Function load	Loads the given docker image to machine
Function deploy	Deploys docker image to OpenFaaS as a function
MinIO bucket	Creates necessary buckets on MinIO serve
MinIO bucket notification	Creates notification on bucket and configures MinIO server

5.2.5. Data Pipelines

In addition to the pure FaaS approach, we also designed a data pipeline approach for utilizing FaaS functions, where the flow of data or events is controlled by the data pipeline and the FaaS function is executed directly. This allows more elaborate control over the flow of data and enables multi-cloud and hybrid cloud scenarios where FaaS functions deployed across multiple clouds and even on-premise can be composed into a single data pipeline. To this end, a set of data pipeline TOSCA node types and related TOSCA elements were designed and made initially designed available in the thumbnail generation with data pipelines demo Github repository:

<https://github.com/radon-h2020/demo-lambda-thumbgen-tosca-datapipeline>.

The data pipeline modules were designed to be fully reusable TOSCA elements, which can easily be connected to each-other for composing larger data pipelines. Furthermore, a set of abstract and generic TOSCA node types were created which define the generic properties, capabilities, requirements and interfaces of the data pipeline modules to facilitate more rapid development of additional reusable data pipeline module modules. The modules are outlined in [Table 9](#) and published in RADON particles repository.

Table 9. Data pipeline modules published in RADON particle repository

TOSCA template	Description
NiFi	Installs Apache NiFi on a target machine
ConsS3Bucket	Pipeline block for consuming data from an AWS S3 bucket and forwarding it to the next pipeline block.
ConsumeLocal	Pipeline block for consuming data from a local file system folder and forwarding it to the next pipeline block.
AWSLambda	Pipeline block for sending incoming data to an AWS Lambda function and forwarding it to the next pipeline block.
PubsS3Bucket	Pipeline block for storing incoming data to an AWS S3 bucket.
PublishLocal	Pipeline block for storing incoming data to a Local file system folder.
ConnectNifiLocal	Pipeline block relationship implementation for dynamically creating connections between two pipeline blocks using NiFi REST API.

6. Function Hub

6.1. Purpose

As Serverless and FaaS are becoming more and more widespread, the need for standardization and common solutions is increasing. Function Hub (FH) is a package manager for generic, reusable functions in the scope of FaaS. Similar to existing package managers, FH also supports basic functionality like user handling, private and public repositories and deployment and resolving of Functions. In order to enforce a common format of these functions, we have built a Function Hub client. This client reads a configuration file containing the function metadata and uploads these data with the function package.

6.2. Architecture

Praqma's Use Case, Serverless Artifact Manager(cloudstash.io), is a 'FaaS driven' product, solely containing Serverless resources. FH being a sub subsection of cloudstash.io, naturally shares the architectural design and structure. This is more thoroughly elaborated in deliverable D6.1.

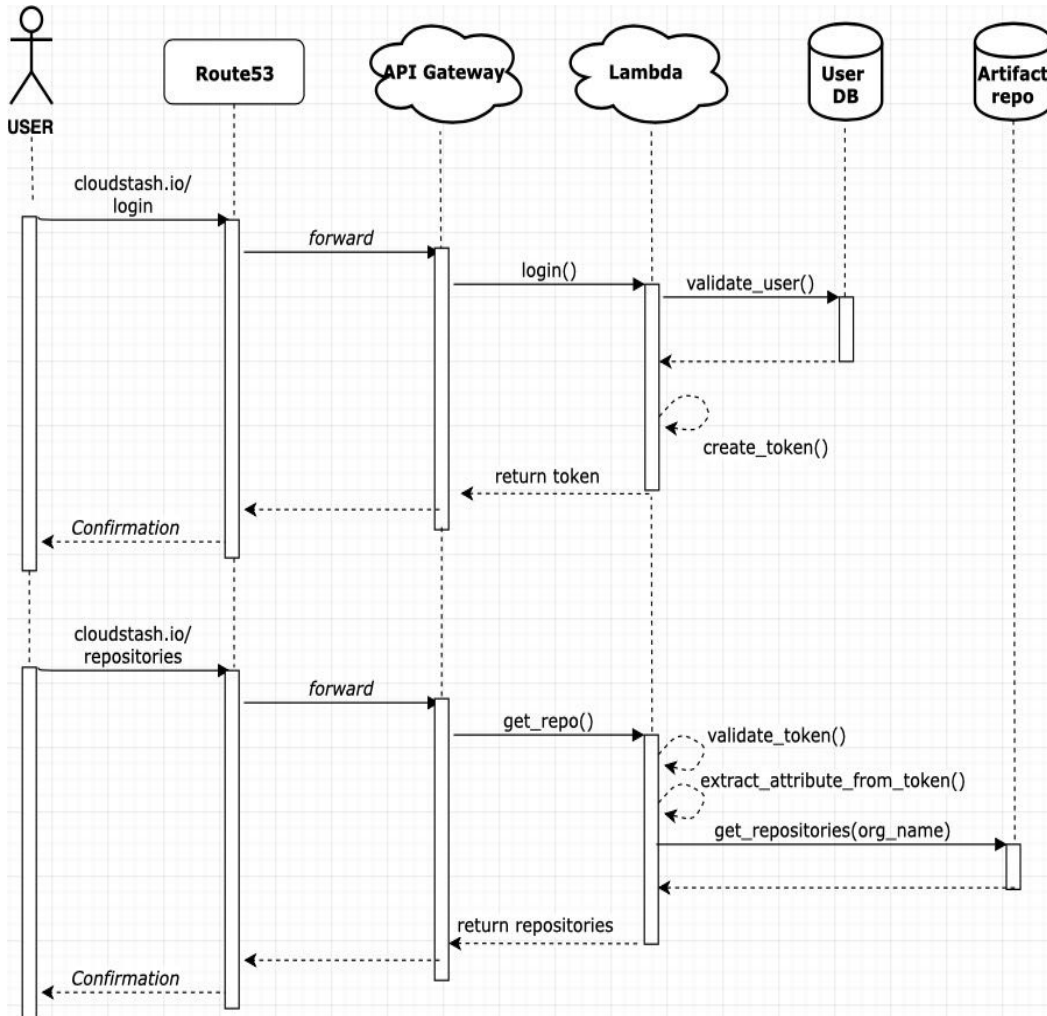


Figure 13 - Sequence diagram for user interaction with FunctionHub

The illustration in [Figure 13](#) depicts a user interacting with FunctionHub attempting to login with credentials and further list the repositories related to his user. This also shows which serverless cloud resources are passed through as the user request is handled.

6.3. API

FunctionHub being a ‘FaaS driven’ product implies that the functionality supported by the product is available through Functions. In order to structure these features and functionalities, we use an API Gateway, which forwards HTTPS requests to the specific Lambda function. Which then access and return the relevant data from the databases.

```
list-repositories:
```

```
  path: /repository
```

```
  method: GET
```

```
upload-artifact:
```

```
    path: /artifact
    method: POST
create-repositories:
    path: /repository
    method: POST
list-artifact:
    path: /repository/{repo_id}
    method: GET
artifact-details:
    path: /artifact/{artifact_id}
    method: GET
create-user:
    path: /signup
    method: POST
login:
    path: /login
    method: POST
logout:
    path: /logout
    method: GET
```

Examples of available API paths.

6.4. Content

The target platforms of these functions are major cloud vendors like AWS, Azure and Google Cloud, in addition to open-sourced solutions like OpenFaaS. FH is a place to store packaged, versioned ready to deploy-Functions in a plug and play manner. As mentioned above, we are creating a pool of reusable Functions relevant to companies across market domains, mostly focusing on cloud security and maintenance. In order to achieve this, we are also encouraging the open community to contribute to the pool available Functions. In addition to the public scope, we are also focusing on private repositories.

```
[REPOSITORY]
    org =
    repository =
[FUNCTION]
```

```
name =  
version =  
description =  
[RUNTIME]  
provider =  
runtime =
```

Example snippet of config file being read upon upload. The user has to specify all fields in order to upload Function.

6.5. Examples

FunctionHub is slowly accumulating content. Simpler, generic Functions has been added in order to validate the different cloud vendors and runtime. Such Functions is the commonly used Toy Application. Within the scope log, security and automated monitoring, there are some available Functions. One of these can be found at cloudstash.io under the public repository ‘radon-functions’ called ‘aws-instance-clean’ and ‘admin-trawler’.

Instance Cleanup: For a big company with an open access for developers to create resources, the need for automated screening is high. As a solution, this Function traverses active virtual machines within a company profile and shuts down the ones not following internal policy. Such internal policy can be complying to fixed tags, specifying creator and purpose, etc.

Admin Trawler: Admin roles in any service tend to be a bottleneck in every organization. Cloud platforms are no exception. In a fast-moving environment, it is tempting to give users elevated privileges. That always comes at the cost of security and failing to comply with infrastructure standards. Admin trawler is a function that goes through a company cloud environment and lists the current admins.

7. Conclusions

This deliverable gives an overview of the work done by the RADON partners inside the T5.2 task. The given results demonstrate RADON technology library services, namely Function Hub and Template Library, including the RADON Particles and Template Library Publishing Service (TPS). Both service are in alpha version and available online to the rest of the RADON consortium. In the following development period, the focus will be on the integration of services, improving user experience and providing the fresh content.

[Table 10](#) shows an overview of the level of fulfilment for each of the agreed requirements. The labels specifying the “Level of fulfilment” are defined as follows:

- (i) ✘ (unsupported): the requirement is not fulfilled by the current version
- (ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version
- (iii) ✔✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version
- (iv) ✔✔✔ (fully supported): the requirement is fulfilled by the current version.

Table 10. Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Level of compliance
R-T5.1-6	Support of FaaS deployment to OpenFaas	MUST_HAVE	✔✔✔
R-T5.1-7	Support of FaaS deployment to AWS cloud platform	MUST_HAVE	✔✔✔
R-T5.2-8	Support of FaaS deployment to Google Cloud Platform	COULD_HAVE	✘
R-T5.2-9	Support of FaaS deployment to Azure cloud platform	MUST_HAVE	✔✔✔
R-T5.2-11	Support deployment to microservices architecture	MUST_HAVE	✔✔
FR-T5.2-12	Template library publishing service filtering of entities	MUST_HAVE	✔
FR-T5.2-13	Template library publishing service	SHOULD_HAVE	✘

	should be RADON IAM compliant		
FR-T5.2-14	Generating a basic entity with template library CLI	COULD_HAVE	X
FR-T5.2-15	Publishing and retrieving entities with template library CLI	COULD_HAVE	✓✓
FR-T5.2-16:	Listing versions of a template with template library CLI	MUST_HAVE	✓✓
R-T5.3-3	The tool must be able to support configuring AWS EC2 auto-scaling service based on the TOSCA auto-scaling policy.	SHOULD_HAVE	X

Current requirement fulfilment status and future work.

The levels of compliance were achieved by prioritizing the requirements in a way to give the partners of the RADON consortium enough freedom to develop their own tools. Therefore some “MUST_HAVE” priorities are not fully supported yet, as they are not desperately required at this stage, but need to be fulfilled until the end of the project.

As table 10 presents, FaaS templates for AWS(R-T5.1-7), Azure (R-T5.2-9) and OpenFaaS(R-T5.1-6) already cover all required basic functionalities. Still, we lack functionalities for managing the templates in TPS (FR-T5.2-15, FR-T5.2-16, R-T5.2-12) and related commands improving user experience. The major push in the development in the next period will be required to close the issues focusing on scaling (R-T5.3-3), GCP support (R-T5.2-8), and integration (FR-T5.2-13).

8. References

[RADD6.1] RADON Consortium deliverable D6.1: Validation Plan

[RADD6.2] RADON Consortium deliverable D6.2: Initial Validation results

[TOS10] TOSCA Standards v1.3 available online:

<https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/csprd01/TOSCA-Simple-Profile-YAML-v1.3-csprd01.html>

9. Appendix A

This appendix lists all TPS REST API endpoints sorted by their groups. The API documentation is available on <https://template-library-radon.xlab.si/swagger/#/> :

AUTH:

- POST /auth/register
- POST /auth/login
- GET /auth/logout

USERS:

- GET /users
- GET /users/user
- PUT /users/user
- DELETE /users/user
- GET /users/user/{user_id}
- GET /users/user/name/{username}
- GET /users/group/{group_id}
- GET /users/group/name/{group_name}

GROUPS:

- GET /groups
- POST /groups
- GET /groups/{group_id}
- PUT /groups/{group_id}
- DELETE /groups/{group_id}
- GET /groups/name/{group_name}
- PUT /groups/name/{group_name}
- DELETE /groups/name/{group_name}
- GET /groups/user
- POST /groups/user
- DELETE /groups/user
- GET /groups/user/{user_id}
- GET /groups/user/name/{username}

- POST /groups/template
- DELETE /groups/template
- GET /groups/template/{template_id}
- GET /groups/template/name/{template_name}

TEMPLATE_TYPES:

- GET /template_types
- GET /template_types/{type_id}
- GET /template_types/name/{type_name}

TEMPLATES:

- GET /templates
- POST /templates
- GET /templates/{template_id}
- PUT /templates/{template_id}
- DELETE /templates/{template_id}
- GET /templates/name/{template_name}
- PUT /templates/name/{template_name}
- DELETE /templates/name/{template_name}
- GET /templates/user
- GET /templates/user/{user_id}
- GET /templates/user/name/{username}
- GET /templates/group/{group_id}
- GET /templates/group/name/{group_name}
- GET /templates/template_type/{type_id}
- GET /templates/template_type/name/{type_name}

VERSIONS:

- POST /versions
- GET /versions/{version_id}
- DELETE /versions/{version_id}
- GET /versions/files/{version_id}
- GET /versions/template_file/{version_id}
- GET /versions/template/{template_id}
- GET /versions/template/name/{template_name}

- GET /versions/implementation/{implementation_id}
- GET /versions/implementation/file/{implementation_id}
- GET /versions/implementations/version/{version_id}

EXPORT:

- GET /export/tosca_types
- GET /export/groups