Domain Registry User Manual

Version 1

February 2016

INESC-ID

Introduction

This document provides instructions on the use of a reThink's architecture component named Domain Registry. Together with the Global Registry, it forms the Registry Service. The Global Registry provides a mapping from a User's Unique Id (GUID) to the several services it uses. Each Communication Service Provider (CSP) runs a Domain Registry service that resolves domain-dependent user identifiers to the actual information about this user's Hyperty Instances (a Hyperty used by a user in one device).

A Domain Registry stores, for each user identifier, the list of Hyperty Instances the user runs on his devices. It also stores, for each Hyperty Instance, the data that enables other applications to contact it, by providing a mapping between the identifier for each Hyperty Instance and the data that characterizes it.

The Domain Registry is a critical service as it stands in the critical path for call establishment. As it will be used very often, it must provide a low access time, high availability and be capable of fast updates (e.g. for when a device changes IP address). It is based on the client-server model and handles high-speed and high-frequency data.

The remaining of this document comprises the following sections: How to deploy, comprising instructions on how to use Docker and how to run the program through the command line, a definition of the REST API and respective available endpoints, and finally, some usage examples.

How to deploy

The Domain Registry can be deployed in one of two ways: using Docker or installing in a Java capable environment. It was only tested in GNU/Linux and Mac environments.

How to deploy using Docker

A Dockerfile is provided, so is possible to run the Domain Registry through a Docker container.

1. Download the code from the official repository in Github.
2. Inside the *server* folder build the Docker image by executing the command: `docker build -t domain-registry ./`
3. Now is possible to run the application inside the container, executing the following command: `docker run -p 4567:4567 domain-registry`

How to run through the command line

In order to being able to run the Domain Registry, one must have Apache Maven version 3[1] and Java SDK 8[2] installed. The first is a build automation tool for Java projects, while the second is a development environment for building applications using the Java programming language. Git[3], a source code management system, may also be needed if the user chooses to clone the Github repository. Using an environment configured with these tools, installation follows these steps:

1. Download the code straight from the respective Github repository[4]. Here, two options are available: clone the repository using a Git client (e.g. through the command line or with the Github desktop application), or download a zip file from the Github repository web page.
2. Inside the *server* folder execute the following command to build the application: `mvn clean compile`
3. Launch the application with the command: `mvn exec:java`
4. Once the server is successfully running requests can be issued by the user to the REST API.

Rest API definition and available endpoints

The Domain Registry is a REST server that allows to create, update and remove data (users and Hyperty Instances in this case). Next, are described the three available API endpoints.

1 https://maven.apache.org

2 http://www.oracle.com

3 https://git-scm.com

4 https://github.com/reTHINK-project/dev-registry-domain

- GET /hyperty/user/:user_id
- PUT /hyperty/user/:user_id/:hyperty_instance_id
- DELETE /hyperty/user/:user_id/:hyperty_instance_id

Possible HTTP status codes returned: 200 OK indicating that the request has succeeded and 404 Not Found indicating that the server has not found anything matching the request URI (users or hyperties). In both cases, a message is returned on the response: “hyperty created”, “user not found” or “data not found”.

Since the users and hyperties URLs contain characters outside the ASCII set, URLs need to be converted to a valid ASCII format. The character “%” followed by two hexadecimal digits replace the unsafe characters. As an example, the enconded version of user://inesc-id.pt/ruijose is user%3A%2F%2Finesc-id.pt%2Fruijose.

GET /hyperty/user/:user_id

Retrieves all Hyperties instances from a user indicated by the **user_id** parameter.

Parameters

user_id	The ID of the user for whom to return results for. **Example value**: user://inesc-id.pt/ruijose

Example request

GET /hyperty/user/user%3A%2F%2Finesc-id.pt%2Fruijose

Example result


```
{
  "hyperty://inesc-id.pt/b7b3rs4-3245-42gn-4327-238jhdq83d8": {
    "descriptor": "hyperty-catalogue://localhost/HelloHyperty",
    "startingTime": "2016-02-08T13:40:26Z",
    "lastModified": "2016-02-08T13:41:27Z
  "hyperty://inesc-id.pt/b7b3rs4-3245-42gn-4127-238jhdq83d8": {
    "descriptor": "hyperty-catalogue://localhost/HelloHyperty",
    "startingTime": "2016-02-08T13:42:00Z",
    "lastModified": "2016-02-08T13:42:53Z"
  }
}
```


The *descriptor* is a link to the Catalogue from where the descriptor of the instance can be retrieved and, the *startingTime* and *lastModified* refer to the date the instance is first registered and the last time the instance was modified. Both dates format are compliant with ISO8601 format. In the future, more information about the hyperty instances will be provided.

In this example, two hyperty instances are returned for the requested user id. Note that the requested URLs are encoded.

If the server could not find what was requested, along with the HTTP status codes, a “user not found” or a “data not found” message is returned to the user.

Examples


```
{
  “message” : “user not found”
}

{
  “message” : “data not found”
}
```

PUT /hyperty/user/:user_id/:hyperty_instance_id

Creates or updates a Hyperty Instance. It also creates a user if it doesn't exists already.

Parameters

user_id	The ID of the user for whom to return results for. **Example value**: user://inesc-id.pt/ruijose
hyperty_instance_id	The ID of the Hyperty to be created. **Example value**: hyperty://ua.pt/428bee1b-887a8ee8cb32

Example request

PUT /hyperty/user/user%3A%2F%2Finesc-id.pt%2Fruijose/hyperty%3A%2F%2Fua. pt %2F428bee1b-887a8ee8cb32

Example result

```
{
  "message" : "hyperty created"
}
```

Note that the requested URL's are encoded.

DELETE /hyperty/user/:user_id/:hyperty_instance_id

Deletes a Hyperty Instance from a user indicated by the **user_id** parameter.

Parameters

user_id	The ID of the user for whom to return results for. **Example value**: user://inesc-id.pt/ruijose
hyperty_instance_id	The ID of the Hyperty to be created. **Example value**: hyperty://ua.pt/428bee1b-887a8ee8cb32

Example request

DELETE /hyperty/user/user%3A%2F%2Finesc-id.pt%2Fruijose/hyperty %3A %2F%2Fua.pt%2F428bee1b-887a8ee8cb32

Example result

```
{
  “message” : “hyperty deleted”
}
```

Note that the requested URL’s are encoded.

Future functionalities

The current version is missing any *authentication mechanisms*. Currently, it is assumed that the Message Node is the only one capable of interacting with the Local Registry, with the former being trusted by the latter to verify the user’s authorization to perform the requests. This model will have to be replaced with a secure mechanism where either the identity of the Message Node or of the user is verified.

Another missing feature is persistent storage. The current version stores data in-memory. In the near future, a NoSQL database will backend the server. The

database that will be used is Cassandra DB, primarily for providing both a masterless cluster with no single point of failures and, fast reads and extremely fast writes. Also, Cassandra uses replication to achieve high availability and durability. Each data item is replicated at N machines, where N is a pre-configured replication factor.

A load balancer will also be added to distribute network traffic across the Domain Registry servers. Thereby, we hope to increase capacity (concurrent users) and application's reliability.