Bachelor´s Thesis

Information Technology

Internet Technology

2015

Fernando Somoza Alonso

DEVELOPMENT OF A RESTFUL API

– HATEOAS & DRIVEN API

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Internet Technology

2015

Tiina Ferm

Fernando Somoza Alonso

DEVELOPMENT OF A RESTFUL API

With the imminent future of the Internet of the Things which implies that everyday objects are connected between them and to the Internet, the need for the development of a software capable of managing communication devices arises.

The goal of this thesis was to create a prototype of an Application Programming Interface in the Java language and implement a REST architectural style, capable of managing and authenticating different kinds of information, such as devices and users, as well as to allow the devices to import, export, store, and post-process relevant data. For this purpose, the thesis is divided in two parts: the theoretical foundation and the practical implementation.

The theoretical foundation examines the difference in terms of software architecture and software architectural style in order to introduce REST, both its elements and constraints.

The implementation of the prototype shows how the development was implemented as well as some samples of its functionality.

After the implementation, the results are presented and assessed. Finally, recommendations for upgrading the prototype are proposed.

KEYWORDS:

REST, API, HTTP, JAVA, Internet Of Things

CONTENT

APPENDICES

FIGURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

DNS	Domain Name System
CLI	Command Line Interface
DAO	Data Acces Object
GUI	Graphical User Interface
HATEOAS	Hypertext As The Engine Of Application State
HTTP	HyperText Transfer Protocol
IoT	Internet of Things
JDK	Java Development Kit
JSON	JavaScript Object Notation
JRE	Java Runtime Environment
POJO	Plain Old Java Object
REST	REpresentational State Transfer
RESTful	That conforms the REST constraints

1 INTRODUCTION

We live a world fully linked with the technology. Almost everybody is member of a social media like Facebook, Twitter or LinkedIn and has a public profile on the internet. In 2009, with the boom of the smartphones, the way of communication of the society underwent a significant change. The first wearable devices have started to gain popularity, from the smartbands to the smart jewellery, without forgetting watches and glasses, everyday devices have started to be interconnected and connected to the internet. A large majority of companies and experts in technological marketing agree on the fact that in a window of five years, billions of devices will be connected in the Internet of Things (IoT) (ABI Research 2013; Anderson et al. 2014).

The Internet of Things is a concept that refers to the digital interconnection of any everyday object with internet. These objects acquire more value due to their ability to send and receive data with the user, the manufacturer and other connected devices. With the aim of being able to connect, the devices require a software component that makes possible its communication and management. The ideal software for this function is an interface that can be implemented in any device, i.e. it is multiplatform, and that allow a fast, simple and efficient communication.

The motivation for this thesis is to study and develop a RESTful API that could satisfy those requirements.

The goal of this thesis is to develop and analyse an API using the REST architectural style. This API will be basic for future development of different multi-platform applications for the Internet of the Things. The API should be able to manage and authenticate different kind of information like devices and users, as well as to allow the devices to import, export, storage and post-process relevant data. There is no need to include a GUI (Graphical User Interface) because the API should be functional from a CLI (Command Line Interface).

In order to accomplish the goal of this thesis, a theoretical foundation is studied and reviewed in the next chapter of the thesis with the aim of developing later on a usable API. In chapter 3, the process of the development is reviewed with some samples of the code and comments about the design. Finally, the last chapter describes the conclusion with the result of the development.

2 REST

The term REST stands for Representational State Transfer and was defined by Roy Thomas Fielding in his PhD dissertation: *”Architectural Styles and the Design of Network-based Software Architectures.”* published in 2000. REST is not a software architecture itself, but “a coordinated set of architectural constraints which attempts to minimize latency and network communication, while maximizing the independence and scalability of component implementations” (Fielding and Taylor 2002).

This chapter presents the theory that supports the goal of this thesis. First of all the terms of software architecture and software architectural style are defined as they are necessary to a better understanding of REST later on. Then, the study moves into REST: introduction in how Fielding approaches REST as an architectural style, description of the specific constraints and elements that compose REST, the importance of HATEOAS, the abstractions of a RESTful system, and finally, an explanation of RESTful systems using HTTP.

2.1 Software architecture

$$\begin{gathered}\text{Software Architecture} = \\ \{\ \text{Elements, Form, Rationale}\ \}\end{gathered}$$

Figure 1: Software architecture model (Perry and Wolf 1992)

One of the problems encountered in the past when talking about software architecture is that the term has been used widely and inconsistently by different authors in different situations. Fielding made his own definition based on previous researches like the “*Foundations for the study of software architecture*” paper from D. E. Perry and A. L. Wolf (**Error! Reference source not found.**). As Fielding asserts in his dissertation:

"A software architecture is defined by a configuration of architectural elements—components, connectors, and data—constrained in their relationships in order to achieve a desired set of architectural properties." (Fielding 2000)

Nowadays, if a definition is needed, there is a standardize definition from ISO (International Organization for Standardization) that generalize more the concept. As ISO/IEC/IEEE establishes, architecture referring to systems or software is defined as the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution (ISO/IEC/IEEE 42010:2011).

According to the previous definitions, it can be stated that a software architecture is a high-level abstraction of a system that instead of focusing on the details of the elements, designs how the elements are used, how they are used by other elements and how they interact among them. Therefore, the structure of a software architecture can be separated in the following terms according to their visible elements in components, connectors and properties, the latter two being those forming the relationships between components.

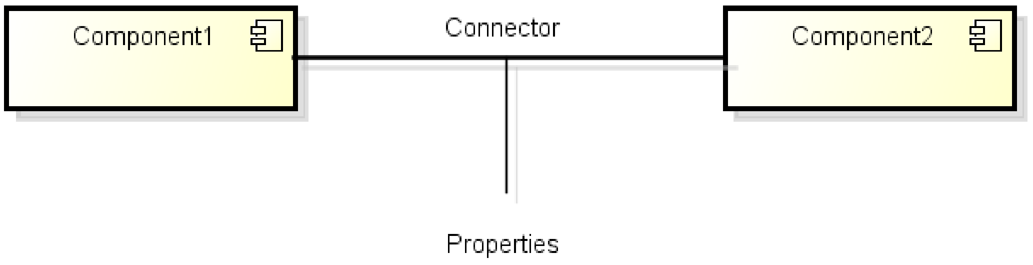


Figure 2: Elements of an architecture

Components are the computational elements which collectively constitute an architecture when accompanied by the description of their interactions (Garlan and Shaw 1994). Those computational elements are abstracts units of software instructions and internal states that provide a transformation of data via its interface (Fielding 2000). If we imagine an architecture as a graph (Figure 2), the nodes represent the components and arcs represent the relationship between nodes (connectors). Some examples of components in an architecture implementation could be clients, servers or databases.

The connectors can be described as the glue that keeps all the architecture components together (Perry and Wolf 1992). In Fielding's dissertation there is a more accurate description defining a connector as "an abstract mechanism that mediates communication, coordination, or cooperation among components" (Shaw and Clements 1997). It has to be noted on both components and connectors definitions that authors emphasize on the abstract nature of the software architecture. Any implementation of the architecture has the details of its components and connectors hidden at the architectural level. Examples of connectors can be any communication protocols as client-server protocol, pipes or procedure calls.

Data elements are those that contain the information that is used and transformed (Perry and Wolf 1992). "A datum is an element of information that is transferred from a component, or received by a component, via a connector" (Fielding 2000).

Properties can be defined as constraints that provide conditions and restrictions for component and components relationships. One of the first definitions of properties as part of a software architecture was a composition between *properties,* which are used to define constraints on the elements and *relationships*, which are used to constrain how the different elements may interact and how they are organized with respect to each other in the architecture (Perry and Wolf 1992). To summarize, properties are additional information about the elements and their associated relations (Clements et al. 2003). The Properties are induced by the set of constraints within an architecture (Fielding 2000). This statement leads to the definition of software architectural style.

2.2 Software architectural style

An architectural style defines a set of rules that describe the way in which component interacts. It is also defined as a specialization of element and relation types, together with a set of constraints on how they can be used (Clements et al. 2003). Although the definition closely resembles the software architecture, note the term specialization of a style versus the high-level abstraction or generalization of the architecture. Therefore, a style may be thought as a set of constraints on an architecture and as an abstraction for a set of related architectures.

Sometimes there is no hard dividing line between where architectural styles ceases and architecture begins. The important thing about an architectural style is that it encloses the important decisions concerning the architectural elements and emphasizes important constraints on the elements and its relationships (Perry and Wolf 1992).

2.3 REST approach

Roy T. Fielding introduced the term REST in his PhD dissertation. That is why "Architectural Styles and the Design of Network-based Software Architectures" is considered the "bible" of REST. Hence almost all the references in this section and the next one come from it.

REST is an architectural style for distributed hypermedia systems. It is a hybrid style inferred from various network-based architectural styles and combined with extra constraints in order to define a uniform connector interface. It focuses on the constraints that must be placed on the connectors semantics whereas other styles focus on the constraints on the components semantics.

In order to derive REST, Fielding, instead of starting from scratch, starts identifying the system needs, without any constraints, and then incrementally starts applying constraints to the elements of the system. This way of designing emphasizes moderation and proper understanding of the system context. Thus, REST is derived from the null style, which is an empty set of constraints so there are no relationships between elements, and then other constraints are added ensuring harmony among them.

The extra constraints of REST come from the next architectural styles: Client-Server, Client-Stateless-Server, Client-Cache-Stateless-Server, Layered System and Code On Demand. REST also includes the concepts of resources and uniform interface.

2.4 Constraints

2.4.1 Client-Server

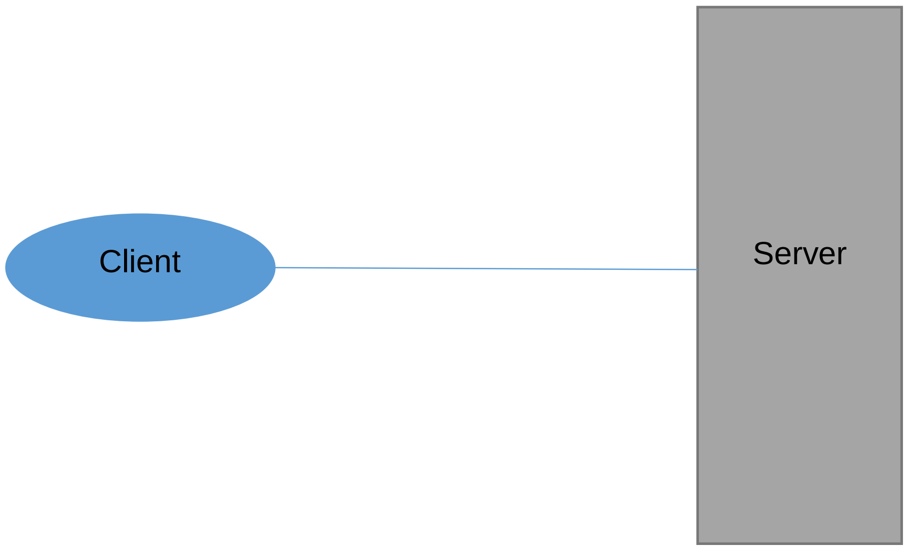


Figure 3: Client-Server

The Client-Server interface requires the existence of a client component that sends requests and a server component that receives requests and may issue a response (Figure 3). This constraint is based on the principle of separation of concerns. A uniform interface separates clients and servers interfaces. This separation of concerns means that clients for example are not related to data-storage that is a server concern, and servers are not related to user interface or user state that are client concerns. This improves portability of interfaces across multiple platforms and scalability by simplifying the server components. Also it supports the independent evolution of the client-side logic and server-side logic as each component can be substituted and developed separately as long as the interface among them does not change.

Client-Server is perhaps the most foundational constraint as all of the other constraints reference its artifacts and so build upon this constraint.

2.4.2 Stateless

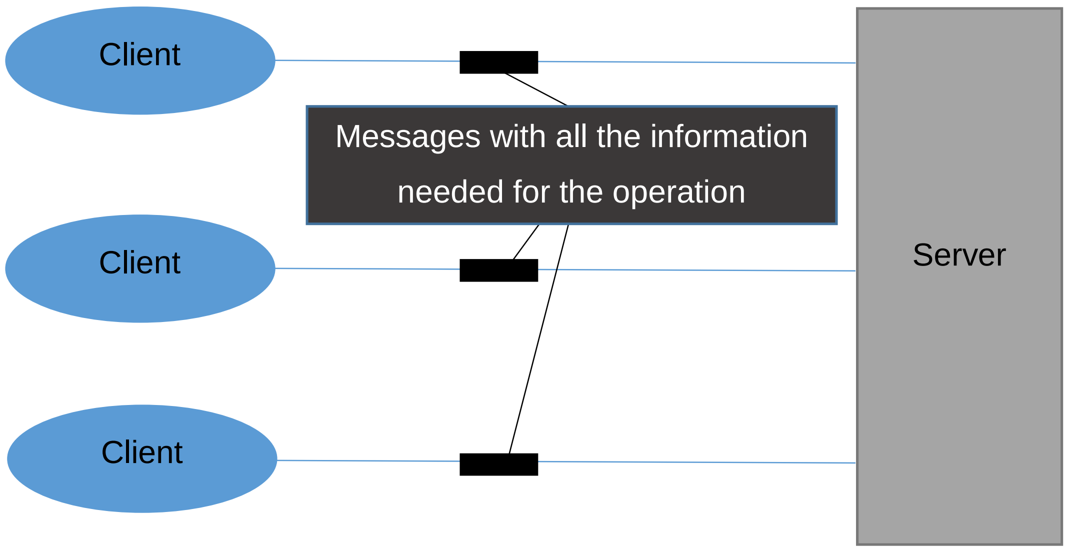


Figure 4: Client-Stateless-Server

Statelessness constraint is added to the Client-Server constraint. In each interaction, the communication between client and server has to be stateless. This means that each request from any client should contain all the information necessary in order to make the server understand the meaning of the request (Figure 4). Then, all the data concerning the session state should be returned to the client. Hence, session state is kept entirely on the client and the server cannot reuse information from previous requests.

This constraint adds some very advantageous properties but also carries some disadvantages. In one hand, stateless adds visibility, reliability, and scalability. Visibility is improved because there is no need of a monitoring system to trace back previous requests in order to determine the full nature of the request as the request contains all the information. Reliability is improved because the recovery from partial failures is much easier. Scalability is better because not having to store the state between requests allows the server component to free resources rapidly, and also simplifies implementation because the server does not have to manage resource usage across requests.

In the other hand, the network performance is reduced due to the need of several requests since the server cannot store shared context data, and server control over application consistency is reduced because any session state is allocated on the client side.

2.4.3 Cacheable

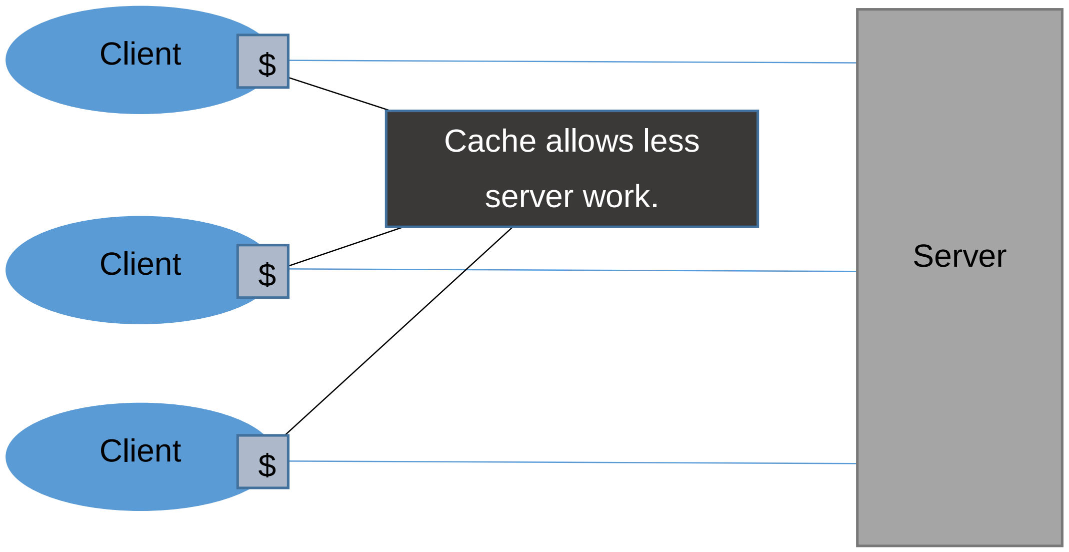


Figure 5: Client-Cache-Stateless-Server

In order to mitigate the reduction of network performance due to the stateless constraint, another constraint is added on top of the client-stateless-server style shown before. REST includes cache constraint so that subsequent requests to the server do not have to be made if the required data is already in a local cache on the client side. So, the client-cache-stateless-server is formed (Figure 5).

This constraint requires the data within a response to a request to be labeled implicitly or explicitly as cacheable or non-cacheable. Requests are passed through a cache component, which may reuse previous responses to partially or completely eliminate some interactions over the network. Adding cache constraints have some advantages but also a trade-off. On the one hand, cache constraints have the potential to eliminate some interactions partially or completely, reducing the average latency of a series of interactions thus improving efficiency, scalability, and user perceived performance. On the other hand, reliability might be compromised if the data within the cache differs from the data that would have been obtained from the server.

2.4.4 Uniform Interface

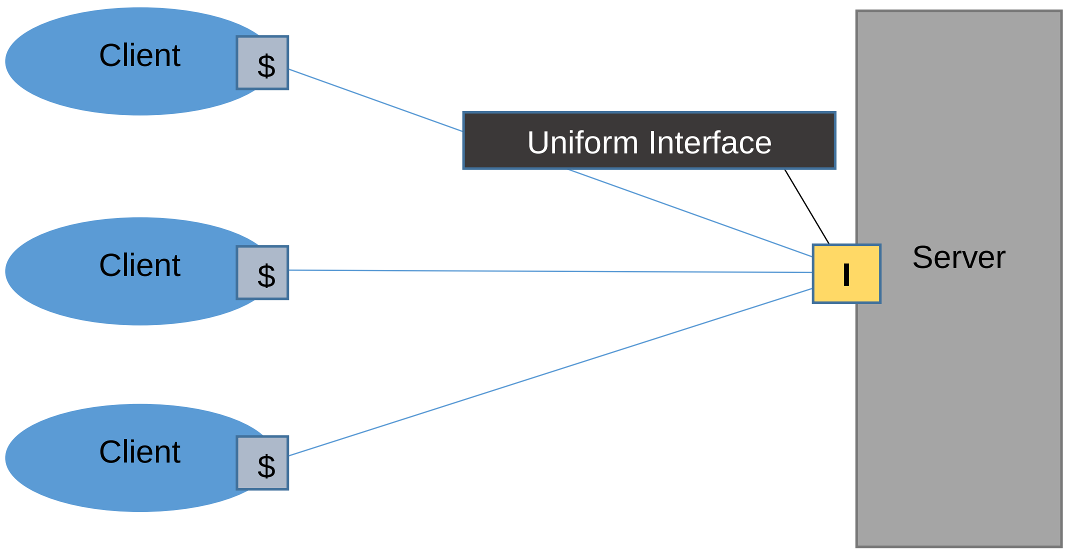


Figure 6: Uniform-Client-Cache-Stateless-Server

Uniform interface constraint states that all the components (clients and servers) within a REST architecture must share a single, prevailing interface (Figure 6). This means that the interface for a component needs to as be generic as possible (usually HTTP). It simplifies and decouples the architecture, which enables each part of the architecture to evolve independently. This emphasis on a uniform interface between components is what distinguished REST from other network-based styles.

By applying this constraint, the overall system architecture is simplified and the visibility of interactions is improved. Also, as seen before, it encourages independent evolvability of components. However, a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs.

To obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components so additionally, uniform interface has four sub-constraints: Identification of resources, manipulation of resources through representations, self-descriptive messages and Hypermedia as the engine of application state (HATEOAS).

2.4.5 Layered System

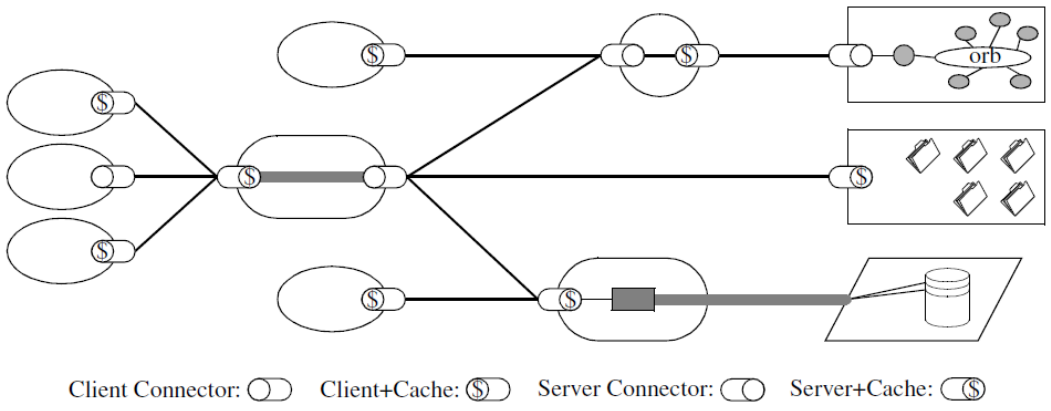


Figure 7: Uniform-Layered-Client-Cache-Stateless-Server (Fielding 2000)

Layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component's knowledge of the system is limited to the immediate layer they interact with (Figure 7).

By having a layered system, it ensures that services are able to communicate only with intermediary layers and other layers are invisible for them, making possible to improve security. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.

The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance. This disadvantage can be relieved with the use of shared caches that acts like the cache constraint but between layers.

2.4.6 Code on Demand

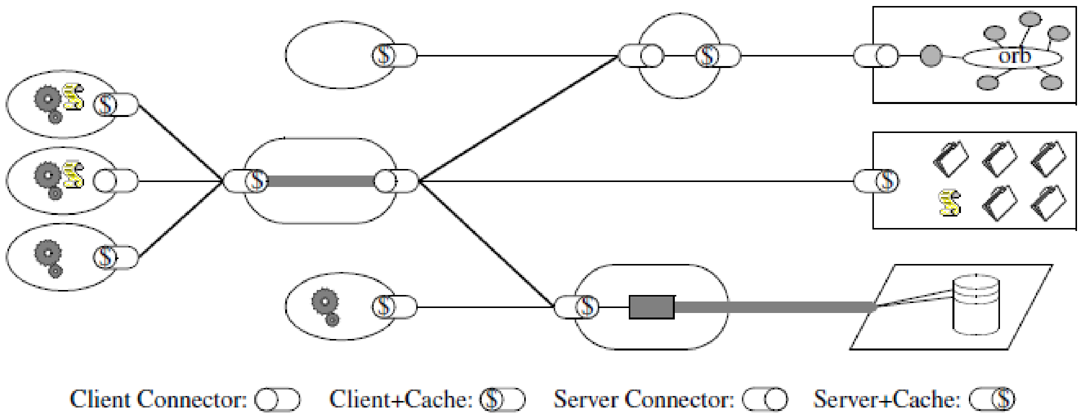


Figure 8: REST style (Fielding 2000)

The last constraint of the REST set is Code on Demand which is an “optional” constraint that allows the clients to download and execute code from a server (Figure 8). Client functionality may be extended with applets or scripts.

It is called optional because it has some advantages and disadvantages depending on the context of the implementation and as it depends on the context, it is not always possible to implement. The advantages are that it simplifies clients, hence promotes the reduced coupling of features, and it improves scalability by virtue of the server off-loading work onto the clients. However, code on demand reduces visibility generated by the code itself, which is hard for an intermediary to interpret.

2.5 REST elements

The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

Each software architecture is composed by components, connectors and data. “REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements” (Fielding 2000).

2.5.1 Components

The role of REST components is to establish communication. They are classified by their function in an overall application action as shown in Table 1.

Table 1: REST Components and examples (Fielding 2000)

Component Modern Web Examples	**Component Modern Web Examples**
Origin server	Apache httpd, Microsoft IIS
Gateway	Squid, CGI, Reverse Proxy
Proxy	CERN Proxy, Netscape Proxy, Gauntlet
User agent	Netscape Navigator, Lynx, MOMspider

- A user agent uses a client connector to initiate a request and becomes the ultimate recipient of the response.
- An origin server uses a server connector to govern the namespace for a requested resource.

- A proxy is an intermediary **selected by** a client to provide interface encapsulation of other services, data translation, performance enhancement, or security protection.
- A gateway is an intermediary **imposed** by the network or origin server to provide an interface encapsulation of other services, for data translation, performance enhancement, or security enforcement.

2.5.2 Connectors

The connectors present an abstract interface for component communication, enhancing simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and communication mechanisms (Fielding 2000). The different connector types are summarized in Table 2.

Table 2: REST Connectors and examples (Fielding 2000)

Connector Modern Web Examples	**Connector Modern Web Examples**
Client	libwww, libwww-perl
Server	libwww, Apache API, NSAPI
Cache	browser cache, Akamai cache network
Resolver	bind (DNS lookup library)
Tunnel	SOCKS, SSL after HTTP CONNECT

- Client and servers are the primary connector types. The different between them is that a client initiates the communication by making a request, whereas the server is constantly listening for connections and responds requests for supplying access to its services.
- Cache can be allocated on the interface of clients or server to provide save cacheable responses to current interactions in order to be reused for later requests. Its main functionality is to reduce interaction latency.
- A resolver translates partial or complete resource identifiers into the network address information needed to establish an inter-component

connection. The DNS is the most well-known and easy to understand. The use of a resolver adds request latency but can improve the longevity of the resources references.
- The tunnel simply relays communication across a connection boundary, such as a firewall or lower-level network gateway.

2.5.3 Data Elements

REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types selected dynamically based on the capabilities or desires of the recipient and the nature of the resource (Fielding 2000). Thus data elements can be summarized as follows: resources, resources identifiers, representations, representations and resource metadata and control data (Table 3).

Table 3: REST Data Elements and examples (Fielding 2000)

Data Element Modern Web Examples	Data Element Modern Web Examples
resource	The intended conceptual target of a hypertext reference.
resource identifier	URL, URN
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
Resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

- Resources: The key abstraction of information in REST is a resource. It is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time. Those are representations.
- Resources identifiers: A uniform resource identifier (URI) is a string of characters used to identify a name of a resource. Such identification enables interaction with representations of the resource over a network (RFC 3986.2005).

- Representations: The representation of resources is what is sent back and forth between components. It is a temporal state of the actual resource. In general terms, it's a binary stream together with its metadata that describes how it has to be consumed.
- Representation metadata: It is a set of name-value pairs, where the name corresponds to a standard that defines the value’s structure and semantics that describes the representation.
- Resource metadata: information about the resource that is not specific to the supplied representation.
- Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response.

Once understood the REST elements, the sub-constraints necessary to achieve a uniform interface can be defined.

- Identification of resources: each resource should have its own unique URI.
- Manipulation of resources through representations: Through a URI, an instance of the resource can be requested. The response can be returned in various formats each of them are representations of the identified resource.
- Self-descriptive messages: Each message (client request and server response) contains all the information necessary to complete the task.
- Hypermedia as the engine of application state: Sharing representations by sending self-descriptive messages to identified resources changes the state of the application.

2.6 HATEOAS

HATEOAS, Hypermedia As The Engine Of Application Style means that hypertext should be used to navigate and find the way through the implementation. REST has to be stateless but at the same time, REST means

Representational state transfer. HATEOAS solves this contradiction. For example in a REST API implementation: The API should be entered with no prior knowledge beyond the initial URI. From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user’s manipulation of those representations. In other words, each representation of a resource should include references (links) that describe the transition to the next state. In each response message, the link for the next request message should be included.

This constraint also allows REST APIs to be self-describing because a single representation has data describing the resource, actions that it can be done by the client to the resource and links to a possible next state. Fielding published on his blog entry on 2008 that “a REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state” (Fielding 2008). In this publication, there are also some assertions of why a RESP API should be hypertext-driven (HATEOAS) as the ones described before.

2.7 REST API

The acronym API comes from Application Programming Interface. An API is a set of functions and procedures that fulfill one or many tasks for the purpose of being used by other software. It allows to implement the functions and procedures that conform the API in another program without the need of programming them back.

As RESTful systems usually communicate with the Hypertext Transfer Protocol (HTTP), a REST API is a library based completely on the HTTP standard. It is used to add functionality to a software somebody already owns safely. The functionality of an API is usually limited by the developer so no more functionality can be added.

2.8 REST through HTTP

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems (RFC 7230.2014).

In a RESTful system, clients and servers negotiate the representations of resources via HTTP. RESTful systems apply the four basic functions of persistent storage, CRUD (Create, Read, Update, Delete), to a set of resources. In terms of the HTTP standard, those actions can be translated to the HTTP methods (also known as verbs): POST, GET, PUT, and DELETE (Table 4). Other HTTP methods that are also used but not as often as the formers are OPTIONS, HEAD, TRACE, PATCH and CONNECT.

Table 4: CRUD and HTTP equivalence

CRUD actions	**HTTP method equivalence**
Create	POST
Read	GET
Update	PUT
Delete	DELETE

The HTTP verbs comprise a major portion of the Uniform Interface constraint because it provides action based on resources instead of verbs. It is possible to make a correlation between the CRUD actions of a system and the equivalent HTTP methods as follows:

- GET: The method GET is used to retrieve/read a representation of a resource and should only be used for that purpose although it is also possible to update the state of data in the server. When used only for reading is considered safe.
- POST: The POST verb is most-often utilized for creation of new resources. It is an unsafe method because it can modify data states on the server
- PUT: It is used to update one existing resource on the server with the information contained on the request. However, PUT can also be used to create a new resource if the data does not already exist in the server. It is not a safe method because it modifies or create data.
- DELETE: As the name suggest, DELETE is used to delete a resource by providing its ID.

A classification of the verbs is possible depending on if they are safe or unsafe, and if they are idempotent or not.

Safe methods never modify resources. From the previous ones, only GET is safe because the others may result in a modification of the resources.

Idempotent methods achieve the same result regardless of how many times the request is repeated. When used them correctly, GET, PUT and DELETE are idempotent. Repeating a PUT method with the same body content should modify a resource with the same data, so it remains unchanged. This is similar with DELETE where you can only remove a resource once.

Apart from HTTP methods, RESTful services also use HTTP headers to specify the representation metadata, for example the content type of the body that it can be used to choose between different representations of the same resource.

3 DEVELOPMENT

The first part of the thesis discusses REST as an architectural style. This chapter discusses how to implement a RESTful API using Java with Spring framework.

First, the necessary tools used for the implementation of the API are explained. Later on, there is an explanation of JSON, the standard data format the API manages. Finally, the implementation and testing are explained.

3.1 Programming Environment

The programming language chosen for the development of the API is Java. Java is a high-level programming language so it enables to write programs for any computer, which are easier to understand for humans than other assembly languages which are very close to machine language. Java is intended to “Write once, run everywhere”. This means that after the compilation, a Java program could be executed in any other platform as long as they support Java (with a Java virtual machine).Thus, Java is one of the most popular languages for developing client-server web applications. For programming in Java, a Java Development Kit (JDK) is required, and it is a development environment for building applications, applets, and components using the Java programming language. It always come with a Java Runtime Environment (JRE) that includes a Java Virtual Machine to run the applications.

For the realization of the API described in this thesis, the version 7 of the Java JDK is used and it can be downloaded from the Oracle official page for free. After the installation, is necessary to set the system PATH variable by including on it the location of the bin directory of the installed Java JDK.

In addition to Java, the API is built with the Spring Framework. The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. In order to use Spring, the only requirement is to have a minimum required JDK installed.

Another tool that is used in this project is Maven. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Maven, among other things, helps the user to build a project through an inner lifecycle that includes validation and compilation. In addition Maven allows to manage the dependencies of a project such as the ones of the Spring Framework in this project. Maven version used is 3.3.3.

Finally, an Integrated Development Environment (IDE) is used for source code editing. The IDE chosen is Eclipse IDE, because it also includes a Java compiler and interpreter. The version of Eclipse IDE used is Luna (v. 4.4.1).

3.2 Other Tools

Additionally to the programming environment, other tools are also needed for the development of the API.

Git is a distributed version control system. Its usage along with gitBucket (an online project hosting) keeps the project in an organized manner and allows to view, revert or commit changes.

cUrl is used for the purpose of making HTTP requests and test the system and its final functionality. It allows to transfer data with URL syntax.

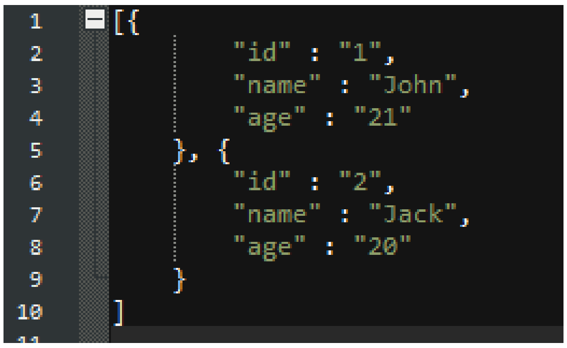
Orchestrate is used for storing data on the cloud. Orchestrate provides a RESTful API with the purpose of storing data without the need of the user to interact directly with a SQL database and its operations.

The project uses other APIs to perform some actions that otherwise would take much more effort and time to implement. Stormpath API serves to manage users and this API uses its functionality to store and manage admins and devices.

3.3 JSON as data format

The communication between components is made through representations of resources. In this API, the format of those representations is JSON.

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It was derived from the ECMAScript Programming Language Standard (ECMA-404 2013; RFC 7159.2014). JSON structure can be defined as a ordered list of objects (array of objects). These objects are a collection of name/value pairs separated by colons (:). An example can be seen in Figure 9.



```
[{
        "id" : "1",
        "name" : "John",
        "age" : "21"
    }, {
        "id" : "2",
        "name" : "Jack",
        "age" : "20"
    }
]
```

Figure 9: JSON example

The utilization of JSON instead of other standard format like XML is due to its simplicity. Both XML and JSON are human readible, but JSON does not need closing tags and is easier to read and is less dense.

3.4 Implementation

The main goal of the API is to create and manage admins and devices by storing them on a database on the cloud. Those devices have to be able to import, export, storage and post-process relevant data. Therefore, the API has to implement three main resources (admins, devices and data) and also the different components that enables to store and manage them. This thesis does not cover the implementation of the security because it has not been made by the author. Also, all the Figures showing any code are only representative due to the API code having more than five thousand lines some of which can be seen in the appendices.

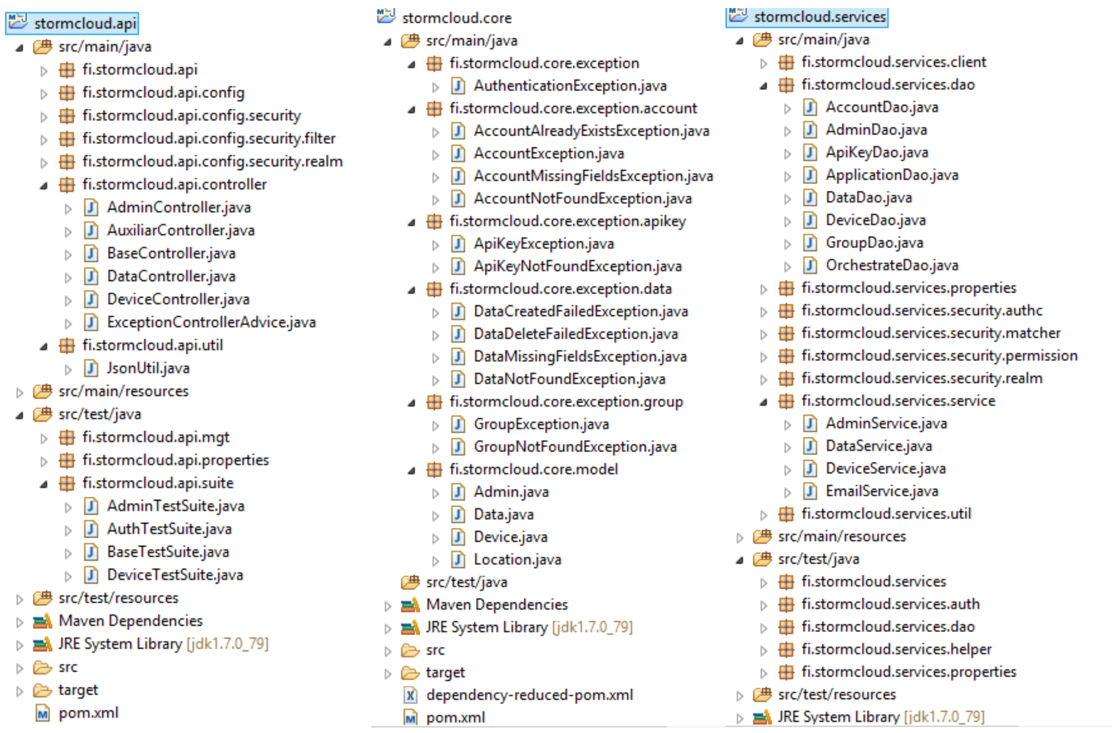
The first step of this process is to set up the different projects that host the API implementation.

3.4.1 Base Project

The whole API is divided into three different Maven projects each of them containing different modules of the API. There are several reasons for the modularization. First of all, the separation on concerns allows to limit the knowledge of the internal classes with different tasks. It also increases the security because it minimizes the potential damage the API could suffer if one of the components fails. Finally, a layered system is easier to maintain and easier to scale.

The skeleton of the API is divided into api, core and services. The project api contains the controller classes of the admins, devices and data. It is the layer which interacts with the client side while doing requests. The project named services, contains classes of the services and DAOs (Data Access Object). This project is where all the logic to connect the databases and the API is programmed. The last project, core, is the one that contains all the resource classes like the POJOs of the admins, devices and controllers, as well as their classes as representations and the definition of exceptions.

Every project has the necessary tests suites in order to test the functionality of each component. The structure of each project with the most representative classes can be seen on Figure 10.

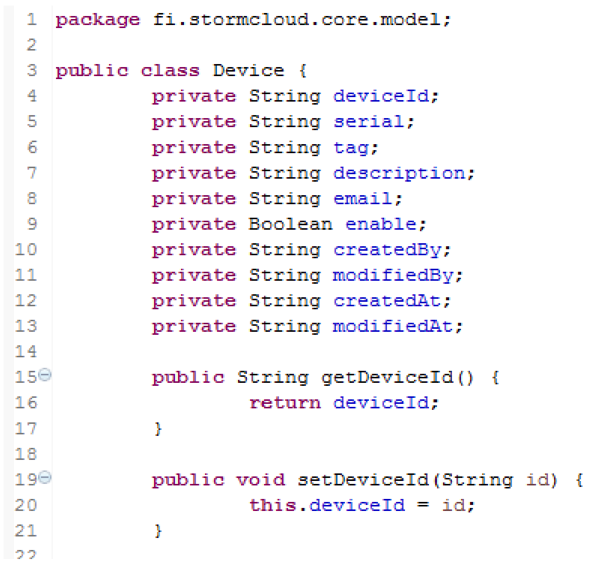


```
stormcloud.api
  src/main/java
    fi.stormcloud.api
    fi.stormcloud.api.config
    fi.stormcloud.api.config.security
    fi.stormcloud.api.config.security.filter
    fi.stormcloud.api.config.security.realm
    fi.stormcloud.api.controller
      AdminController.java
      AuxiliarController.java
      BaseController.java
      DataController.java
      DeviceController.java
      ExceptionControllerAdvice.java
    fi.stormcloud.api.util
      JsonUtil.java
  src/main/resources
  src/test/java
    fi.stormcloud.api.mgt
    fi.stormcloud.api.properties
    fi.stormcloud.api.suite
      AdminTestSuite.java
      AuthTestSuite.java
      BaseTestSuite.java
      DeviceTestSuite.java
  src/test/resources
  Maven Dependencies
  JRE System Library [jdk1.7.0_79]
  src
  target
  pom.xml

stormcloud.core
  src/main/java
    fi.stormcloud.core.exception
      AuthenticationException.java
    fi.stormcloud.core.exception.account
      AccountAlreadyExistsException.java
      AccountException.java
      AccountMissingFieldsException.java
      AccountNotFoundException.java
    fi.stormcloud.core.exception.apikey
      ApiKeyException.java
      ApiKeyNotFoundException.java
    fi.stormcloud.core.exception.data
      DataCreatedFailedException.java
      DataDeleteFailedException.java
      DataMissingFieldsException.java
      DataNotFoundException.java
    fi.stormcloud.core.exception.group
      GroupException.java
      GroupNotFoundException.java
    fi.stormcloud.core.model
      Admin.java
      Data.java
      Device.java
      Location.java
  src/test/java
  Maven Dependencies
  JRE System Library [jdk1.7.0_79]
  src
  target
  dependency-reduced-pom.xml
  pom.xml

stormcloud.services
  src/main/java
    fi.stormcloud.services.client
    fi.stormcloud.services.dao
      AccountDao.java
      AdminDao.java
      ApiKeyDao.java
      ApplicationDao.java
      DataDao.java
      DeviceDao.java
      GroupDao.java
      OrchestrateDao.java
    fi.stormcloud.services.properties
    fi.stormcloud.services.security.authc
    fi.stormcloud.services.security.matcher
    fi.stormcloud.services.security.permission
    fi.stormcloud.services.security.realm
    fi.stormcloud.services.service
      AdminService.java
      DataService.java
      DeviceService.java
      EmailService.java
    fi.stormcloud.services.util
  src/main/resources
  src/test/java
    fi.stormcloud.services
    fi.stormcloud.services.auth
    fi.stormcloud.services.dao
    fi.stormcloud.services.helper
    fi.stormcloud.services.properties
  src/test/resources
  JRE System Library [jdk1.7.0_79]
```

Figure 10: API structure

3.4.2 Core sub-project: Admins, Devices and Data

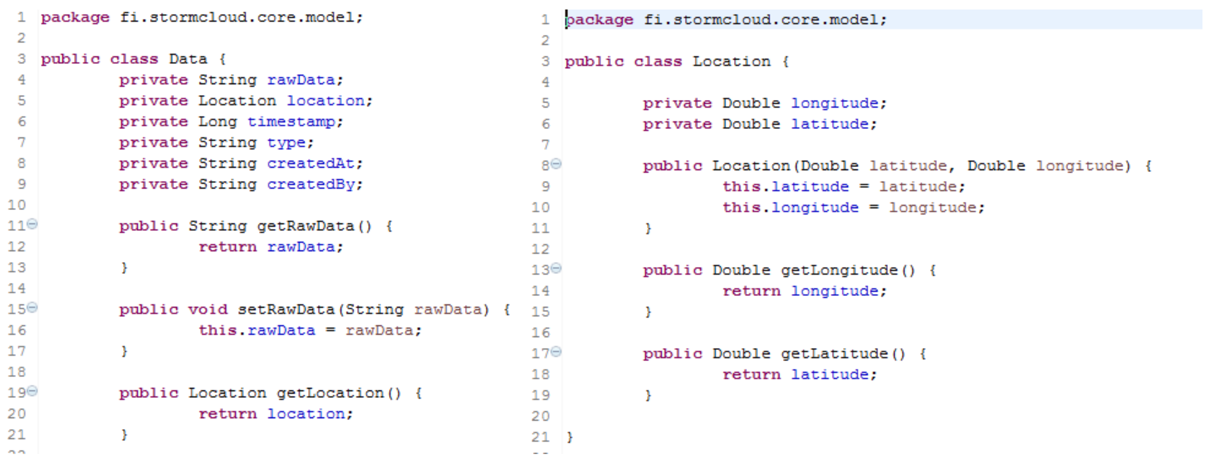
Once the skeleton is created, the next step is to create the entities the API is managing. First, the admins and devices are created. These classes are POJOS, so they do not extend of any interface or abstract classes and they have each attributes, getters and setters. The attribute “xxxId” in addition to the path in which each resource is located is used as URI. The other attributes give information about the resource. The Figure 11 shows an example of (a part of) the device class.



```java
package fi.stormcloud.core.model;

public class Device {
        private String deviceId;
        private String serial;
        private String tag;
        private String description;
        private String email;
        private Boolean enable;
        private String createdBy;
        private String modifiedBy;
        private String createdAt;
        private String modifiedAt;

        public String getDeviceId() {
                return deviceId;
        }

        public void setDeviceId(String id) {
                this.deviceId = id;
        }
```

Figure 11: Device POJO

The resource Data attributes contains from the raw data with all the information a device want to share, to the location and timestamp, which would serve to perform searches with those criteria afterwards. Data class and the its location attribute class can be seen in Figure 12.



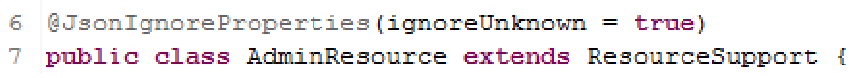
```java
package fi.stormcloud.core.model;

public class Data {
        private String rawData;
        private Location location;
        private Long timestamp;
        private String type;
        private String createdAt;
        private String createdBy;

        public String getRawData() {
                return rawData;
        }

        public void setRawData(String rawData) {
                this.rawData = rawData;
        }

        public Location getLocation() {
                return location;
        }
```

```java
package fi.stormcloud.core.model;

public class Location {

        private Double longitude;
        private Double latitude;

        public Location(Double latitude, Double longitude) {
                this.latitude = latitude;
                this.longitude = longitude;
        }

        public Double getLongitude() {
                return longitude;
        }

        public Double getLatitude() {
                return latitude;
        }

}
```

Figure 12: Data class and Location attribute class.

3.4.3 Api sub-project: Representations, implementing HATEOAS

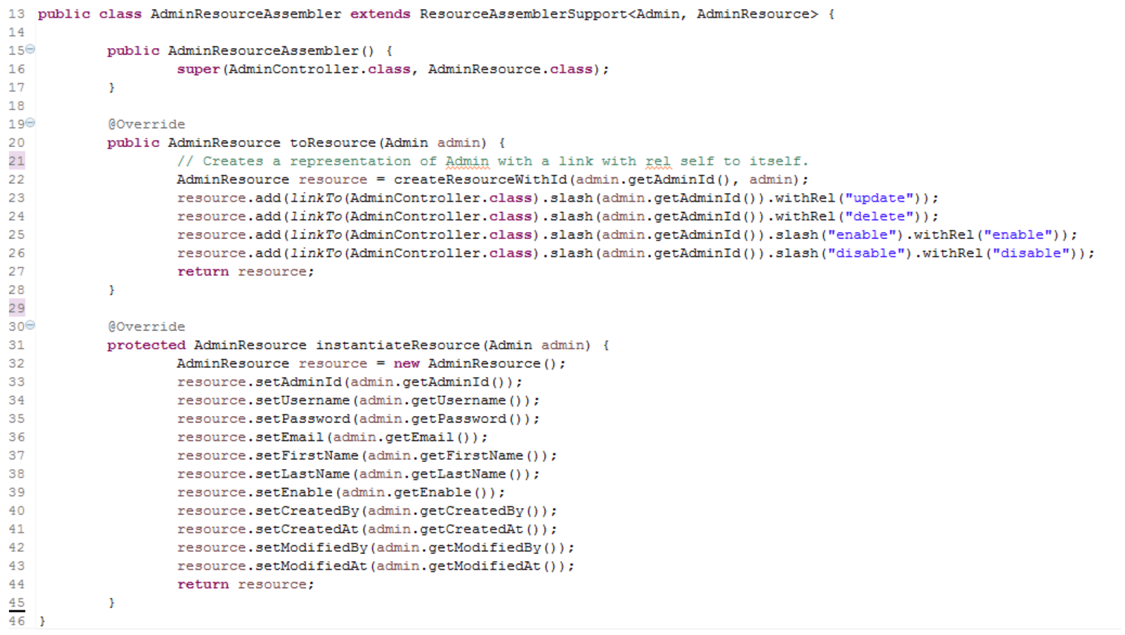
The next step is to create the classes which give support for giving the API the HATEOAS constraint. This step is divided into two smaller. Firstly, a class with each representation admins and devices is created. The classes are almost equal to the ones created before, but now, the classes extend from “ResourceSupport” which is imported from the spring framework library and allow the representations to collect links. Additionally one annotation is included which imported from the “jackson” library. This annotation helps the controller when receiving data by the user(Figure 13).



```java
@JsonIgnoreProperties(ignoreUnknown = true)
public class AdminResource extends ResourceSupport {
```

Figure 13: Jackson annotation and inheritance from ResourceSupport.

The second step is to create classes, which are named XXXResourceAssembler and extend from ResourceAssemblerSupport, that transform the resources into representation while adding hyperlinks pointing to themselves and to the possible usable methods (Figure 14).



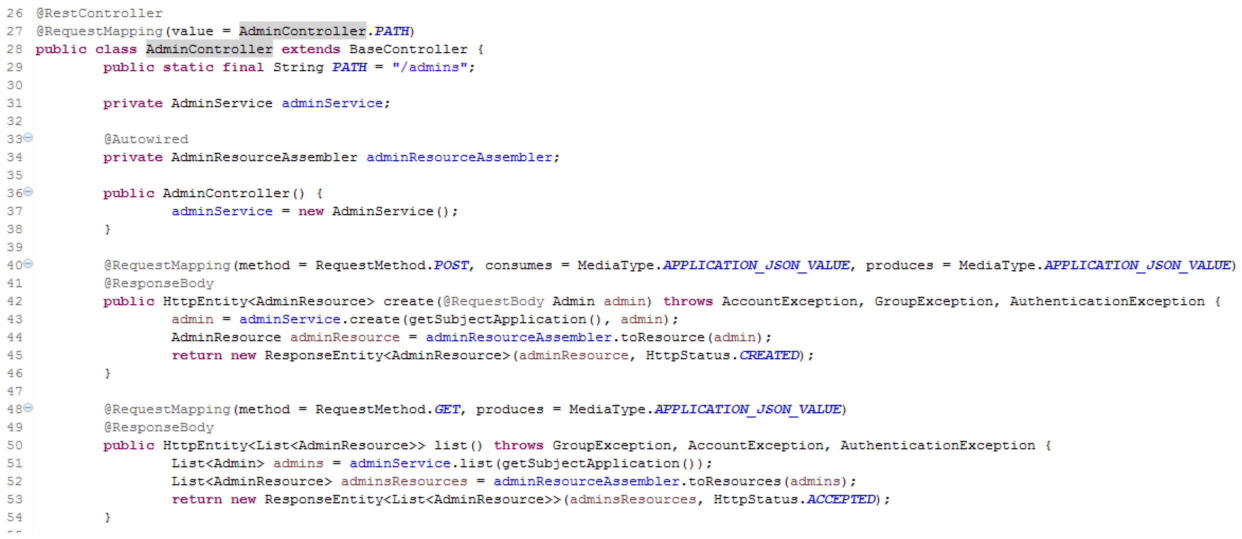
```java
public class AdminResourceAssembler extends ResourceAssemblerSupport<Admin, AdminResource> {

	public AdminResourceAssembler() {
		super(AdminController.class, AdminResource.class);
	}

	@Override
	public AdminResource toResource(Admin admin) {
		// Creates a representation of Admin with a link with rel self to itself.
		AdminResource resource = createResourceWithId(admin.getAdminId(), admin);
		resource.add(linkTo(AdminController.class).slash(admin.getAdminId()).withRel("update"));
		resource.add(linkTo(AdminController.class).slash(admin.getAdminId()).withRel("delete"));
		resource.add(linkTo(AdminController.class).slash(admin.getAdminId()).slash("enable").withRel("enable"));
		resource.add(linkTo(AdminController.class).slash(admin.getAdminId()).slash("disable").withRel("disable"));
		return resource;
	}

	@Override
	protected AdminResource instantiateResource(Admin admin) {
		AdminResource resource = new AdminResource();
		resource.setAdminId(admin.getAdminId());
		resource.setUsername(admin.getUsername());
		resource.setPassword(admin.getPassword());
		resource.setEmail(admin.getEmail());
		resource.setFirstName(admin.getFirstName());
		resource.setLastName(admin.getLastName());
		resource.setEnable(admin.getEnable());
		resource.setCreatedBy(admin.getCreatedBy());
		resource.setCreatedAt(admin.getCreatedAt());
		resource.setModifiedBy(admin.getModifiedBy());
		resource.setModifiedAt(admin.getModifiedAt());
		return resource;
	}
}
```

Figure 14: AdminResourceAssembler showing both methods.

3.4.4 Api sub-project: Controllers

The controller's function is to handle the HTTP requests. Stated differently, they receive the http request from the user and forward the information to the services which are the ones that have the logic to attend the request. Finally, the controllers deliver the response to the user.

In order to program them, the annotations for the API to recognize them as controllers and the ones that map the class to the URI path have to be included. Also, each of them have to include the constructor and the declaration of the service and the assembler class. Lastly, the methods of the controller are programmed. A part of the AdminController with the HTTP methods GET and POST for create and read can be seen in Figure 15.



```java
@RestController
@RequestMapping(value = AdminController.PATH)
public class AdminController extends BaseController {
        public static final String PATH = "/admins";

        private AdminService adminService;

        @Autowired
        private AdminResourceAssembler adminResourceAssembler;

        public AdminController() {
                adminService = new AdminService();
        }

        @RequestMapping(method = RequestMethod.POST, consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
        @ResponseBody
        public HttpEntity<AdminResource> create(@RequestBody Admin admin) throws AccountException, GroupException, AuthenticationException {
                admin = adminService.create(getSubjectApplication(), admin);
                AdminResource adminResource = adminResourceAssembler.toResource(admin);
                return new ResponseEntity<AdminResource>(adminResource, HttpStatus.CREATED);
        }

        @RequestMapping(method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
        @ResponseBody
        public HttpEntity<List<AdminResource>> list() throws GroupException, AccountException, AuthenticationException {
                List<Admin> admins = adminService.list(getSubjectApplication());
                List<AdminResource> adminsResources = adminResourceAssembler.toResources(admins);
                return new ResponseEntity<List<AdminResource>>(adminsResources, HttpStatus.ACCEPTED);
        }
```

Figure 15: AdminController sample

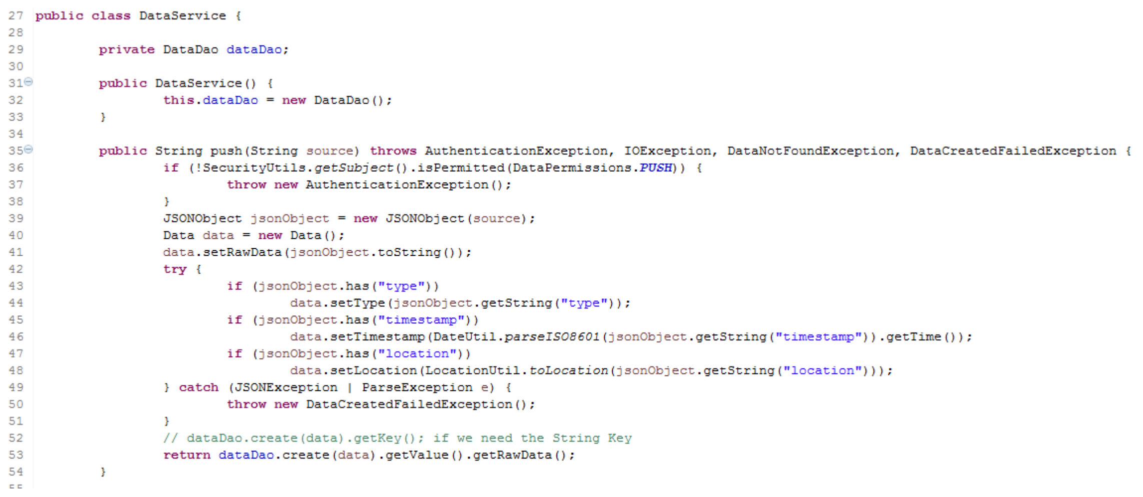
Each of the methods must contain the http verb and path which are associated to, and the type of media they support. Furthermore, the methods have to return a response whose body includes the requested representation and whose headers reflect the status of the operation.

3.4.5 Services sub-project: Services

The services sub-project contains the classes which are more oriented to the server-side. One can distinguish two layers: the DAO classes which interact with the database where the data is located and the services classes. This middle layer is responsible for transforming the controller requests to be understood by the DAO classes and vice versa.

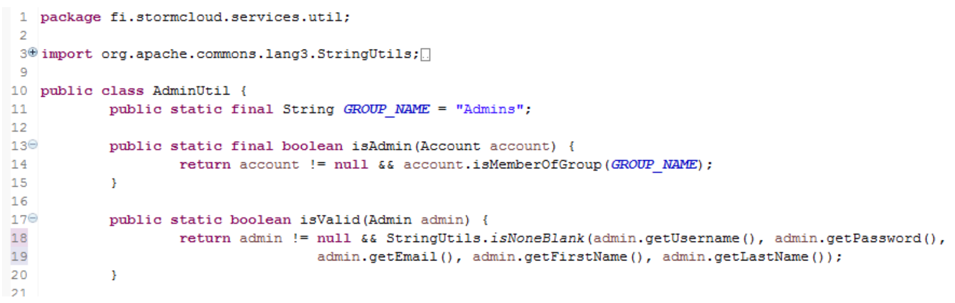
Furthermore, the services are responsible for checking that data from controllers is processable and includes all the required fields, as well as to verify the permissions of the user who is making the requests. If any of the issues listed above occur, a proper exception is thrown and it stops attending the request.

Samples of the DataService class and a Utility class for AdminService with some methods pushing and checking data are shown in Figure 16 and Figure 17 where one can see how the security access and missing field exceptions are handled.



```java
public class DataService {

        private DataDao dataDao;

        public DataService() {
                this.dataDao = new DataDao();
        }

        public String push(String source) throws AuthenticationException, IOException, DataNotFoundException, DataCreatedFailedException {
                if (!SecurityUtils.getSubject().isPermitted(DataPermissions.PUSH)) {
                        throw new AuthenticationException();
                }
                JSONObject jsonObject = new JSONObject(source);
                Data data = new Data();
                data.setRawData(jsonObject.toString());
                try {
                        if (jsonObject.has("type"))
                                data.setType(jsonObject.getString("type"));
                        if (jsonObject.has("timestamp"))
                                data.setTimestamp(DateUtil.parseISO8601(jsonObject.getString("timestamp")).getTime());
                        if (jsonObject.has("location"))
                                data.setLocation(LocationUtil.toLocation(jsonObject.getString("location")));
                } catch (JSONException | ParseException e) {
                        throw new DataCreatedFailedException();
                }
                // dataDao.create(data).getKey(); if we need the String Key
                return dataDao.create(data).getValue().getRawData();
        }
```

Figure 16: DataService sample



```java
package fi.stormcloud.services.util;

import org.apache.commons.lang3.StringUtils;

public class AdminUtil {
        public static final String GROUP_NAME = "Admins";

        public static final boolean isAdmin(Account account) {
                return account != null && account.isMemberOfGroup(GROUP_NAME);
        }

        public static boolean isValid(Admin admin) {
                return admin != null && StringUtils.isNoneBlank(admin.getUsername(), admin.getPassword(),
                                admin.getEmail(), admin.getFirstName(), admin.getLastName());
        }
```

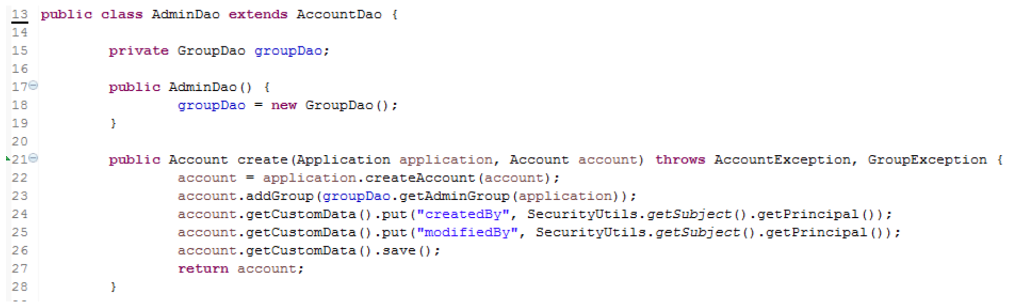
Figure 17: AdminUtil sample

3.4.6 Services sub-Project: DAOs

DAO classes provide access to an underlying database or any other persistence storage. They are in charge of having the correct syntaxes to communicate with the storage unit.

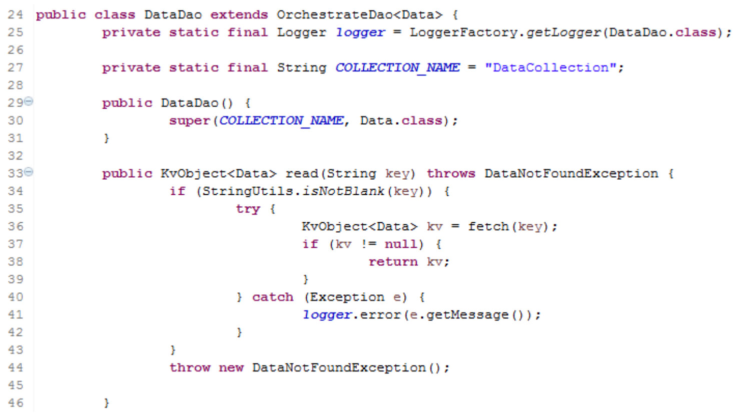
This project interacts with two different storages (Stormpath and Orchestrate) and both of them need to create a client beforehand. Also, both use a set of api key and api secret for authentication. Once both clients are instantiated, the way of communication is different on each of them.

The one for admins and devices needs to handle all the requests via an account which once created, is the responsible of storing and managing admins and servers in the shape of different groups. The persistent storage where the data is stored uses a collection method to store the data and its own objects class called KVObject that extends from the serializable class. Both samples can be seen in Figure 18 and Figure 19.



```java
public class AdminDao extends AccountDao {

        private GroupDao groupDao;

        public AdminDao() {
                groupDao = new GroupDao();
        }

        public Account create(Application application, Account account) throws AccountException, GroupException {
                account = application.createAccount(account);
                account.addGroup(groupDao.getAdminGroup(application));
                account.getCustomData().put("createdBy", SecurityUtils.getSubject().getPrincipal());
                account.getCustomData().put("modifiedBy", SecurityUtils.getSubject().getPrincipal());
                account.getCustomData().save();
                return account;
        }
```

Figure 18: AdminDao sample



```java
public class DataDao extends OrchestrateDao<Data> {
        private static final Logger logger = LoggerFactory.getLogger(DataDao.class);

        private static final String COLLECTION_NAME = "DataCollection";

        public DataDao() {
                super(COLLECTION_NAME, Data.class);
        }

        public KvObject<Data> read(String key) throws DataNotFoundException {
                if (StringUtils.isNotBlank(key)) {
                        try {
                                KvObject<Data> kv = fetch(key);
                                if (kv != null) {
                                        return kv;
                                }
                        } catch (Exception e) {
                                logger.error(e.getMessage());
                        }
                }
                throw new DataNotFoundException();

        }
```

Figure 19: DataDao sample

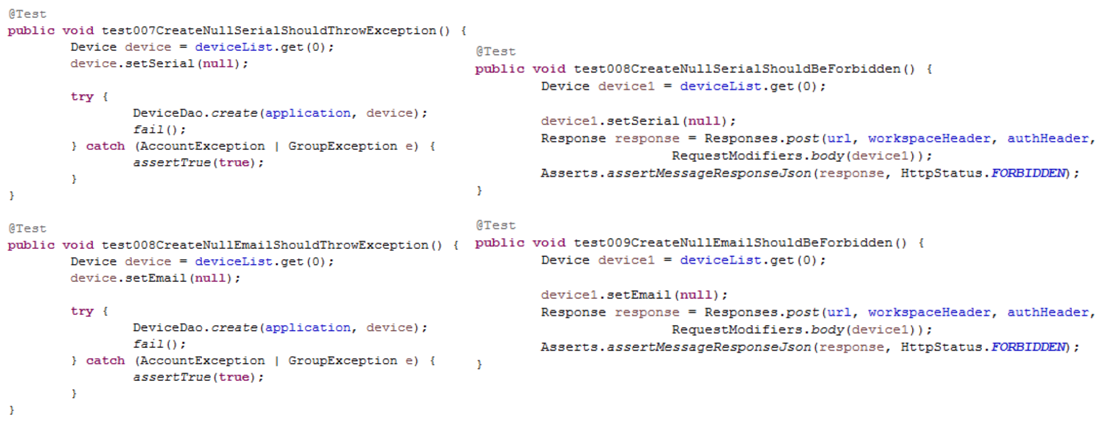
3.5 Testing

3.5.1 JUnit tests

Once the whole API is programmed, it has to be tested. For this purpose, some test classes are created in each of the different layers to check that every component is working as intended. Tests are also very helpful to ensure that the code still works as intended in case a modification of the code is done for fixing a bug or extending functionality.

This API implements testing with JUnit which is a simple framework to write repeatable tests. When there are several tests for the same components, they can be combined into a bigger class or test suite. Each test is a method with an annotation @Test in which a method provided by the JUnit framework is used to check the expected result of the code execution versus the actual result. In Figure 20 there is a sample of the tests for the devices DAO and controller class.

In the controller tests, RestServerDriver class from restdriver library is used to make http requests and the class Response to compare them with the expected results.



```java
@Test
public void test007CreateNullSerialShouldThrowException() {
        Device device = deviceList.get(0);
        device.setSerial(null);

        try {
                DeviceDao.create(application, device);
                fail();
        } catch (AccountException | GroupException e) {
                assertTrue(true);
        }
}

@Test
public void test008CreateNullEmailShouldThrowException() {
        Device device = deviceList.get(0);
        device.setEmail(null);

        try {
                DeviceDao.create(application, device);
                fail();
        } catch (AccountException | GroupException e) {
                assertTrue(true);
        }
}
```

```java
@Test
public void test008CreateNullSerialShouldBeForbidden() {
        Device device1 = deviceList.get(0);

        device1.setSerial(null);
        Response response = Responses.post(url, workspaceHeader, authHeader,
                        RequestModifiers.body(device1));
        Asserts.assertMessageResponseJson(response, HttpStatus.FORBIDDEN);
}

@Test
public void test009CreateNullEmailShouldBeForbidden() {
        Device device1 = deviceList.get(0);

        device1.setEmail(null);
        Response response = Responses.post(url, workspaceHeader, authHeader,
                        RequestModifiers.body(device1));
        Asserts.assertMessageResponseJson(response, HttpStatus.FORBIDDEN);
}
```

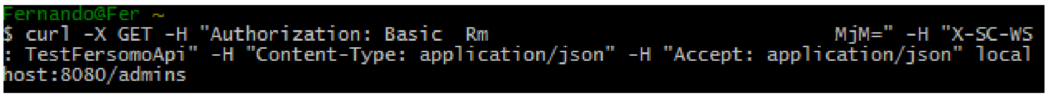
Figure 20: Tests samples

3.5.2 Testing as a User

To check that everything is working as intended, a simulation of a real use of the API is done. For this test, the API is run on a local server with path localhost:8080 and the requests are made using cUrl.

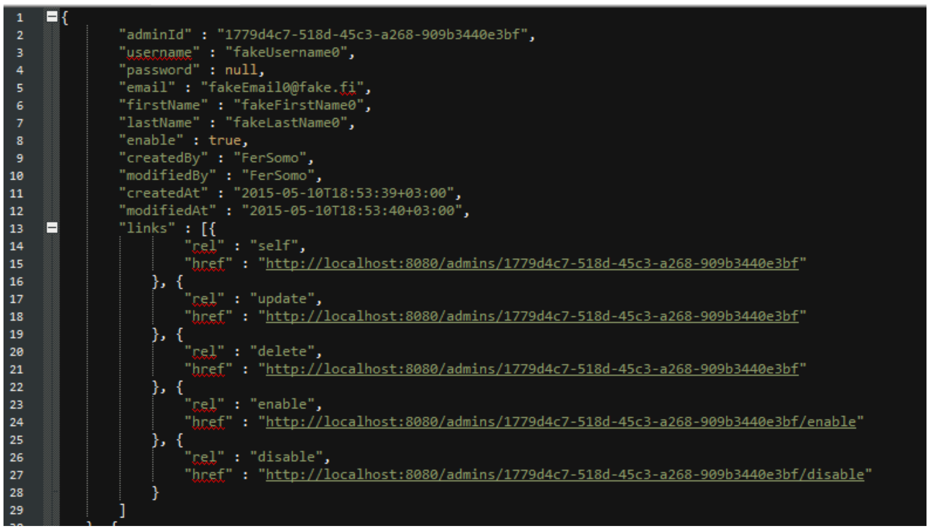
GETing a list of admins:

The controller has mapped the URI of the admins in /admins. With a GET request to that URI (Figure 21), the representations of the admins are received in the response.



```
Fernando@Fer ~
$ curl -X GET -H "Authorization: Basic  Rm[redacted]MjM=" -H "X-SC-WS
: TestFersomoApi" -H "Content-Type: application/json" -H "Accept: application/json" local
host:8080/admins
```

Figure 21: GET admins



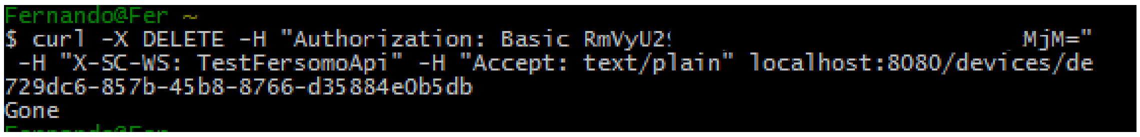
```
{
        "adminId" : "1779d4c7-518d-45c3-a268-909b3440e3bf",
        "username" : "fakeUsername0",
        "password" : null,
        "email" : "fakeEmail0@fake.fi",
        "firstName" : "fakeFirstName0",
        "lastName" : "fakeLastName0",
        "enable" : true,
        "createdBy" : "FerSomo",
        "modifiedBy" : "FerSomo",
        "createdAt" : "2015-05-10T18:53:39+03:00",
        "modifiedAt" : "2015-05-10T18:53:40+03:00",
        "links" : [{
                "rel" : "self",
                "href" : "http://localhost:8080/admins/1779d4c7-518d-45c3-a268-909b3440e3bf"
            }, {
                "rel" : "update",
                "href" : "http://localhost:8080/admins/1779d4c7-518d-45c3-a268-909b3440e3bf"
            }, {
                "rel" : "delete",
                "href" : "http://localhost:8080/admins/1779d4c7-518d-45c3-a268-909b3440e3bf"
            }, {
                "rel" : "enable",
                "href" : "http://localhost:8080/admins/1779d4c7-518d-45c3-a268-909b3440e3bf/enable"
            }, {
                "rel" : "disable",
                "href" : "http://localhost:8080/admins/1779d4c7-518d-45c3-a268-909b3440e3bf/disable"
            }
        ]
```

Figure 22: Sample of JSON representation receive

In the JSON representation of Figure 22, one can observe a field “Links”. Those links are the engine of the API. From the list of admins, the user can perform http verbs in order to perform actions such as enable or disable admins or devices, delete them, or simply get a representation of each of them. By doing a POST request into the root path and including the proper data on the body, the user can create new resources. And in case of a PUT request, a user is modified with the new data.

DELETEing a device:

When deleting, the response does not need to have any representation so there is no need of the headers supporting JSON. The response is a String with the status of the http request (Figure 23). This can also be seen in the header of the response.

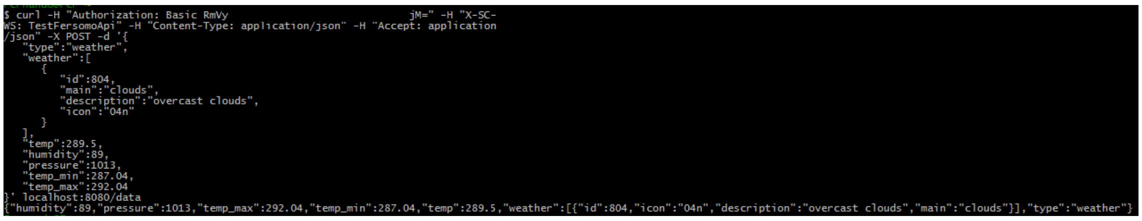


```
Fernando@Fer ~
$ curl -X DELETE -H "Authorization: Basic RmVyU2![redacted]MjM="
 -H "X-SC-WS: TestFersomoApi" -H "Accept: text/plain" localhost:8080/devices/de
729dc6-857b-45b8-8766-d35884e0b5db
Gone
```

Figure 23: DELETE device and HttpResponse

POSTing and GETing data:

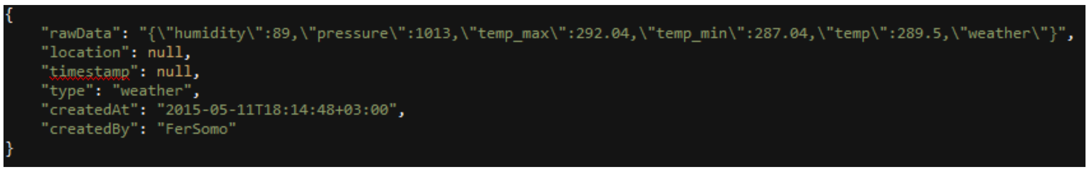
Any data with JSON format can be pulled to the persistent unit as long as one of the fields is “type”. The service creates a JSON object from a string to store it mapping each pair and value (Figure 24).



```
$ curl -H "Authorization: Basic RmVy[redacted]jM=" -H "X-SC-
WS: TestFersomoApi" -H "Content-Type: application/json" -H "Accept: application
/json" -X POST -d '{
   "type":"weather",
   "weather":[
      {
         "id":804,
         "main":"clouds",
         "description":"overcast clouds",
         "icon":"04n"
      }
   ],
   "temp":289.5,
   "humidity":89,
   "pressure":1013,
   "temp_min":287.04,
   "temp_max":292.04
}' localhost:8080/data
{"humidity":89,"pressure":1013,"temp_max":292.04,"temp_min":287.04,"temp":289.5,"weather":[{"id":804,"icon":"04n","description":"overcast clouds","main":"clouds"}],"type":"weather"}
```

Figure 24: POST of Data

By doing a GET request, the user is able to obtain the data (Figure 25). This data contains the raw data the user has pushed and includes other fields like the creator and the date of creation.



```
{
    "rawData": "{\"humidity\":89,\"pressure\":1013,\"temp_max\":292.04,\"temp_min\":287.04,\"temp\":289.5,\"weather\"}",
    "location": null,
    "timestamp": null,
    "type": "weather",
    "createdAt": "2015-05-11T18:14:48+03:00",
    "createdBy": "FerSomo"
}
```

Figure 25: JSON representation of data

4 CONCLUSION

The main goal of this thesis is to develop a usable API for managing admins and devices that could be able to store and manipulate data in the scope of Internet of Things. This goal has been achieved firstly by acquiring the necessary knowledge about REST, its elements and constraints, presenting how it works with HTTP and why HATEOAS is so important in RESTful APIs. Finally, the implementation of an API in Java language using Spring framework completes the research purpose.

Although the API created is usable, it is just a prototype or alpha version because its behaviour is quite limited. The API can be easily scalable adding more functionality, for example searches for the data which can be by location or by timestamp; or implementing a graphical interface which will make the API more user friendly. In addition, although the format chosen for this thesis is JSON, it can be extended to other formats such as XML without having to re-program everything, just adding or moddifying a layer on the implementation.

Internet of Things is going to be a reality in a couple of years if not before, and some forecast, assert that more than 26 billion of devices excluding PCs, tablets and smartphones will be connected (Gartner Inc., 2013). Thus, any company which produces any device and want them to be connected has to use an API with similar characteristics but with more functionality as the one developed in this thesis.

REFERENCES

ABI Research, 2013. More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020. Allied Business Intelligence, Inc. London, United Kingdom. Consulted 22.4.2015
https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/

Anderson J. and Rainie L. 2014. Main Report: An In-depth Look at Expert Responses. Pew Research Center Internet, Science & Tech. Consulted 22.4.2015
http://www.pewinternet.org/2014/05/14/main-report-an-in-depth-look-at-expert-responses/

Clements P.; Bachmann F.; Bass L.; Garlan D.; Ivers J.; Little R.; Merson P.; Nord R.; Stafford J. 2003. Documenting Software Architectures: Views and Beyond. Addison-Wesley.

ECMA-404. 2013. The JSON Data Interchange Format. Consulted April 2015
http://www.ecma-international.org/publications/standards/Ecma-404.htm

Fielding R. T. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine

Fielding R. T. and Taylor R. N. 2002. Principled Design of the ModernWeb Architecture. Information and Computer Science University of California, Irvine

Fielding, R. T. 2008. REST APIs must be hypertext-driven. Consulted 9.3.2015
http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Garlan D. and Shaw M. 1994. An Introduction to Software Architecture. School of Computer Science, Carnegie Mellon University.

Gartner, Inc. December 12, 2013.Press Release. Stamford, Connecticut. Consulted 22.4.2015
http://www.gartner.com/newsroom/id/2636073

Hazlewood, Les, 2012. Designing a Beautiful REST+JSON API.
https://www.youtube.com/watch?v=5WXYw4J4QOU

ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description, 2011. Consulted April 2015
https://www.iso.org/obp/ui/#iso:std:50508:en

JUnit 4.12 API Documentation. Consulted April 2015
http://junit.org/javadoc/latest/index.html

Orchestrate API Documentation. Consulted April 2015
https://orchestrate.io/docs/apiref

Perry D. E. and Wolf A. L. 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes.

RestDriver API Documentation. Consulted April 2015
https://github.com/rest-driver/rest-driver/wiki/Server-Driver

RFC 3986.2005. Uniform Resource Identifier (URI): Generic Syntax. Consulted April 2015
https://tools.ietf.org/html/rfc3986

RFC 7159.2014. The JavaScript Object Notation (JSON) Data Interchange Format. Consulted April 2015
https://tools.ietf.org/html/rfc7159

RFC 7230.2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Consulted April 2015 https://tools.ietf.org/html/rfc7230

Spring Framework 4.1.6. API Documentation. Consulted April 2015 http://docs.spring.io/spring/docs/current/javadoc-api/overview-summary.html

Spring HATEOAS 0.18.0 API Documentation. Consulted April 2015 http://docs.spring.io/spring-hateoas/docs/current-SNAPSHOT/api/

Shaw M. and Clements P. 1997. A field guide to boxology: Preliminary classification of architectural styles for software systems. COMPSAC '97. Proceedings.

Stormpath API Documentation Consulted April 2015 http://docs.stormpath.com/rest/product-guide