

TECHNICAL UNIVERSITY OF CRETE

THESIS

---

# Thesis Title

---

*Author:*

Stefanos Stathatos

*Committee:*

Associate Professor Vasilis Samoladas (Supervisor)

Associate Professor Minos Garofalakis

Associate Professor Kostas Petrakis

*A thesis submitted in fulfillment of the requirements  
for the degree of Electrical and Computer Engineer  
in the*

School of Electrical and Computer Engineering  
Technical University of Crete

August 16, 2017

Technical University of Crete

# *Abstract*

Technical University of Crete  
School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Thesis Title**

by Stefanos Stathatos

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

## *Acknowledgements*

The acknowledgments and the people to thank go here, don't forget to include your project advisor. . .

# Contents

<b>Abstract</b>	<b>1</b>
<b>Acknowledgements</b>	<b>2</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>2</b>
2.1 Models and Model Driven Software Development . . . . .	2
2.2 Data Access Object . . . . .	3
2.3 Access Control List . . . . .	4
2.4 HTTP . . . . .	4
2.4.1 HTTP session . . . . .	4
2.4.2 Request methods . . . . .	5
2.4.3 Status Codes . . . . .	5
2.5 REST . . . . .	7
2.5.1 Client-server architecture . . . . .	7
2.6 Application Programming Interface . . . . .	8
2.7 Three-tier Architecture . . . . .	9
2.8 Single Page Application . . . . .	11
2.9 Hierarchical Data Format . . . . .	11
2.9.1 Data Model . . . . .	13
<b>3 Related Work and Used Technologies</b>	<b>14</b>
3.1 Related Work . . . . .	14
3.1.1 Plotly . . . . .	14
3.1.2 Loopback . . . . .	14
3.2 Used Technologies . . . . .	15
3.2.1 Javascript . . . . .	15
3.2.1.1 Node.js . . . . .	15
3.2.1.2 Node Package Manager . . . . .	15
3.2.2 NoSQL Database . . . . .	16
3.2.2.1 MongoDB . . . . .	17

<b>4</b>	<b>Design</b>	<b>18</b>
4.1	Framework Design . . . . .	18
4.2	Application description . . . . .	18
4.3	Generic Framework Modules . . . . .	19
4.3.1	Nosql access . . . . .	19
4.3.1.1	Entities . . . . .	20
4.3.1.2	Data Access Object/ CRUD . . . . .	22
4.3.1.3	DbOperations . . . . .	23
4.3.1.4	Permissions . . . . .	23
4.3.2	Main server . . . . .	24
4.3.2.1	Helpers . . . . .	25
4.3.2.2	Python files . . . . .	27
4.3.2.3	Routes . . . . .	28
4.3.2.4	Defender . . . . .	28
4.3.3	User interface . . . . .	29
<b>5</b>	<b>Implementation</b>	<b>30</b>
5.1	General JavaScript practices and patterns . . . . .	30
5.1.1	Closures . . . . .	31
5.1.2	Thunks . . . . .	31
5.2	Back-End implementation . . . . .	33
5.2.1	REST API . . . . .	33
5.2.2	NoSQL Access Management . . . . .	36
5.2.3	Security Management . . . . .	37
5.2.4	Python - Node.js Interoperability . . . . .	38
	<b>Bibliography</b>	<b>41</b>

# List of Figures

2.1	Three-tier Architecture . . . . .	10
2.2	The contents of an HDF file . . . . .	12
2.3	The general structure of a dataset . . . . .	12
3.1	Developers survey for Stack Overflow website in 2017. . . . .	16
4.1	General Framework Design . . . . .	19
4.2	NoSQL access module structure . . . . .	20
4.3	Main server module structure . . . . .	25
5.1	Example of a callback function . . . . .	31
5.2	Example of a closure function . . . . .	32
5.3	Example of a thunk function . . . . .	32
5.4	Example of a User instance . . . . .	34
5.5	Example of a Project instance with three users . . . . .	35
5.6	Example of a Post instance . . . . .	36
5.7	Example of a Plot instance . . . . .	37

# List of Tables

5.1	Authentication URI's . . . . .	39
5.2	Users URI's . . . . .	39
5.3	Projects URI's . . . . .	39
5.4	Datasets URI's . . . . .	40
5.5	Posts URI's . . . . .	40
5.6	Plots URI's . . . . .	40

# Chapter 1

## Introduction



## Chapter 2

# Background

This chapter presents some background for the content of this thesis.

### 2.1 Models and Model Driven Software Development

Software development is a complex and difficult task that requires the investment of significant resources and carries major risk of failure. According to its proponents, model-driven (MD) software development approaches are improving the way we build software. Model-driven approaches putatively increase developer productivity, decrease the cost (in time and money) of software construction, improve software reusability, and make software more maintainable. Likewise, model-driven techniques promise to aid in the early detection of defects such as design flaws, omissions, and misunderstandings between clients and developers.

If a model is a representation of a system, then in some sense, programming in any language involves some kind of model. Whether we explicitly create artifacts we call models—especially conceptual models—or whether we implicitly map between our internal mental models of the world and the systems we produce, we are nonetheless involved in a modeling process as we construct software. And so MD is more about raising the level of abstraction of our programming models rather than introducing models into the process in the first place. Models help software engineers communicate more effectively with the many stakeholders who need to participate in the software development process. Improved communication leads to increased understanding, more reasonable expectations, and a better overall work product. Models also let programmers visualize the finished product without requiring its full construction first. By examining the model we can

discover design flaws that are far less expensive to resolve up-front rather than after construction has begun (or worse, been completed).

Model-driven software development [6] is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Model-driven architecture is a kind of domain engineering, and supports model-driven engineering of software systems.

The three primary goals of MDA are portability, interoperability, and reusability, and the key abstraction for delivering on these goals is architectural separation of concerns. MDA describes three main layers of architectural abstraction, called viewpoints: computation independent, platform independent, and platform specific. MDA describes systems using models that correspond to the three viewpoints. A computation independent model (CIM) describes a system environment and its requirements using terminology that is familiar to practitioners in the system domain. A platform independent model (PIM) describes a system's structure and functions formally, and yet without specifying platform-specific implementation details. At the lowest level of the MDA architecture, a platform specific model (PSM) includes details that are important to the implementation of a system on a given platform.

## 2.2 Data Access Object

In computer software, a data access object (DAO) [11] is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database. This isolation supports the Single responsibility principle. It separates what data access the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), from how these needs can be satisfied with a specific DBMS or database schema. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

## 2.3 Access Control List

An Access Control List (ACL) [10] is a mechanism that implements access control for a system resource by enumerating the system entities that are permitted to access the resource and stating, either implicitly or explicitly, the access modes granted to each entity. A filesystem ACL is a data structure (usually a table) containing entries that specify individual user or group rights to specific system objects such as programs, processes, or files. The privileges or permissions determine specific access rights, such as whether a user can read from, write to, or execute an object. +Role,Permission

## 2.4 HTTP

The Hypertext Transfer Protocol (HTTP) [3] is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. HTTP functions as a request-response protocol in the client-server computing model. A web browser, for example, may be the client and an application running on a computer hosting a website may be the server. The client submits an HTTP request message to the server. The server, which provides resources such as HTML files and other content, or performs other functions on behalf of the client, returns a response message to the client. The response contains completion status information about the request and may also contain requested content in its message body. HTTP resources are identified and located on the network by Uniform Resource Locators (URLs), using the Uniform Resource Identifiers (URI's) schemes `http` and `https`. URIs and hyperlinks in HTML documents form inter-linked hypertext documents.

### 2.4.1 HTTP session

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server (typically port 80, occasionally port 8080; see List of TCP and UDP port numbers). An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The body of this message is typically the

requested resource, although an error message or other information may also be returned.

### 2.4.2 Request methods

HTTP defines methods to indicate the desired action to be performed on the identified resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. The most common methods which are used in this thesis are presented below.

**GET** The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. The W3C has published guidance principles on this distinction, saying, "Web application design should be informed by the above principles, but also by the relevant limitations.

**POST** The POST method requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, for example, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.

**PUT** The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.

**DELETE** The DELETE method deletes the specified resource.

### 2.4.3 Status Codes

In HTTP/1.0 and since, the first line of the HTTP response is called the status line and includes a numeric status code (such as "404") and a textual reason phrase (such as "Not Found"). The way the user agent handles the response primarily depends on the code and secondarily on the other response header fields. Custom status codes can be used since, if the user agent encounters a code it does not recognize, it can use the first digit of the code to determine

the general class of the response. HTTP status code is primarily divided into five groups for better explanation of request and responses between client and server, and are presented below.

**1xx Informational responses** An informational response indicates that the request was received and understood. It is issued on a provisional basis while request processing continues. It alerts the client to wait for a final response. The message consists only of the status line and optional header fields, and is terminated by an empty line.

**2xx Success** This class of status codes indicates the action requested by the client was received, understood, accepted, and processed successfully.

**3xx Redirection** This class of status code indicates the client must take additional action to complete the request. Many of these status codes are used in URL redirection.[2] A user agent may carry out the additional action with no user interaction only if the method used in the second request is GET or HEAD. A user agent may automatically redirect a request. A user agent should detect and intervene to prevent cyclical redirects.

**4xx Client errors** This class of status code is intended for situations in which the client seems to have errored. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents should display any included entity to the user.

**5xx Server errors** The server failed to fulfill an apparently valid request. Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and indicate whether it is a temporary or permanent condition. Likewise, user agents should display any included entity to the user. These response codes are applicable to any request method.

## 2.5 REST

Representational state transfer (REST) [5] or RESTful web services is a way of providing interoperability between computer systems on the Internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations. In a RESTful Web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. The term is intended to evoke an image of how a well-designed Web application behaves: it is a network of Web resources (a virtual state-machine) where the user progresses through the application by selecting links, such as `/user/tom`, and operations such as GET or DELETE (state transitions), resulting in the next resource (representing the next state of the application) being transferred to the user for their use. There are six guiding constraints that define a RESTful system. These constraints restrict the ways that the server may process and respond to client requests so that, by operating within these constraints, the service gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability. If a service violates any of the required constraints, it cannot be considered RESTful. The formal REST constraints are represented below.

### 2.5.1 Client-server architecture

The first constraints added to our hybrid style are those of the client-server architectural style. Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

**Statelessness** The client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the

server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.

**Cacheability** As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.

**Layered system** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

**Uniform interface** The uniform interface constraint is fundamental to the design of any REST service. It simplifies and decouples the architecture, which enables each part to evolve independently.

## 2.6 Application Programming Interface

In computer programming, an Application Programming Interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer. An API may be for a web-based system, operating system, database system, computer hardware or software library. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables or remote calls. API uses are listed below.

**Libraries and Frameworks** An API is usually related to a software library. The API describes and prescribes the expected behavior (a specification) while

the library is an actual implementation of this set of rules. A single API can have multiple implementations (or none, being abstract) in the form of different libraries that share the same programming interface. The separation of the API from its implementation can allow programs written in one language to use a library written in another.

**Operating systems** An API can specify the interface between an application and the operating system. POSIX, for example, specifies a set of common APIs that aim to enable an application written for a POSIX conformant operating system to be compiled for another POSIX conformant operating system.

**Remote APIs** Remote APIs allow developers to manipulate remote resources through protocols, specific standards for communication that allow different technologies to work together, regardless of language or platform.

**WEB APIs** Web APIs are the defined interfaces through which interactions happen between an enterprise and applications that use its assets. An API approach is an architectural approach that revolves around providing programmable interfaces to a set of services to different applications serving different types of consumers. When used in the context of web development, an API is typically defined as a set of Hypertext Transfer Protocol (HTTP) request messages, along with a definition of the structure of response messages, which is usually in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

## 2.7 Three-tier Architecture

Three-tier architecture [8] is a client-server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. An image of a three-tier architecture can be seen in figure 2.1. The three layers are presented below.



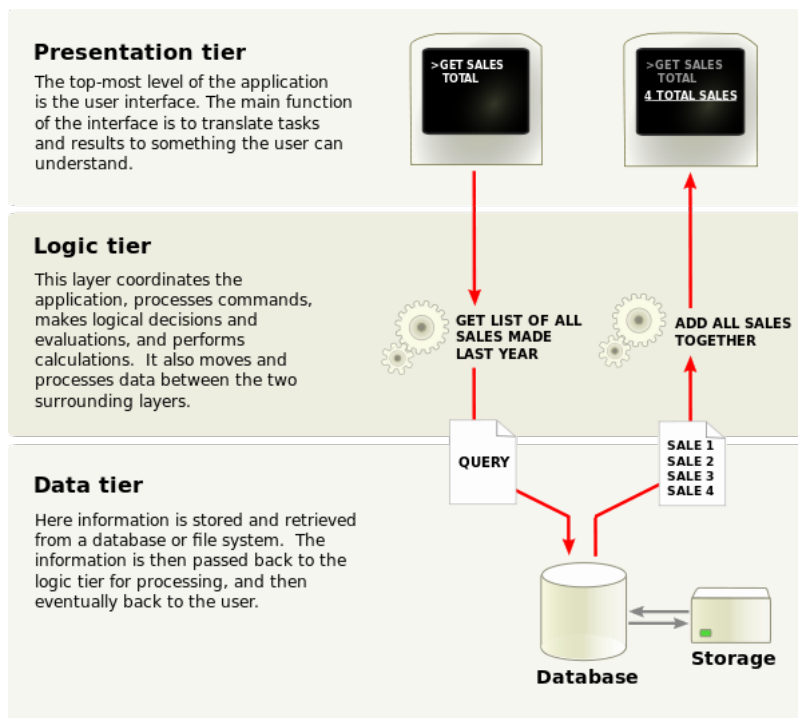


FIGURE 2.1: Three-tier Architecture

**Presentation tier** This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing and shopping cart contents. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it is a layer which users can access directly (such as a web page, or an operating system's GUI).

**Logic tier** The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing.

**Data tier** The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The data access layer should provide an API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms. Avoiding dependencies on the storage mechanisms allows for updates or changes without the application tier clients being affected by or even aware of the change.

## 2.8 Single Page Application

A single-page application (SPA) [7] is a web application or web site that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application. Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

There are various techniques available that enable the browser to retain a single page even when the application requires server communication. The most prominent technique currently being used is Ajax. Ajax is a set of Web development techniques using many Web technologies on the client side to create asynchronous Web applications. With Ajax, Web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for Web pages, and by extension Web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly substitute JSON for XML due to the advantages of being native to JavaScript.

## 2.9 Hierarchical Data Format

Hierarchical Data Format (HDF) [4] is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. Many HDF adopters have very large datasets, very fast access requirements, or very complex datasets. Others turn to HDF because it allows them to easily share data across a wide variety of computational platforms using applications written in different programming languages.

HDF allows hierarchical data objects to be expressed in a very natural manner, in contrast to the tables of a relational database. Whereas relational databases support tables, HDF supports n-dimensional datasets and each element in the dataset may itself be a complex object. Relational databases offer

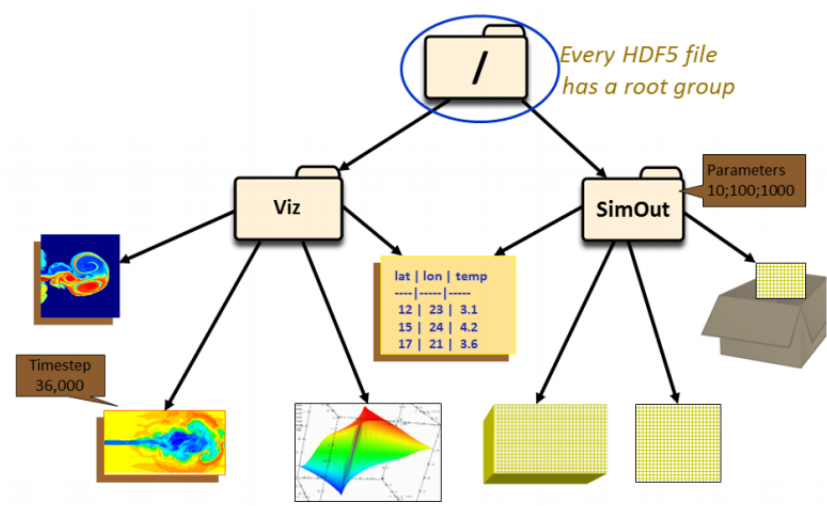


FIGURE 2.2: The contents of an HDF file

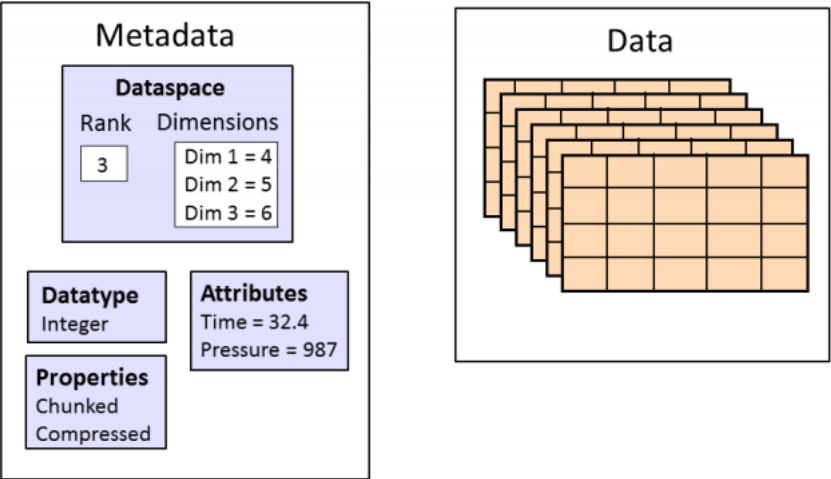


FIGURE 2.3: The general structure of a dataset

excellent support for queries based on field matching, but are not well-suited for sequentially processing all records in the database or for subsetting the data based on coordinate-style lookup. The contents of an HDF file can be seen in figure 2.2.

HDF5 consists of a File Format for storing HDF5 data, a Data Model for logically organizing and accessing HDF5 data from an application, and the Software (libraries, language interfaces, and tools) for working with this format. The data model is described below.

### 2.9.1 Data Model

The HDF Data Model, also known as the HDF5 Abstract (or Logical) Data Model consists of the building blocks for data organization and specification in HDF5. An HDF5 file (an object in itself) can be thought of as a container (or group) that holds a variety of heterogeneous data objects (or datasets). The datasets can be most anything: images, tables, graphs, or even documents, such as PDF or Excel. The two primary objects in the HDF5 Data Model are described below.

**Groups** HDF5 groups (and links) organize data objects. Every HDF5 file contains a root group that can contain other groups or be linked to objects in other files. Working with groups and group members is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, objects in an HDF5 file are often described by giving their full (or absolute) path names.

**Datasets** HDF5 datasets organize and contain the “raw” data values. A dataset consists of metadata that describes the data, in addition to the data itself. Datatypes, dataspace, properties and (optional) attributes are HDF5 objects that describe a dataset. The datatype describes the individual data elements. The general structure of a dataset can be seen in figure 2.3.

## Chapter 3

# Related Work and Used Technologies

This chapter presents some background for the content of this thesis.

### 3.1 Related Work

#### 3.1.1 Plotly

Plotly is a popular public data visualization cloud service provider. Plotly provides community, professional and enterprise data storage, visualization and analytics services to the user. Excel, CSV and XML data formats are used to upload the data to its cloud servers. Plotly provides online graphing, analytics, and statistics tools for individuals and collaboration, as well as scientific graphing libraries for Python, R, MATLAB, Perl, Julia, Arduino, and REST.

#### 3.1.2 Loopback

LoopBack is a highly-extensible, open-source Node.js framework which assimilates the best practices of model driven software development. LoopBack simplifies and speeds up REST API development. It consists of a library of Node.js modules for connecting web and mobile apps to data sources such as databases and REST APIs, a command line tool, and client-SDKs. A loopback application has three components: models that represent business data and behavior, data sources and connectors, and mobile clients. An application interacts with data sources through the LoopBack model API, available locally within Node, remotely over REST, and via native client APIs for iOS, Android, and HTML5. Using the API, apps can query databases, store data,

upload files, send emails, create push notifications, register users, and perform other actions provided by data sources. Loopback is implemented with many of the technologies we use in this thesis. It uses Mongoose as a general practice, data access objects for the communication between database and the REST API, access control list for authorization and is written in Node.js.

## 3.2 Used Technologies

### 3.2.1 Javascript

JavaScript (JS) [2], is a high-level, dynamic, weakly typed, object-based, multi-paradigm, and interpreted programming language. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production. It is used to make webpages interactive and provide online programs, including video games. As a multi-paradigm language, JavaScript supports event-driven, functional, and imperative (including object-oriented and prototype-based) programming styles. Initially only implemented client-side in web browsers, JavaScript engines are now embedded in many other types of host software, including server-side in web servers and databases. The most famous server-side Javascript implementation is Node.js.

#### 3.2.1.1 Node.js

Node.js [12] is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code server-side. Node.js provides an event-driven architecture and a non-blocking I/O API designed to optimize application's throughput and scalability for real-time Web applications. It uses Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a stand-alone Web server. The increasing popularity of Node.js in the last years can be seen in figure 3.1.

#### 3.2.1.2 Node Package Manager

Npm [9] is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via

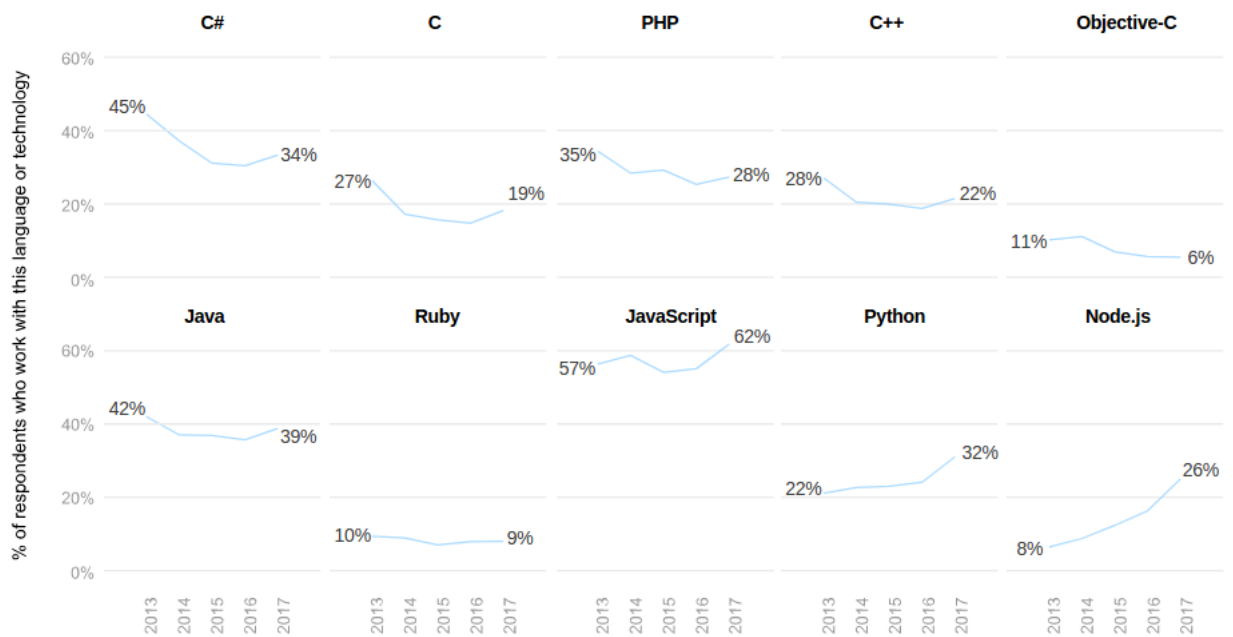


FIGURE 3.1: Developers survey for Stack Overflow website in 2017.

the npm website. Npm is included as a recommended feature in Node.js installer. Npm consists of a command line client that interacts with a remote registry. It allows users to consume and distribute JavaScript modules that are available on the registry. Packages on the registry are in CommonJS format and include a metadata file in JSON format.

### 3.2.2 NoSQL Database

A NoSQL (originally referring to "non SQL" or "non relational") [1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. NoSQL databases are increasingly used in big data and real-time web applications. Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases), and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.

### **3.2.2.1 MongoDB**

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas.



## Chapter 4

# Design

This chapter provides a detailed description of the designed system. In section 4.1, the general framework design diagram is presented, along with a simplifying description of each module the diagram contains.

### 4.1 Framework Design

In figure 4.1 a general framework design diagram is presented. The system architecture is divided in three parts: front-end/presentation tier, logic tier and data tier, as described in chapter 2.7. The data tier contains the database where all model data is stored, and the filesystem in which all uploaded datasets are saved in hdf form. The logic tier contains all the modules necessary for ensuring the framework functionality, such as crud operations, database access, authentication, authorization etc. The communication between logic and presentation tier is established over a rest api service. In the presentation tier, the user may interact with the server by sending requests and retrieve JSON data as responses. Before we continue in further details about the core system modules, next we describe the basic concepts of the application we created on top of our framework, in order to better demonstrate its capabilities.

### 4.2 Application description

The application makes use of the infrastructure which the framework implements, aiming to present an example of a real use case. Its purpose is to create user networks, which work on common projects with their data. These data could be text or multi-dimensional datasets. The application implements data visualization, with purpose the better understanding and conclusion extraction from the arithmetic datasets. The users may upload their

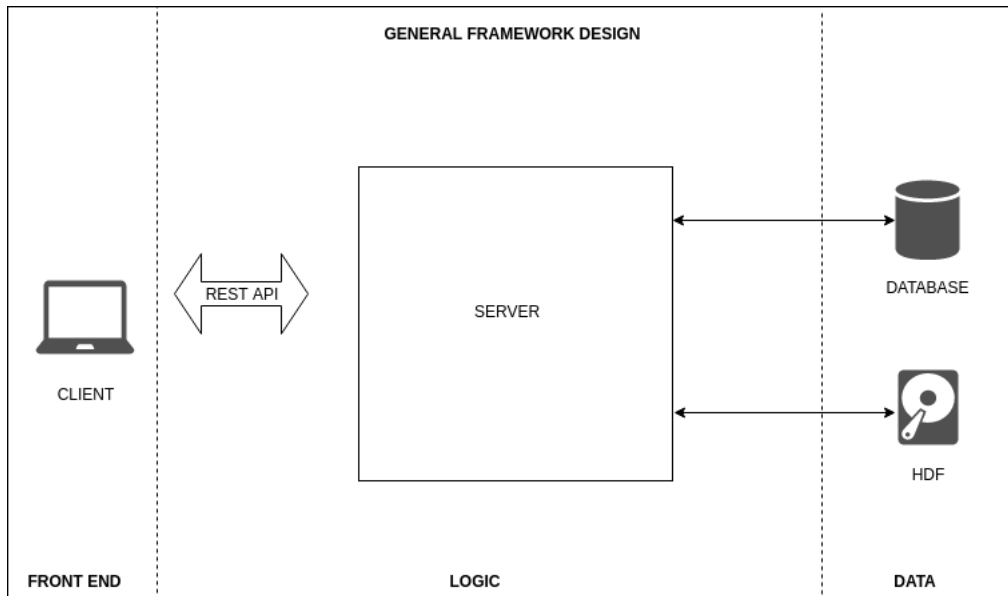


FIGURE 4.1: General Framework Design

datasets (in hdf form), in order to become visible from all the project users, while at the same time they may create plots based on them. In this chapter we describe the general design of the framework, but we provide some application design details when necessary. The detailed application implementation and functionality is presented in the next chapter.

### 4.3 Generic Framework Modules

In this section the framework modules used in this thesis are presented in detail. Diagrams are shown in some of these subsections for better understanding of the framework design. The modules are divided in three big categories: nosql access, main server and user interface. All modules and sub-modules are developed in a generic way, are highly customizable, extendible and reusable. This gives us the opportunity to often refer to these modules as models, as described in chapter 2.1. The level of abstraction may change, but the main characteristics of model driven software development are common in the framework implementation. Also, all modules are isolated and expose only the methods that must be used in the rest of the framework.

#### 4.3.1 Nosql access

The nosql access module is a very important component of the framework. It is responsible for a big part of the system's functionality, such as schema

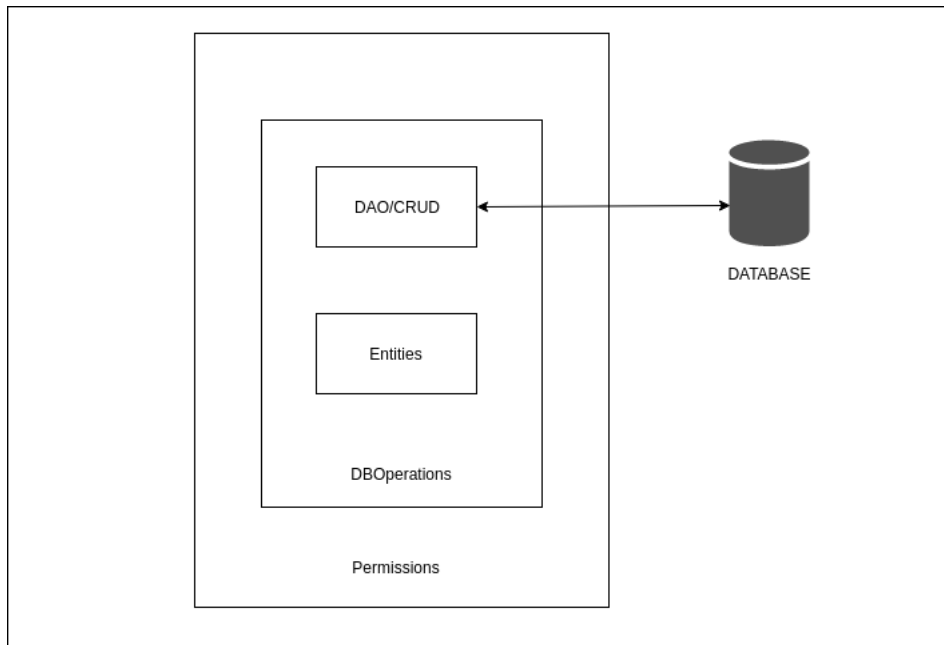


FIGURE 4.2: NoSQL access module structure

creation, CRUD operations generation, communication between server and database etc. As a result of its importance, this module is used as a dependency in many other models of the framework. This model is developed in such a way, so its functionality is isolated from the rest of the framework. Aside from the fact that this practice applies to the model-driven approach, it is generally a clean way for developing frameworks. The structure of the module can be seen in figure 4.2. Nosql access module consists of many submodules and each of those is presented separately below.

#### 4.3.1.1 Entities

An entity is a representation of a database resource. The creation of an entity through our framework, corresponds to the creation of a collection in our database. Likewise, the creation of an instance of an entity, corresponds to the creation of a document inside a collection in our database. In this way it becomes very easy to make changes to the database, without having direct access to it, which is crucial for our framework.

In the entities module, we define the entities we are going to use in the framework. More specific, we define the structure of the entities we use, their fields, their fields type, as well as the fields restrictions. For example, we can define if a field must be unique in relation with the rest of the instances in the same collection of the database. The definition of the entities is written in JSON form, so it becomes really easy to change or add more entities in

the database. So, this module obtains the property of extensibility, which is really important for the model-driven architecture we created.

For the purpose of the application demo, we defined a list of entities with specific fields, field types and restrictions. A visualization of this list is shown in figure MPLAAAAA. An analysis of each of the entities can be seen below.

**Users** In order to use the application, a user is obligated to create a new user account. The necessary properties to create a new account are username, full name, password and email. Username and email must have unique values in relation with all other user instances. The entity Users contains all necessary fields to save the above information, in the addition of a unique id.

**Projects** A basic concept of the application is the project entity. The application users may create new projects, in which multiple users may participate. The content of a project is visible for all users that participate in it. The required fields for the creation of a project instance are name, date and description. The project entity uses the concept of ACL, as described in chapter 2.3. Thus, project entity contains information about which users can or cannot, use a CRUD operation for the specific instance of the project entity. This way, the fields that contain this information reference an instance of a user entity.

**Datasets** Another usefull entity of the application is Datasets. The application users may create their own datasets inside a project they participate. In this entity are saved the metadata of an hdf file which was uploaded on the application. The fields of this entity are the dataset name, date, a reference to the user instance who creates the entity, and the hdf filename. Also, a project instance reference, in which the dataset is in, is saved. Last, just as in project entity, information is retained about the access control list of the dataset.

**Posts** An application user may create a new post inside a project. The required fields for the creation of a post entity are title, description, date, a dataset instance reference and a project instance reference. Also, acl information is saved in the post entity instance. The user may create a post in response of another post. In that case, information is retained about the post instance in reference.

**Plots** A user may add plots inside his posts. The required fields for the creation of a post entity are title, description, the path inside the hdf file in

which the array used for the plot is saved, and the post instance reference. Also, the plot entity retains information about its metadata, such as plot type and the dimension values used in the plot.

#### 4.3.1.2 Data Access Object/ CRUD

As described in chapter 2.2, DAO provides some specific data operations without exposing details of the database which are not needed. In our case, the dao model ensures that it is the only module with access in the database, and exposes only the information and CRUD functions which are vital for the rest of the framework. The dao module is developed in a generic way and it is customizable by the entities module. Thus, for each of our entities, we generate CRUD operations for interaction between the specific database model and the framework. The CRUD methods used in the dao module are described in detail below.

**createItem** The method `createItem` is responsible for the creation of new instances of an entity. It receives an object as an argument, it converts it in model instance form and then saves it. If an error occurs, it returns a new `Error` object. Based in the single responsibility principle, the method does not control the kind of data that are about to be saved.

**readItems** The method `readItems` is used to read data from the database. It receives a query object as an input. The result, successful or not, is returned as an array object. If an error occurs, it returns a new `Error` object.

**updateItem** The method `updateItem` is responsible for updating an existing object in the database. To achieve that, it receives as arguments a query object and an object with the new property values. If found, the object is replaced and returned. If an error occurs, it returns a new `Error` object.

**deleteItem** The method `deleteItem` is used to delete an existing document from the database. It receives a query object as an input, and if the document is found, the method deletes it from the database. If an error occurs, it returns a new `Error` object.

#### 4.3.1.3 DbOperations

As mentioned in the section 4.3.1.2, the DAO module is responsible for interacting with the database. The model methods are interacting with the database, perform direct changes and may return a result where applicable. But is there a way to guarantee the integrity of our operations in live data? Throughout our study, we determined that all functions must be wrapped in functions responsible for the verification and validation of the requested data mutation. So in essence, we provide a single sandboxed environment securing the database from malicious adversaries, as well as potential internal missuse.

The DBOperations model was developed for this purpose. It receives as an argument the entities and the dao models. The DBOperations model returns a group of functions which control the data they receive as an input and then call the corresponding dao model methods.

The verification of the data is essential for the CRUD operations. In our case we primarily check if all the required properties and references are contained in the input object. Then, we verify that the references values are valid id's of the corresponding entities instance. In order to achieve that, the necessary read operations in the database are made.

#### 4.3.1.4 Permissions

In chapter 4.3.1.1, we described that most of the entities contain some fields that are responsible for the access control list. To be able to alter these fields in relation with who can access or not a resource, we developed a module specialized for this task. This module contains some functions which implement the acl functionality. The dependencies of permissions model are entities and DBOperations models.

The Permissions modules defines all the roles the framework uses. At anytime we can add or remove a role, therefore the model-driven design approach is implemented. This model is also responsible for the verification of the changes that it is about to make. The module exposes only the desired methods, so that any unwanted functionality is isolated. The generic methods we developed are described below.

**addUserRole** This method adds a specific role to a user for the given resource. It receives four arguments as an input, the user id, the object id, the

role and the model name. All arguments are verified before proceeding with the entity instance update.

**removeUserRole** RemoveUserRole is a method that does the opposite of the addUserRole. It removes from a user a certain role of a resource. It also receives the same arguments as an input. All arguments are verified before proceeding with the entity instance update.

**isAllowed** This method is the core of the acl logic. Its purpose is to check if a user has the permission to perform a crud operation in a resource instance. To accomplish that, this method checks if a user id is contained in the model instance of the resource id which is given as an input. If its not found, the same check is performed in the parent model instance, if it exists.

The reason behind this extra check, is that the parent model instance, may have been given a default order which applies for every kid model instance. This operation has the advantage, that we reduce many database operations, since we dont have to check the access permission of -usually many- users, each time we create a new kid model instance. On the contrary, we set a general rule which applies in all cases, unless it is overridden by another permission access.

An example of this functionality is the creation of a post inside a project. One way to apply permission access, is to add all user id's which have read permission for the post, inside its instance. It is preferable though, to set a general rule inside the project which states that, all users who have read access to this project, have also access to its posts.

**isAllowedCreate** isAllowedCreate had the same functionality as isAllowed method, but it focuses exclusively on create operation, which has slightly different functionality than the rest of them.

### 4.3.2 Main server

In this section we describe the core framework's component. The structure of this module can be seen in figure 4.3. To achieve that, we present all the sub-modules that are used in this model, and combined together are creating its complete functionality. Thus, apart from routes module which is responsible for receiving requests and sending responds to the client, it is also described the helpers module, which is containing a list of functions necessary for this

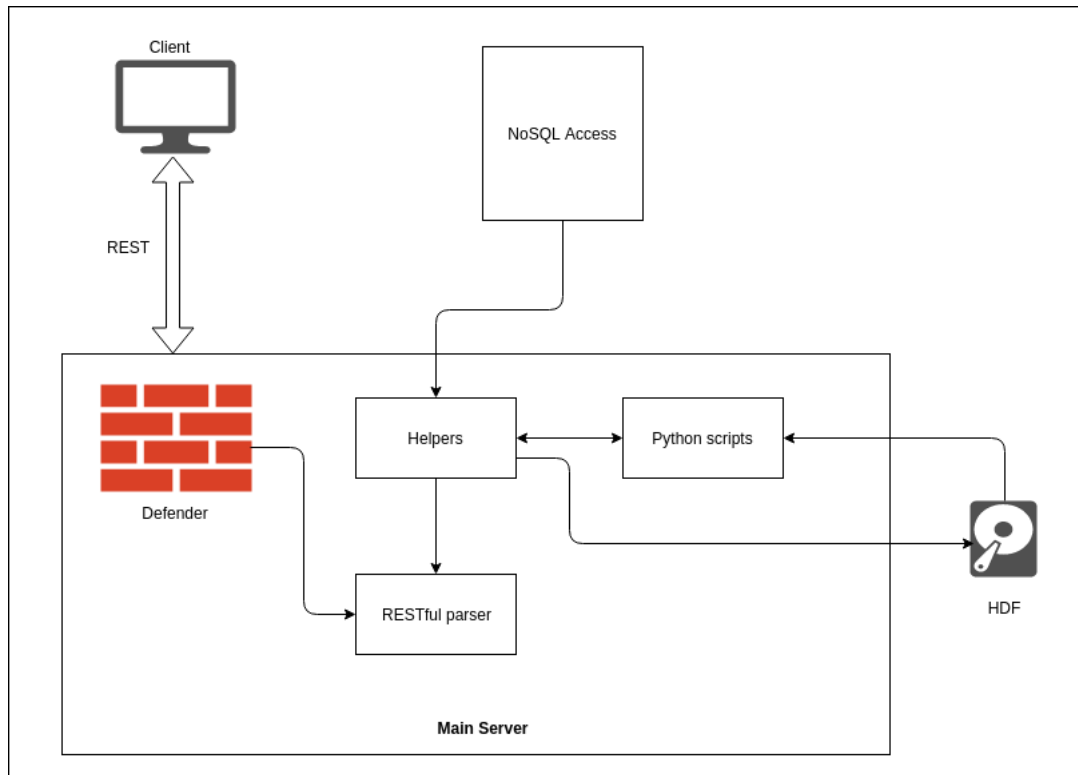


FIGURE 4.3: Main server module structure

model. Also, in this chapter we present a python bridge model, which is used for the utilization of the hierarchical data format, as well as our error handling model, our session setup module and the defenders module. Next we describe each of these modules.

#### 4.3.2.1 Helpers

In chapter 4.3.1 we defined a list of methods which are responsible for the communication with the database. In order to implement the framework's functionality though, it is necessary to combine these methods into a higher level module. The helpers model is developed for this reason and receives as an argument the nosql access module. Also, this module has a dependency on a list of python files, which are vital for the retrieval of saved hdf files (hdf explained in chapter 2.2). The returned result of this module's methods is the one that is sent to the client.

Next we are going to present some of these functions. Most of them are developed in conjunction with the application demo, but the module is extendible for any usage. We will skip the functions which are related to the python files. When the python module's purpose is explained, we will come back to present the corresponding functions.



**saveData** The function `saveData` is used for the saving of an uploaded hdf file in the filesystem. The uploaded file is divided in chunks of data, if it exceeds a specific limit of size. Initially, the method checks if the extension of the file is `.h5`, and then it collects all the chunks and concatenates them. When the reconstruction of the hdf file is completed, the file is renamed with a unique name and is saved in a specific path inside the filesystem. Then, the function returns the new file name, so it can be saved in the Dataset entity later. In case an error occurs in the parsing state, an error object is returned.

**searchRelatedPosts** As explained in section 4.3.1.1, in the context of the application a Post may be created in response of another Post. We define a parent post as a post that does not come in response of another post, and a kid post as a post that comes in response of another post, respectively. This function receives as an argument a post id, and is searching all the post instances for related posts, kids or parent. The result object of posts is returned in chronological order, ready to be sent to the client.

**confFunctions** `confFunctions` is a group of methods which are used for the proper configuration of the corresponding objects. It is common that some of the properties that an object contains, must be removed before the object is sent to the client. For example, a project entity instance does not need to contain the `acl` property, if it is intended to be sent to the client. Thus, these functions are removing the unnecessary properties from the corresponding objects. If an error occurs, an error object is returned.

**errorHandler** A very important aspect of a framework, is the way in which the system handles its errors. In general, exceptions [1] are divided in two major categories, those that are created from the client's wrong behaviour, and those that occur from poor software design, also known as bugs. The process with which the framework is managing its errors or exceptions of the system, is called error or exception handling. In a framework environment, a developer must foresee all the possible exceptions that may occur and provide a handling solution for all of them. The framework must be designed accordingly, by creating a general way of exception handling.

PREPEI  
NA MPEI STO IMPLEMENTATION ORIZONTAS TO ERRORHANDLING  
SERVICE

The framework's error handling, concerning the back end side of the system, sends all possible errors to the `errorHandling` function. This function categorizes the error and defines an according error status and message. Then, the function sends the response object to the client.

#### 4.3.2.2 Python files

Some of the functions that are contained in the `helpers` module are dependent to a python model, which includes a list of python modules. Next we are going to explain the functionality of these modules, and leave the `node.js-python` communication for the implementation section.

In general, the python modules are used for the interaction with the `hdf` files. As explained in chapter 2.2, the `hdf` contains multidimensional datasets that are saved in a binary form in the filesystem. But in order to make use of this data, we must first convert it from this binary form into floats. The purpose of the python modules is to accomplish exactly that. We present each of these modules below.

**getHDFContent** This python script is searching for all dataset arrays inside an `hdf` file, and returns their metadata. The script receives as an argument the name of the `hdf` file. Then the function recursively searches all the possible paths inside the file and locates the datasets. The module retains information about each dataset name, shape, size and number of dimensions. This metadata info is crucial for the framework, and its necessary for its functionality.

**getHDFArray** This script is responsible for returning a chunk of a specific dataset. In order to do that, the module receives a number of arguments, such as the array path, and some state properties. These are used to specify which part of the -usually big- dataset must be returned. Also, if the dataset is three-dimensional, an extra parameter is send to declare which dimension's value is going to be used.

**getHDFPlot** The last script is developed to return a chunk of a dataset which is going to be presented in a plot. It is similar to the `getHDFArray` script, with the exception of the zoom functionality and the sampling method. In the `getHDFPlot` module, a maximum size of chunk is defined, and based on that, a sampled part of the requested dataset is returned. If the dataset is small in comparison with the maximum size of chunk, the sampling frequency is high. In other case, the sampling frequency is small.

The problem with this method is that if the sampling frequency is small, it is impossible to detect details of the dataset. Thus, extra parameters are included, related to which part of the dataset must be returned.

In helpers module, a list of functions which use the python model is defined. Thus, the helpers model is dependent to the python model we described. The basic functionality of these functions is to call the python scripts, parse their results and return it to the next function, respectively. These functions also check for errors that may occur in the python scripts.

#### 4.3.2.3 Routes

This module is almost exclusively responsible for the communication between server and clients. It is the model which receives the request, distributes it in the corresponding operation and sends the final result back to the client. The routes model implements the restful parser, as described in chapter 2.5. It includes all the routes which are integral for the framework's functionality. The model is a requirement for many other modules, such as defender, session, the NoSQL access module, helpers etc.

Routes module combines different operations to achieve its own functionality. The core requirement module is the helpers, from which it receives the result that is in many cases ready to be sent to the client. Routes is a middleware module, meaning that it mediates in the execution of every request between itself and the respond object that is about to be sent to the user. Furthermore, the defender module is a middleware module which precedes the execution of the routes module, and is analyzed below.

#### 4.3.2.4 Defender

The defender module is responsible for the authentication and authorization of the framework's users. Because of the model's middleware functionality, for each request the server receives, the defender module executes to ensure the framework security. In order to achieve that, the system checks if the user who sent the request is identified, a procedure which is called authentication. Then, the framework implements an authorization check, that is to check if the same user is allowed to send the current request. In any case that some of the above operations are not successful, the user access in the system is denied.

The authentication process, although vital for the system, it comes in a great cost, because of the constant user identification check with the database. Thus, the framework implements the session management through cookie parsing. During that process, the user logs in the system for the first time, and then receives a cookie which contains all his credential information. From then on, in each future request, the user sends back the cookie in the framework for identification. This way, the system ensures both security and low cost in database access.

### 4.3.3 User interface

Up until now, we have only examined the backend module of the framework. The user interface module is the front end layer of the three-tier architecture, as described in chapter 2.7. The addition of this module in the already described framework functionality, completes the architecture component which is related to communication between server and client.

The user interface module is developed in such way, that it implements the single page application architecture (see chapter 2.8). This means that in the initialization of the communication between server and client, the client receives all the mandatory resources for the framework operation. These include all the web complaint resources, including HTML5, CSS3 and javascript. Since this point after, there is no page reload or further code execution. Furthermore, this model implements the model-driven approach, and because of that the developed components are extensible and reusable. We deepen in the front end's implementation in the next chapter.

## Chapter 5

# Implementation

In this chapter we describe the framework and application implementation. We refer to the technologies we used and their interaction. In some cases we cite some crucial code chunks and explain them. Initially we describe the REST API and we analyze its contents. After an extensive illustration of the back end tier of the framework, we emphasize in the front end. There we mention the technologies that are used and their implementation. Alongside the front end description, we present some fragments of front end use cases.

### 5.1 General JavaScript practices and patterns

In the framework we developed, we make use of javascript in both front and back end. This programming language uses asynchronous logic, which takes place through the usage of asynchronous callback functions. A callback function is a method which receives as an argument another method, in order to call it later. The execution of the first function does not block the execution of the rest of the program commands. When the execution of the first function ends, the execution of the callback function begins. An example of a callback function can be seen in figure 5.1.

In our framework, the callback functions are used constantly, thus the design and implementation are transformed accordingly. As a convention, all the callback functions receive as arguments two objects, the error and the result. Before the execution of the callback function, in case an error occurs, the execution stops and the error object is passed to the callback. The result object is obviously null. If no errors occur, the error object is null and the result object contains the function outcome. Next we describe two design patterns, which are based in callback functions and are heavily used in the framework implementation.

```
let myFunction = (cb) => {  
    //do stuff  
    //if error occurs  
    if(err) return cb(err,null);  
    //ready to return  
    return cb(null, result);  
}  
  
myFunction((err, result) => {  
    if (err) throw new Error(err);  
    // do stuff with the result  
})
```

FIGURE 5.1: Example of a callback function

### 5.1.1 Closures

Javascript is an object oriented language and is created to perform well in any platform. Thus, it has many ways of defining certain structures, such as classes. In our framework, we define our classes through the closure design pattern. Closures fully implement a class functionality.

Generally, a closure in javascript is a function. This function receives arguments that initialize its state. The function body defines its contents, such as variables, objects and other functions. In the end the function returns, or exposes in the rest of the framework, only what is necessary. This may include also variables, objects and functions. Everything that is not returned, equals to what classes define as private. An example of this pattern is shown in figure 5.2.

### 5.1.2 Thunks

The thunk pattern has a similar structure with closure, but its used in a different way. Basically, its a function which returns another function. The interesting part is that the execution of the first function does not imply the execution of the second. This way we define lazy functions, meaning methods which are scheduled to execute, but they are not until they have to. This pattern

```
let closure = () => {  
    let var1,var2;  
    let obj = {...  
    };  
  
    let inside_function1 = () => {  
        //does some stuff  
    }  
  
    let inside_function2 = () => {  
        //does some stuff too  
    }  
  
    return {  
        var1, obj, inside_function1  
    }  
}  
  
let closure_contents = closure();  
// contains var1, obj, inside_function1
```

FIGURE 5.2: Example of a closure function

```
let thunk = () => {  
    return () => {  
        //do some stuff  
        return;  
    }  
}  
  
let myFunction = thunk();  
//do some more stuff  
myFunction();
```

FIGURE 5.3: Example of a thunk function

is important for solving a problem that occurs in javascript, known as call-back hell. This problem takes place due to the generation of large sequence of nested callback functions, which is considered a bad design practice and creates debug problems. An example of the thunk pattern is shown in figure 5.3.

## 5.2 Back-End implementation

In this section we focus in the back end implementation of the framework. Initially we present the REST API and then we describe the development stages of the framework modules. We emphasize in the technologies used and their in between communication.

### 5.2.1 REST API

An important part of the back end framework implementation is the development of the rest api, which is achieved through routing. Routing refers to determining how an application responds to a client request for a specific endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, PUT or DELETE). Each of our routes has different route handler functions, which are executed when the route is matched. The route handler functions use the information, which is given to them through the request, and after the execution of internal operations, return a response object to the client.

The response object has a concrete structure and includes all the required information. Among other things, it contains the status code and the result data. Generally the content of the response object is determined from the request method and the success of the operation (errorHandling is described thoroughly in chapter FDHJKSAJFJKASDJKJFAS). The structure of the response object is in JSON format.

Next we present all the routes for each model entity separately. For some of them we describe their corresponding handlers.

The routes of the table 5.1 are used from the framework for the authentication of the users. The login route checks whether the client's credential exist in the users model of the database. If the client is identified, a positive confirmation is sent to the user, while at the same time a session cookie



```
{
  "_id": ObjectId("59528548557cac377763ecd1"),
  "name": "stefanos",
  "username": "stefanos",
  "email": "stefanos@stefanos.com",
  "password": "stefanos",
  "__v": 0
}
```

FIGURE 5.4: Example of a User instance

is saved (session management is described in chapter FDHJSAFAS). In opposite case, the server's response is negative. The logout route disconnects the user from the framework and its corresponding session cookie is deleted from the database. The last route is used from the framework to check if the user is already authenticated. If this event occurs, the authentication operation is skipped.

**Users** In table 5.2 are presented the routes which concern the users model. The POST method is responsible for the creation of new user instances. It checks whether all fields have a value and the username and email properties are unique. The figure 5.4 presents the structure of a User instance. The GET method returns the user instance information and the projects, in which he participates. The PUT and DELETE methods are used for their corresponding functionalities.

**Projects** The Project model rest api is presented in table 5.3. The POST method is used for the creation of project model instances. The figure 5.5 presents the structure of a Project instance. The GET method searches out for a specific project and its elements. Then the datasets and posts which are related to the projects, are returned along with the project information. The join route is responsible for the addition of project members. In order to implement it, an update of the acl property of the project instance takes place.

**Datasets** The routes of the table 5.4 are developed for the Dataset model. A general analysis of the dataset save and retrieval management is described in chapter TADEDE. The GET method returns the contents of an hdf file. The POST and DELETE methods are responsible for the creation and deletion of a dataset, respectively. The list route returns a list of datasets, which are contained in a project. Also, the grid route returns a chunk of an array, which is located inside a dataset, in order to present it to the client.

```

{
  "_id" : ObjectId("595cc59c8a9b1e1a936447b0"),
  "name" : "many users project",
  "description" : "a project with many users.",
  "date" : ISODate("2017-07-05T10:55:24.644Z"),
  "acl" : {
    "delete" : {
      "deny" : [
        ObjectId("59528548557cac377763ecd4"),
        ObjectId("59528548557cac377763ecd3")
      ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    },
    "create" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1"),
        ObjectId("59528548557cac377763ecd4"),
        ObjectId("59528548557cac377763ecd3")
      ]
    },
    "update" : {
      "deny" : [
        ObjectId("59528548557cac377763ecd4"),
        ObjectId("59528548557cac377763ecd3")
      ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    },
    "read" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1"),
        ObjectId("59528548557cac377763ecd4"),
        ObjectId("59528548557cac377763ecd3")
      ]
    }
  },
  "__v" : 0
}

```

FIGURE 5.5: Example of a Project instance with three users

```

{
  "_id" : ObjectId("59590b127156f50deb284416"),
  "title" : "new post",
  "description" : "new post",
  "inproject" : ObjectId("59529063e453783c84a9b43d"),
  "dset_link" : ObjectId("5952b2f55ed1754eae0d5dc6"),
  "date" : ISODate("2017-07-02T15:02:42.113Z"),
  "acl" : {
    "delete" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    },
    "create" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    },
    "update" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    },
    "read" : {
      "deny" : [ ],
      "allow" : [
        ObjectId("59528548557cac377763ecd1")
      ]
    }
  },
  "__v" : 0
}

```

FIGURE 5.6: Example of a Post instance

**Posts** In table 5.5 are presented the routes which are related to the posts model. All the methods are used for the corresponding functionalities. The figure 5.6 presents the structure of a Post instance.

**Plots** The Plot model rest api is presented in table 5.6. All the methods are used for the corresponding functionalities. The figure 5.7 presents the structure of a Plot instance.

## 5.2.2 NoSQL Access Management

In chapter 4.3.1 a thorough description of the design of the NoSQL access module was presented. In this chapter we mention the technologies we used

```
{
  "_id" : ObjectId("59590b507156f50deb28441b"),
  "inpost" : ObjectId("59590b507156f50deb284418"),
  "title" : "new plot",
  "description" : "new description",
  "array_path_saved" : "subgroup1/d2dset",
  "plot_metadata" : {
    "dim1" : 1,
    "dim2" : 2,
    "dim2Value" : 79,
    "plot_type" : "line"
  },
  "__v" : 0
}
```

FIGURE 5.7: Example of a Plot instance

and the implementation of the module.

A standard library we use in this module is Mongoose.js. This library offers a schema based solution in the modeling of the database, for node.js. It grants complete CRUD operations which we use in data access object module. These operations are developed as callback functions, so the execution of the function does not stop the code flow. When the execution of a CRUD operation is completed, callback function is called. Also, the data communication between the internal modules and the mongoose library is implemented through javascript objects.

### 5.2.3 Security Management

In chapter [4.3.2.4](#) we presented the general idea of the security design. Here we describe how these modules are combined to implement the security objective. Also, the technologies we used are presented. Initially, we introduce the passport javascript module, which is the key role in session management.

Passport is a node.js library which is used for the user authentication. When a user logs in the framework, the library creates a cookie object, which is saved in a database collection. Also, it includes this cookie to the respond object, which is sent to the client. This operation is called serialization. The client saves the cookie object and sends it back in the future requests. When the user logs out or a specific amount of time has passed without connecting to the system, the passport library deletes the user cookie from the database. This operation is called deserialization. The framework ensures the correct communication between the login, logout routes and their corresponding handlers, the passport library and the database collection.

The defender middleware module combines the authentication/session management functionality we described above, with the authorization functions that are introduced in the NoSQL access module. Initially, the system investigates if the request object includes the required cookie, so the authentication can be completed. If the operation succeeds, then the request object URI is examined. With some exceptions, the URI's first part ought to be part of one of the models, that the database provides. Also, the request method must be one of GET, POST, UPDATE or DELETE.

The last part of the security module is the authorization investigation of the framework. As mentioned in chapter 4.3.1, the functionality of the permissions model, includes functions which investigate over permissions of a model instance. At this point we use `isAllowed` and `isAllowedCreate` functions to check if the request is allowed to proceed. If the examination is successful, the middleware operation is completed, and the routes module is called. In all situations that the check is failed, the access is denied for the user and a corresponding status code is returned (401 if not authenticated and 403 if not authorized).

#### 5.2.4 Python - Node.js Interoperability

As aforementioned, HDF is an essential ingredient of the framework for storing multidimensional datasets. The HDF library is developed in C/C++, which constitutes it as foreign code to our framework. Firstly we studied ways of embedding foreign code in node.js. An option for establishing inter-processing communication (IPC) is over Standard Input/Output (STDIO), available in all major operating systems.

Although an opensource project which exposes the HDF functionality directly to node exists, we observed prohibitive inefficiencies in its implementation. Thus, we set upon a more mature python implementation, the `h5py`. A module responsible for sending messages from our framework to `h5py`, and inversly, was developed. The rest of this section is dedicated to the way this module operates. For visual reference, see figure `AJSDFKAJFKDASK`.

Node includes a spawn functionality of the `child process` module, which allows invocation of external processes, parameterized by arguments. The results of the python execution, are manipulated by the event listener, returned by the child process. The controller listens for the data and error events. In case of and error, a new error object is propagated, while in the opposite case the result is parsed.

URI	method
/login	GET
/logout	GET
/isauthenticated	GET

TABLE 5.1: Authentication URI's

Model	URI	method
Users	/users/?id={user_id}	GET
Users	/register	POST
Users	/users/?id={user_id}	PUT
Users	/users/?id={user_id}	DELETE

TABLE 5.2: Users URI's

We used the included python's json library, which enables easy information parsing to a javascript object. Via the STDOUT buffer, all the results are serialized and accessed from javascript. We designed a communication protocol between python and node.js, for the customization of the script functions. Python executes the corresponding commands, returning successful results or potential errors.

Model	URI	method
Projects	/projects/?id={project_id}	GET
Projects	/projects	POST
Projects	/projects/join/?id={project_id}	GET

TABLE 5.3: Projects URI's

Model	URI	method
Datasets	/datasets/?id={dataset_id}	GET
Datasets	/datasets	POST
Datasets	/datasets/list/?id={dataset_id}	GET
Datasets	/datasets/grid/?id={dataset_id}	GET
Datasets	/datasets/?id={dataset_id}	DELETE

TABLE 5.4: Datasets URI's

Model	URI	method
Posts	/posts/?id={post_id}	GET
Posts	/posts	POST
Posts	/posts/?id={post_id}	UPDATE
Posts	/posts/?id={post_id}	DELETE

TABLE 5.5: Posts URI's

Model	URI	method
Plots	/plots/?id={plot_id}	GET
Plots	/plots	POST
Plots	/plots/?id={plot_id}	DELETE

TABLE 5.6: Plots URI's

# Bibliography

- [1] Brent Carlson et al. *Method of error handling in a framework*. US Patent 6,052,525. 2000.
- [2] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. "O'Reilly Media, Inc.", 2008.
- [3] Roy Fielding et al. "Hypertext transfer protocol-HTTP/1.1, 1999". In: *RFC2616* (2006).
- [4] HDF Group et al. "Hierarchical data format, version 5". In: (2014).
- [5] W3C Working Group, W3C Working Group, et al. *Web services architecture*. 2004.
- [6] Stephen W Liddle. "Model-driven software development". In: *Handbook of Conceptual Modeling*. Springer, 2011, pp. 17–54.
- [7] Michael S Mikowski and Josh C Powell. "Single page web applications". In: *B and W* (2013).
- [8] Ariel Ortiz Ramirez. "Three-tier architecture". In: *Linux Journal* 2000.75es (2000), p. 7.
- [9] IZ Schlueter. "The node package manager and registry". In: URL: <https://www.npmjs.org> ().
- [10] Robert W Shirey. "Internet security glossary, version 2". In: (2007).
- [11] Inc Sun Microsystems. "Core J2EE Patterns - Data Access Object". In: (2001-2002).
- [12] Stefan Tilkov and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.