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 28, 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

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The World Wide Web has been central to the development of the Information Age and is the primary tool billions of people use to interact on the Internet. The huge growth of Web in the last years has lead to the demand for specialized applications, which operate in different devices, are secure and guarantee the constant provision of services to the user, under any circumstances. The software development process can be delivered at the same time by multiple stakeholders, which are located in different areas of the world, studying and working on the same code. Meanwhile, the time-consuming stages of production necessitate the proper software design, with the purpose of minimizing the possibility of large design flaws or discovering them in early stages.

The model driven software development is a design technique which is based in model utilization and assists in the creation of software, which is extensible, reusable and comprehensible by different stakeholders. It contributes in the development of isolated modules with seperated responsibilities. Hence, this design approach is an ideal solution for creating carefully developed, well tested software, which can be upgraded at any moment.

Another challenge of the time we are in, is the daily touch with large amounts of complex data, which one must constantly study in order to draw conclusions and make decisions. Because of the way the human brain processes information, using charts or graphs to visualize data is easier than poring over spreadsheets or reports. Data visualization is the presentation of data in a pictorial or graphical format. It enables decision makers to see analytics presented visually, so they can grasp difficult concepts or identify new patterns.

In this thesis, we developed a data visualization framework, which follows the principles of model driven programming. The hierarchical data format is utilized for storing complex multidimensional datasets. The framework implements a method to present visually the datasets to the users, which enables easy understanding, conclusion extraction and decision making.

The framework consists of modules which contain generic code and are configurable by other modules. The framework's functionality is expanded, either through the development of a new independent module or via the growth of an existing one. The model driven approach enables the system's adaptability based on the designer's requirements.

The framework presentation is implemented through the creation of a web application, which operates as a data visualization tool, for networks of users working on the same projects. The projects contain datasets, posts and plots, which are visible by all project members. The plots are used as a graphical representation of the datasets.

Among other things, the user may interact with the datasets through the diagrams. We implemented a zoom functionality, which enables switching between the detail levels of the visualized datasets. While the abstract representation of large datasets is possible, the display of small details is also available. The retrieval of sampled chunks of HDF files makes the zoom functionality efficient.

The framework, as well as the application, are mainly developed in the javascript programming language. Especially the front end tier was developed exclusively in pure javascript, without the usage of a large framework. The front end follows the principles of the single page application design approach.

1.2 Outline

Chapter 2 provides the necessary theoretical background used throughout this thesis, including model driven software development, data access object, access control list and RESTful web service. An overview of the related work and technologies used is presented in chapter 3. The proposed detailed framework architecture is described in chapter 4. In chapter 5 we discuss the framework and application implementation as well as the technologies we utilized. Next, we evaluate the framework performance in chapter 6. Finally, the conclusion and future work are presented in chapter 7.

Chapter 2

Background

This chapter presents some background for the content of this thesis. We introduce theoretical concepts, such as model driven software development, data access object, access control list, HTTP protocol and RESTful web services.

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 a major risk of failure. According to its proponents, model-driven (MD) software development approaches are improving the way we build software. Model-driven approaches hypothetically 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 contribute to 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 nevertheless 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 [8] 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 (MDA) 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. Architectural separation of concerns is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections. In this way, architectural separation of concerns accomplishes the primary goals of MDA. Thus, developing generic, customizable, independent modules with seperated responsibilities implements the model driven software development approach.

2.2 Data Access Object

In computer software, a data access object (DAO) [15] 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 adaptor between the component and the data source.

The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. All details of storage are hidden from the rest of the application. Thus, possible changes to the persistence mechanism can be implemented by just modifying one DAO implementation while the rest of the application isn't affected. DAOs act as an intermediary between the application and the database. They move data back and forth between objects and database records.

2.3 Access Control List

An Access Control List (ACL) [14] 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.

In computing, permission is defined as the delegation of authority over a computer system. A permission allows a user to perform an action. Examples of various privileges include the ability to create a file in a directory, or to read or delete a file, access a device, or have read or write permission to a socket for communicating over the Internet. The permissions to perform certain operations are assigned to specific roles. Members or administrators (or other system users) are assigned particular roles, and through those role assignments acquire the computer permissions to perform particular computer-system functions.

2.4 HTTP

The Hypertext Transfer Protocol (HTTP) [4] 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 (URL), using the Uniform Resource Identifiers (URI) schemes http and https. URI 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. 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 accepts 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. 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 Representational State Transfer

Representational state transfer (REST) [7] 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. 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

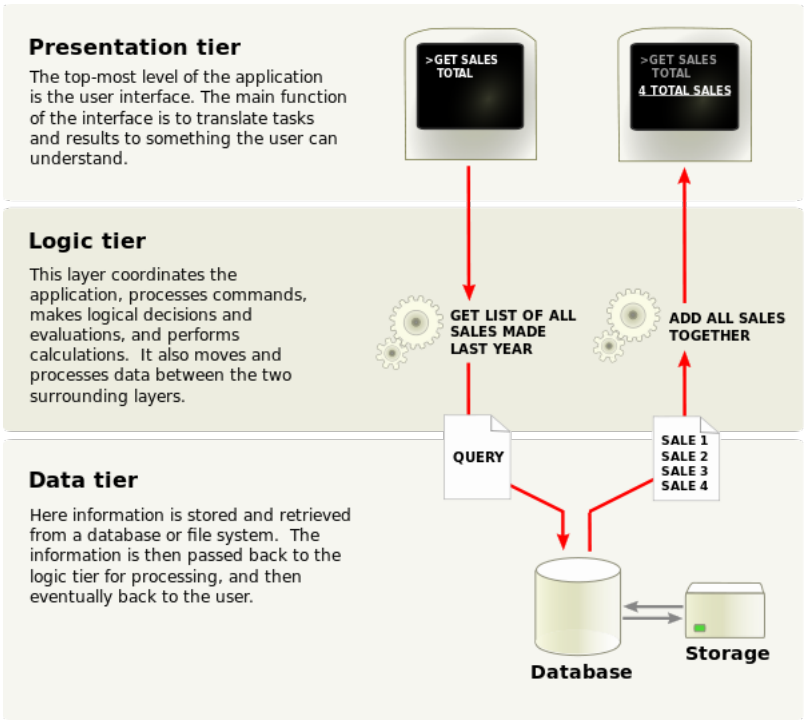


FIGURE 2.1: Three-tier Architecture

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 [12] 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.

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) [10] 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 a 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

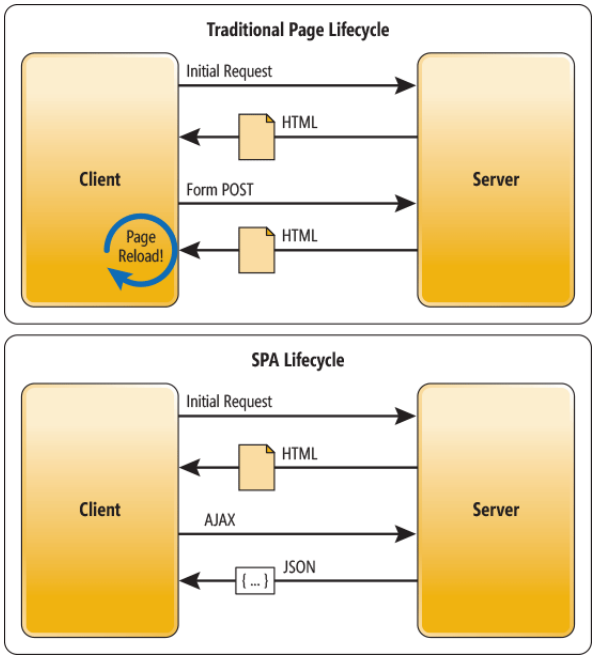


FIGURE 2.2: Traditional page lifecycle vs Single Page Application Lifecycle

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. The difference between the traditional page and the SPA lifecycle can be seen in figure 2.2. 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)) [6] 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

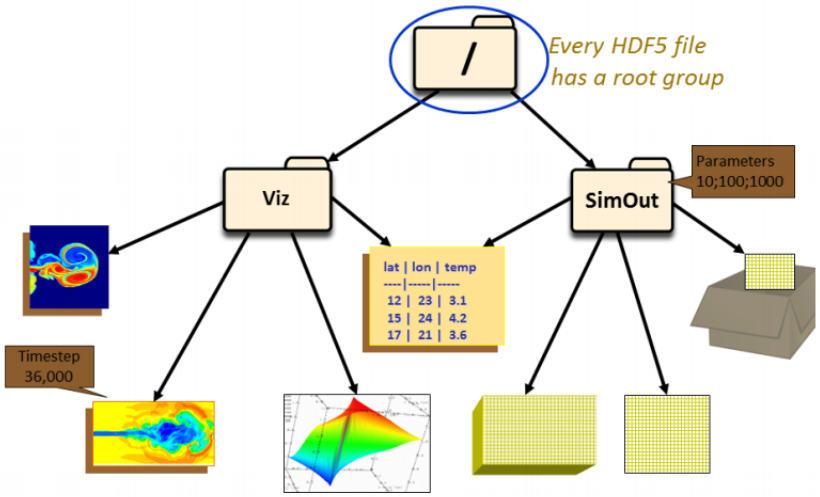


FIGURE 2.3: The contents of an HDF file

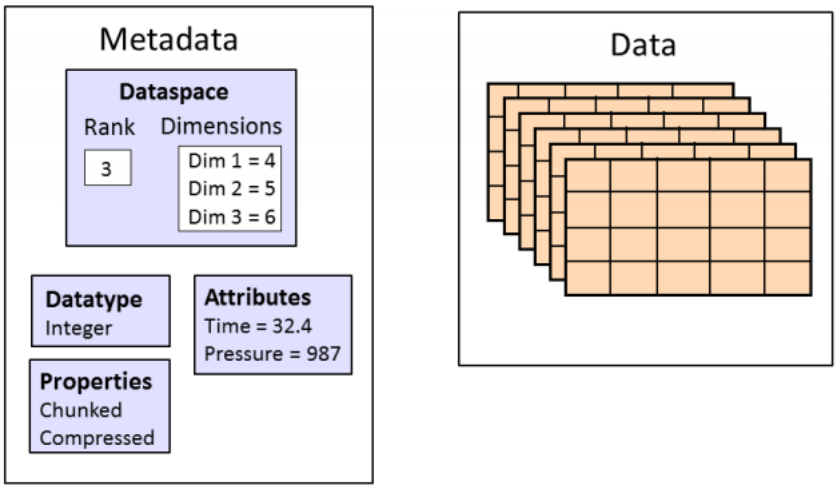


FIGURE 2.4: The general structure of a dataset

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 2.3.

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, dataspaces, 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 2.4.

Chapter 3

Related Work and Technologies Used

This chapter contains the related work and the technologies used throughout this thesis. Initially, technologies that make use of functionalities similar with the content of our framework, are described. Following, we emphasize on the technologies we used throughout the development of our system.

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. It also offers 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. Although Plotly provides a large set of functionalities, it does not offer a way to visualize large multidimensional datasets.

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 MDSD as a general practice, data access objects for the communication between database and the server, access control list for authorization and is written in Node.js.

3.2 Technologies Used

3.2.1 HTML5

HTML5 [11] is a markup language used for structuring and presenting content on the World Wide Web. It is the fifth and current major version of the HTML standard.

It was published in October 2014 by the World Wide Web Consortium (W3C) to improve the language with support for the latest multimedia, while keeping it both easily readable by humans and consistently understood by computers and devices such as web browsers, parsers, etc. HTML5 is intended to subsume not only HTML 4, but also XHTML 1 and DOM Level 2 HTML.

HTML5 includes detailed processing models to encourage more interoperable implementations; it extends, improves and rationalizes the markup available for documents, and introduces markup and application programming interfaces (APIs) for complex web applications. For the same reasons, HTML5 is also a candidate for cross-platform mobile applications, because it includes features designed with low-powered devices in mind.

3.2.2 CSS3

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language. Although most often used to set the visual style of web pages and user interfaces written in HTML and XHTML, the language can be applied to any XML document, including plain XML, SVG and XUL, and is applicable to rendering in speech, or on other media. Along with HTML and JavaScript, CSS is a

cornerstone technology used by most websites to create visually engaging webpages, user interfaces for web applications, and user interfaces for many mobile applications.

CSS is designed primarily to enable the separation of presentation and content, including aspects such as the layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple HTML pages to share formatting by specifying the relevant CSS in a separate .css file, and reduce complexity and repetition in the structural content. CSS3 [9] is the latest version of CSS and is used in this thesis.

3.2.3 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.

3.2.3.1 Node.js

Node.js [16] 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.

3.2.3.2 JSON

In computing, JavaScript Object Notation or JSON [3], is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value).

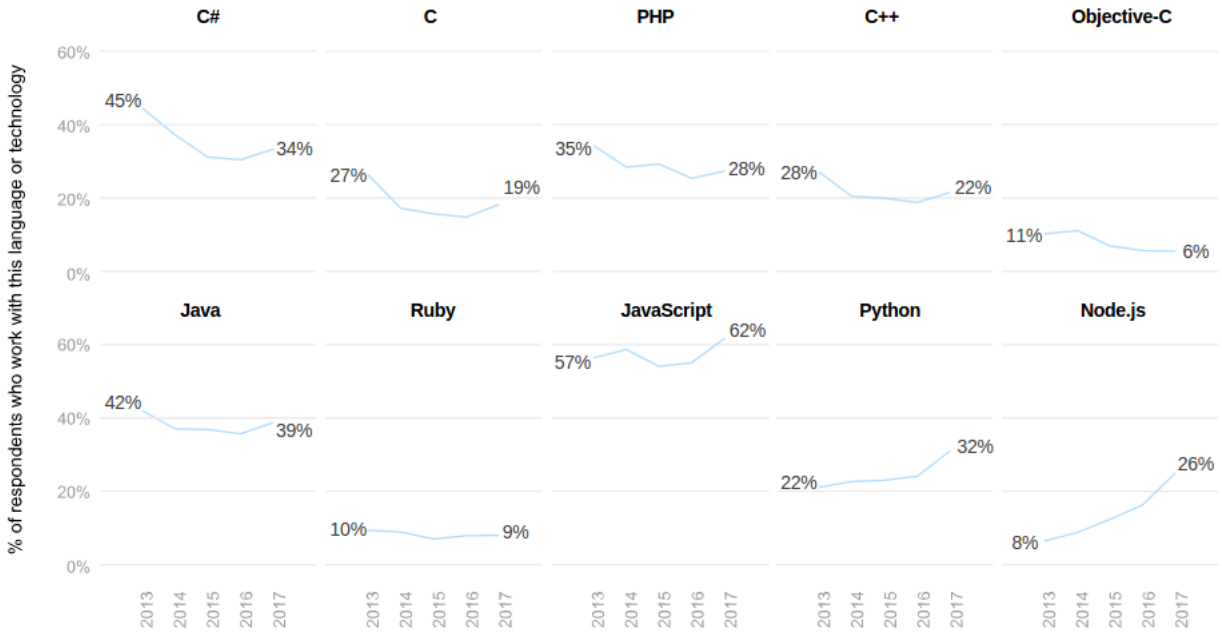


FIGURE 3.1: Developers survey for Stack Overflow website in 2017.

It is a very common data format used for asynchronous browser/server communication, including as a replacement for XML in some AJAX-style systems. JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parse JSON-format data. The official Internet media type for JSON is application/json. JSON filenames use the extension .json. JSON's basic data types are described below.

Number A signed decimal number that may contain a fractional part and may use exponential E notation, but cannot include non-numbers like NaN. The format makes no distinction between integer and floating-point. JavaScript uses a double-precision floating-point format for all its numeric values, but other languages implementing JSON may encode numbers differently.

String A string is a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.

Boolean Either of the values true or false.

Array An ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma-separated.

Object An unordered collection of name/value pairs where the names (also called keys) are strings. Since objects are intended to represent associative arrays, it is recommended, though not required, that each key is unique within an object. Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.

null An empty value, using the word null.

3.2.3.3 Ajax

Ajax (short for "asynchronous JavaScript and XML") [5] 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.

Ajax is not a single technology, but rather a group of technologies. HTML and CSS can be used in combination to mark up and style information. The DOM is accessed with JavaScript to dynamically display – and allow the user to interact with – the information presented. JavaScript and the XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads.

The term Ajax has come to represent a broad group of Web technologies that can be used to implement a Web application that communicates with a server in the background, without interfering with the current state of the page, such as HTML, CSS, DOM, XMLHttpRequest etc. The conventional model for a Web Application versus an application using Ajax can be seen in figure 3.2.

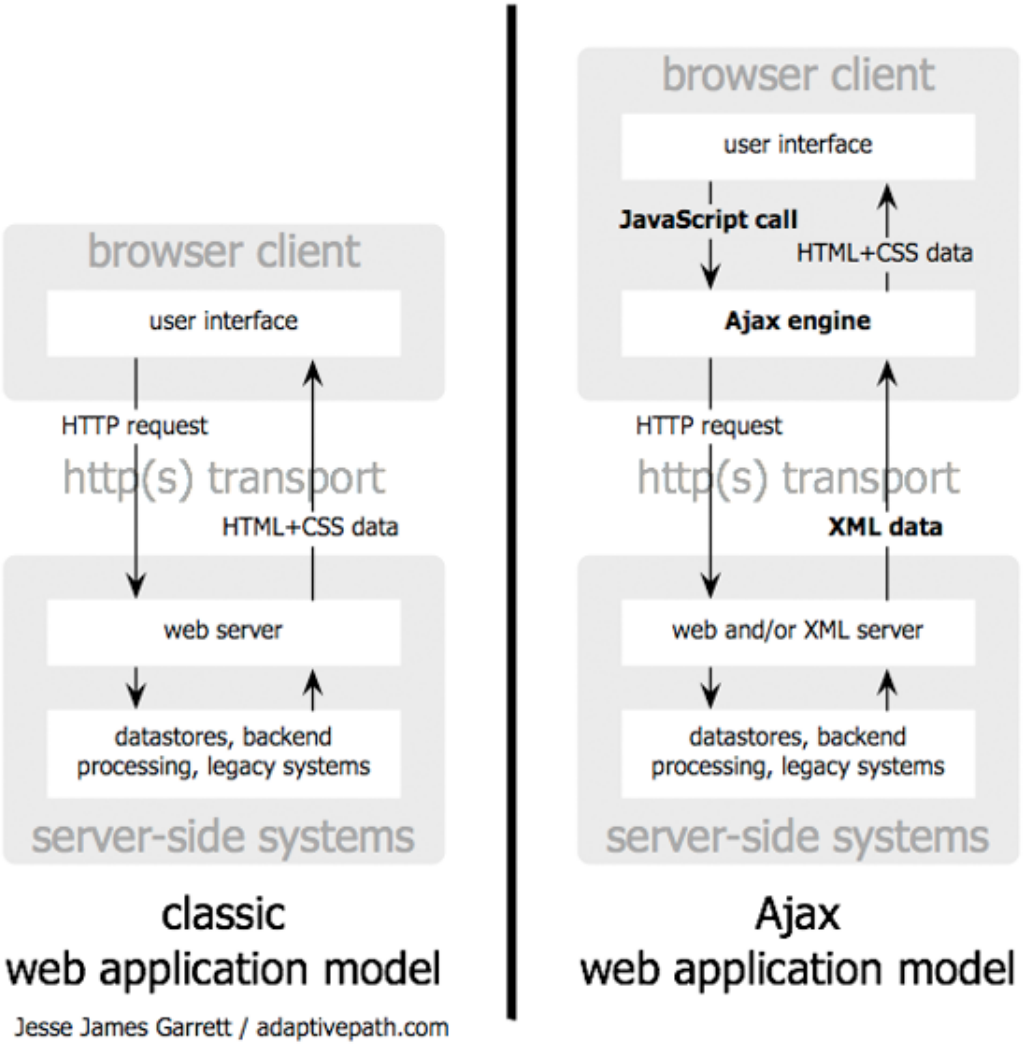


FIGURE 3.2: The traditional model for web applications (left) compared to the Ajax model (right)

3.2.3.4 Node Package Manager

Npm [13] 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 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.

3.2.4 NoSQL Database

A NoSQL (originally referring to "non SQL" or "non-relational") 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.

3.2.4.1 MongoDB

MongoDB is a free and open-source cross-platform document-oriented database program. Instead of using tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents, meaning fields can vary from document to document and data structure can be changed over time. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB. Collections contain sets of documents and function as the equivalent of relational database tables.

Like other NoSQL databases, MongoDB supports dynamic schema design, allowing the documents in a collection to have different fields and structures. The database uses a document storage and data interchange format called BSON, which provides a binary representation of JSON-like documents.

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. In chapter 4.2 we describe the basic characteristics of the application we developed. Following, we present in detail all framework modules.

4.1 Framework Design

In figure 4.1 a general framework design diagram is presented. The system architecture is divided into 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 file format. 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, 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 framework's infrastructure, aiming to present an example of a real use case. Its purpose is to create user networks, which work in common with projects and their data. The data could be text or

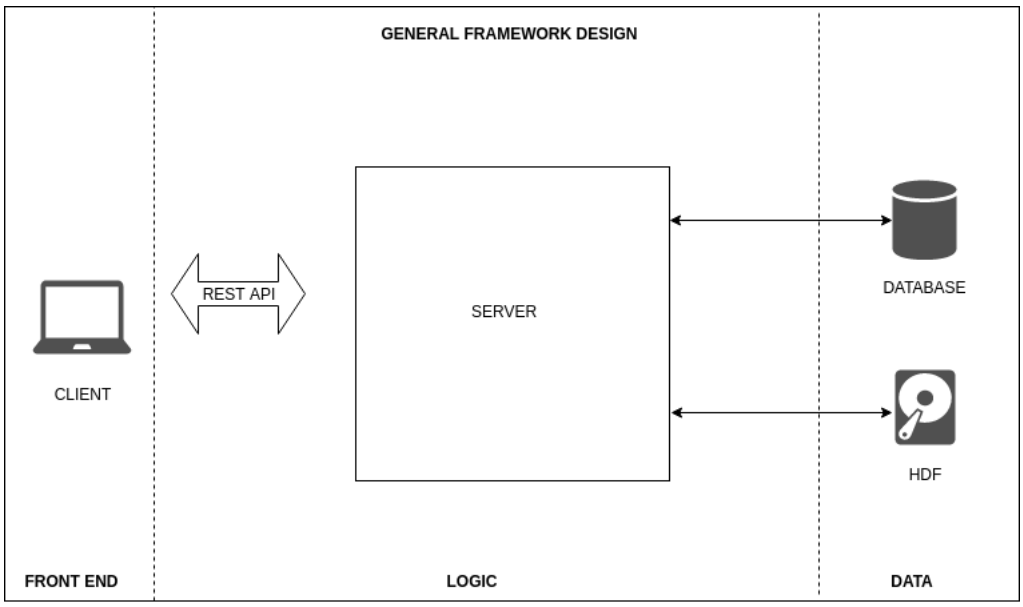


FIGURE 4.1: General Framework Design

multi-dimensional datasets. The application utilizes data visualization, aiming to a better understanding and conclusion extraction from the arithmetic datasets. The users may upload their datasets (in hdf file format), in order to make them visible to 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 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 a better understanding of the framework design. The modules are divided into three big categories: NoSQL access, main server and user interface. All modules and submodules are developed in a generic way, are highly customizable, extensible 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 constantly used in the framework implementation. Also, all modules are isolated and expose only the methods that must be used in the rest of the framework.

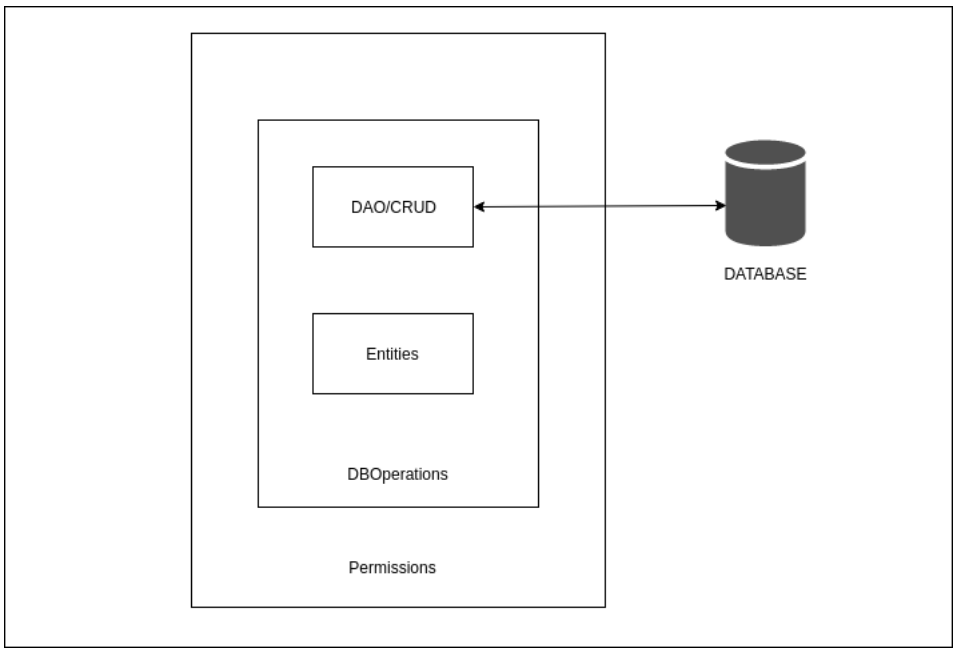


FIGURE 4.2: NoSQL access module structure

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 creation, CRUD operations, 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 that its functionality is isolated from the rest of the framework. In spite of the fact that this practice applies to the model-driven approach, it is generally a neat 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 them 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 resources to be used in the framework. More specifically, we define the structure of the entities we use, their

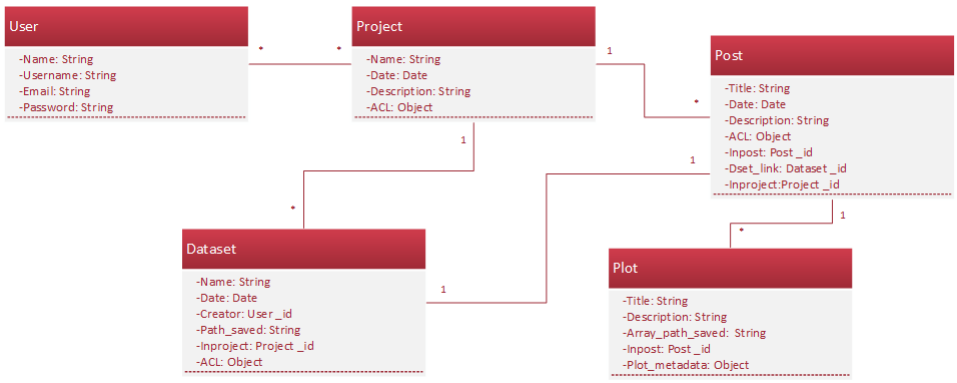


FIGURE 4.3: Back end class diagram

fields, their fields type, as well as the fields restrictions. For example, we can specify if a field must be unique in relation to the rest of the instances in the same collection of the database. The definition of the entities is developed in JSON form, therefore it becomes really easy to change or add more entities in the database. So, this module obtains the property of extensibility, which is quite important for the model-driven architecture we developed.

For the purpose of the application demo, we set a list of entities with specific fields, field types and restrictions. A visualization of this list is shown in figure 4.3. An analysis of each of the entities can be seen below.

Users In order to use the application, a user is obliged to create a new user account. The necessary properties to create a new account are a username, full name, password and email. Username and email must have unique values in relation to all other user instances. The user entity contains all necessary fields to save the above information, along with 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 concerning the users which have access to modify a specific instance of the project entity. In this way, the fields that contain this information refer to an instance of a user entity.

Datasets Another valuable entity of the application is Datasets. The application users may create their own datasets inside a project they participate.

This entity stores the metadata information of the HDF files which are uploaded on the application. The fields of this entity are the dataset name, date, an id that refers to the instance of the user who creates the entity, and the hdf filename. Additionally, saving the project id of the corresponding instance, in which the dataset is in, is required. Finally, just as in project entity, information about the access control list of the dataset, is retained.

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. Furthermore, ACL information is saved in the post entity instance. The user may create a post in response to another post. In this case, information about the post instance in reference is retained.

Plots A user may add plots inside his posts. The required fields for the creation of a plot 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. Finally, 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 to 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 it saves it. If an error occurs, it returns a new error object. Based on 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 misuse.

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 ids 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 to users 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 module 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 of the reduced database operations, since we don't 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 of a project. One way to apply for permission access, is to add all user ids 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.

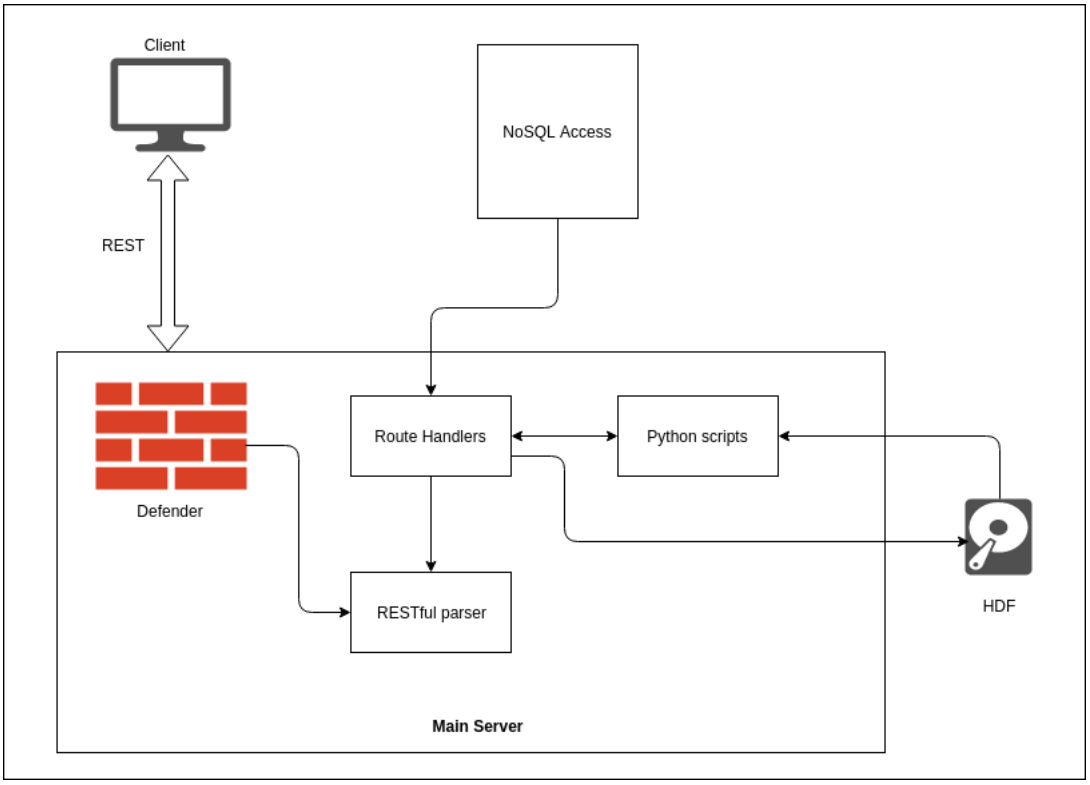


FIGURE 4.4: Main server module structure

isAllowedCreate IsAllowedCreate has the same functionality as isAllowed method, but it focuses exclusively on create operation, which has slightly different functionality than the rest of the CRUD operations.

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 4.4. To achieve that, we present all the submodules 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 contains a list of functions necessary for this model. Additionally, 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 Route Handlers

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 Route Handlers model is developed for this reason and receives the NoSQL access module as an argument. 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 2.3). The returned result of this module's methods is the one that is sent to the client.

Subsequently we present some of these functions. Most of them are developed in conjunction with the application demo, but the module is extensible 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 return to present the corresponding functions.

saveData The function saveData is used for saving an uploaded HDF file in the filesystem. The uploaded file is divided into 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 that 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 to another post. We define a parent post as a post that does not come in response to another post, and a kid post as a post that comes in response to 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 usual that some of the properties 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 remove the unnecessary properties from the corresponding objects. If an error occurs, an error object is returned.

errorHandler 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 selects a relevant 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 route handlers module are dependent on a python model, which includes a list of python modules. In this section we 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 2.3, the hdf contains multidimensional datasets that are saved in a binary format in the filesystem. But in order to make use of this data, we must first convert it from this binary format 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 the name of the HDF file as an argument. 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 it's 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 sent to declare which dimension's value is to be used.

getHDFPlot The last script is developed to return a chunk of a dataset which is 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 a 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 a chunk, the sampling frequency is high. In another case, the sampling frequency is low.

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

In route handlers module a list of functions, which use the python model, is defined. Thus, the route handlers model is dependent on the python model we described. The basic functionality of these methods 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's 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 crucial for the framework's functionality. The model is a requirement of many other modules, such as defender, session, the noSQL access module, route handlers etc.

Routes module combines different operations to achieve its own functionality. The core requirement module is the route handlers, 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 model 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 is executed 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 case that any of the above operations are not successful, the user access in the system is denied.

The authentication process, although vital for the system, comes at 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. In 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 back end 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 the communication between server and client.

The user interface module is developed in such a way, that it implements the single page application architecture (see chapter 2.2). This means that in the initialization of the communication between server and client, the client receives all the required resources for the framework operation. These include all the web complaint resources, including HTML5,CSS3 and javascript. From this point on, there is no page reload or further code execution. Furthermore, this model implements the model-driven approach, and that's the reason 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, where we mention the used technologies 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, also known as a higher-order function, is a function that is passed to another function (e.g "first") as a parameter, and the callback function is called (or executed) inside the first function. A callback function is essentially a pattern (an established solution to a common problem), and therefore, the use of a callback function is also known as a callback pattern. The execution of the first function does not block the execution of the rest of the program commands. The execution of the callback function begins, when the execution of the first function is completed. An example of node.js callback function pattern can be seen in figure 5.1.

In our framework, the callback functions are used constantly, so that the design and implementation are transformed accordingly. As a convention, all the callback functions receive as arguments two objects, the error and the result object. 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 undefined. If no errors occur, the error object is undefined and the result object contains the function outcome. Following we describe two

```
let myFunction = (cb) => {

        //do stuff

        //if error occures
        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

design patterns, which are based in callback functions and are heavily used in the framework implementation.

5.1.1 Closures

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

Generally, a closure in javascript is a function. This function receives arguments that initialize its state. The function body defines contents, such as variables, objects and other functions. Finally, the function returns, or exposes in the rest of the framework, only what is necessary. This may include also variables, objects and functions. Anything that is not returned, is defined 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 it's used in a different way. Basically, it's a function which returns another function. The interesting part is that the execution of the first function does not imply the

```
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

execution of the second. In this way we define "lazy" functions, that is methods which are scheduled to execute but don't until they have to. Thus, asynchronicity is encapsulated, aiding composability and avoiding the "callback hell" problem. This problem is caused by the generation of large sequences 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 (interoperability).

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 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 by the request method and the success of the operation (error handling is described thoroughly in chapter 5.4.3). The structure of the response object implements the JSON format.

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

The routes of the table 5.1 are used by the framework for the authentication of the users. The login route checks whether the client's credentials exist in an instance of 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

URI	method
/login	GET
/logout	GET
/isauthenticated	GET

TABLE 5.1: Authentication URI's

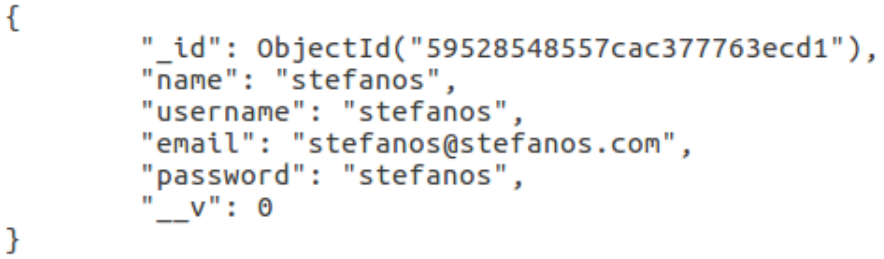


FIGURE 5.4: Example of a User instance

cookie is saved (session management is described in chapter 5.2.3). Otherwise, 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 step is skipped.

Users Table 5.2 presents 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

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

```
{
        "_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

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

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 Datasets model. A general analysis of the dataset save and retrieval management is described in chapter 4.3.2.2. 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.

Posts Table 5.5 presents 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.

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

TABLE 5.6: Plots URI's

```
{
        "_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


```
{
        "_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

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 is presented. In this chapter we mention the technologies we used and the implementation of this module.

A standard library we use in this module is Mongoose.js. This library offers a schema based solution in the modelling 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, a 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 present the general idea of the security design. Here we describe how these modules are combined to implement the security objective. The technologies we use are presented too. Initially, we introduce the passport javascript module, which has 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. Furthermore, it embodies this cookie in the respond object, which is sent to the client. This operation is called serialization. The client saves the cookie object and sends it back to the future requests. When the user logs out or a certain 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 that the authentication can be completed. If the operation succeeds, the request object URI is examined. With some exceptions, the URI's first part corresponds to one of the models 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 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 fails, 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 in chapter 2.3, 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.js 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 inversely, was developed. The rest of this section is dedicated to the way this module operates.

Node.js 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 an error, a new error object is propagated, while in the opposite case the result is parsed.

We use 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 design 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.

5.3 Front End Implementation

In this section we continue with the description of the front end part of the implementation. While initially we investigated different framework options for the client tier, we ended up using pure javascript. We considered overkill the usage of a large framework for our requirements. This way we pursued the better understanding of javascript's front end functionality. In addition, it is important to point out that the lack of context switching aids productivity, when using javascript end to end. We were satisfied with the usage of small libraries, whenever it was necessary.

A fundamental characteristic of our framework's front end is the single page application design approach, as explained in chapter 2.2. According to this architecture, all the essential code is retrieved in the front end in a single page load. Then, the communication between client and server is based entirely in ajax requests (see chapter 3.2.3.3). In order to implement this approach, we used a library called browserify. This library offers the ability to require files, as node.js uses it. Then it scans and finds all the files which use the require functionality and bundles them in one file, so it can be sent to the client.

This functionality solves many of our problems. Firstly, it's not vital anymore to write all the files inside script tags in the html files one by one, based on the dependency and execution order. But the great advantage is that a corresponding model of the back end is implemented in the front end part of the framework. Thus, a model hierarchy, which includes modules that can be combined and reused, is defined. In this way, a model driven software development design approach is adopted by the client.

5.3.1 Code organization and content

Figure 5.8 represents the general internal structure of the client files. The front end makes use of an approach corresponding to the server and the node.js implementation through the utilization of the node package manager. The package.json file contains vital information for the client network,

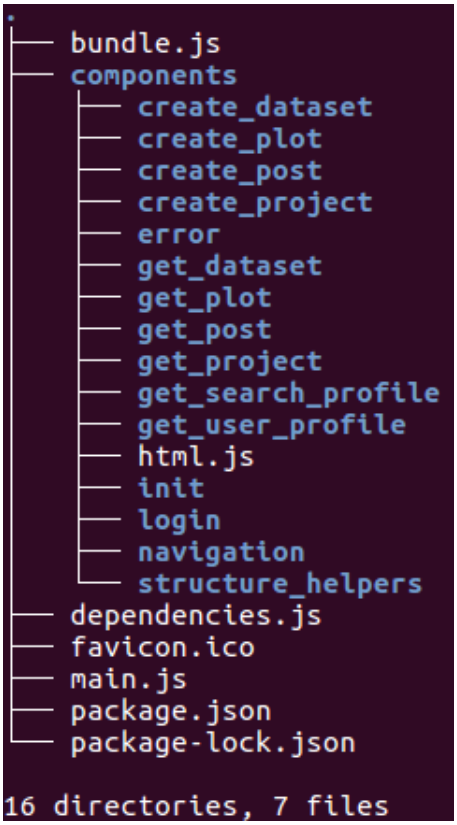


FIGURE 5.8: Client structure tree

such as library dependencies, different environment execution scripts etc. The main.js file is the first file to execute, by starting the client's initialization.

The components folder includes, among others, all the modules which are executed according to the corresponding user choices. Figure 5.9 represents one of these components. It contains submodules responsible for the creation of the appropriate elements. Event handlers modules are also included, in order to manage the functionality of elements, e.g a button. Finally, the component utilizes a module, the purpose of which is the exchange of information

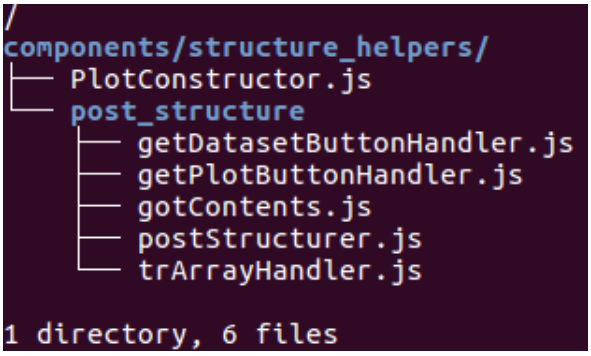


FIGURE 5.9: Post component structure tree example

between server and client, established via ajax requests.

The communication between the components is secured through the dependencies.js file. This file is received as a parameter in all the components of the client, in order to become available in any case. When the service of a component is terminated, the user interface's content is removed in order to execute another component, which is available through the dependencies.js file.

html.js This file is developed to define the front end basic operations, which are related to the html functionality. Starting from core javascript methods, such as createElement or addEventListener, we developed a wrapper adapted to our purpose. The functions create, mountTo and addListenerTo are exposed from this file to the front end and are used constantly in all components.

5.3.2 Plot generation with c3.js

Plot generation is an essential service of the developed application. In combination with the HDF functionality in the back end, we created a service, through which the users upload their datasets and can visualize them with plots. C3.js is a javascript library, wrapper of D3.js - the core of data visualization tools. This library offers the ability to create simple and minimal diagrams, and supports all simple plot types.

In order to generate a plot, the arithmetic data must be received and the corresponding configuration must be implemented. The client receives the arithmetic data and the matching metadata in JSON format. After those are parsed, the arithmetic data are prepared for plotting, via the C3 library. The metadata contain useful information, such as the plot type, by means of which we may present our data.

This library allows us to update the visual plot, which is incredibly useful in front end time saving. Thus, even before the data are received by the client, the corresponding plot presenting process has already started with empty data input. When the data arrive, the plot updates and it is not necessary to create it again. In addition, this logic offers a more smooth experience for the user.

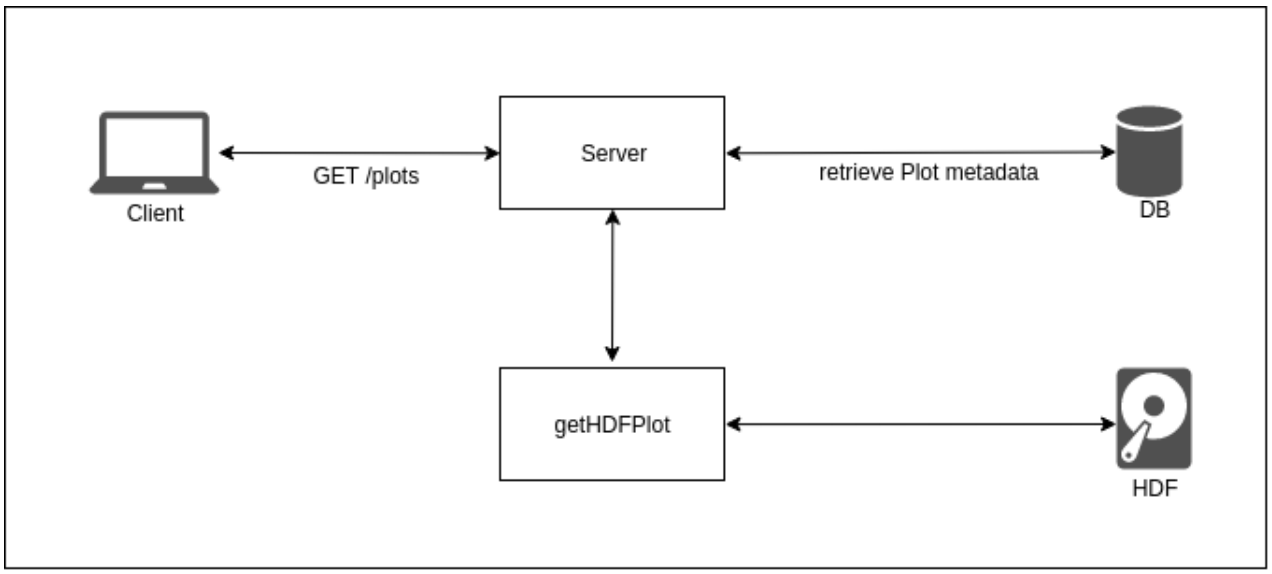


FIGURE 5.10: Plot presentation mechanism

5.4 Specific framework operations

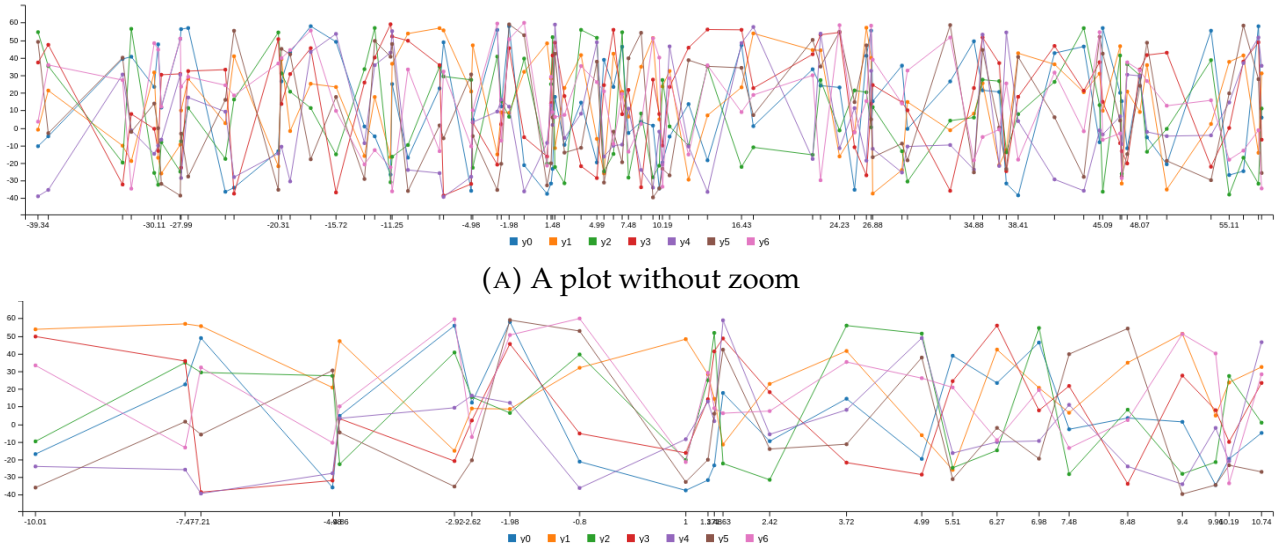
In this section we present some parts of the framework which combine the usage of both front and back end.

5.4.1 Plot presentation - Zoom functionality

In chapter 4.3.2.2 we presented the python scripts which are responsible for the HDF dataset retrieval in JSON format. But how does the plot presentation and zoom functionality operate?

Figure 5.10 presents the actions that must be made inside the framework in order to show a plot to the user. Initially, the user requests the plot display. The front end sends a request, which contains the GET method and the /plot URI, to the corresponding route, along with the specific plot ID. After it is received, the server searches the database for the plot instance and retrieves the related metadata. Following, node.js spawns a new process which executes the getHDFPlot script. This script fetches the HDF file, which contains the plot data, and returns these data in JSON format. Finally, the server is ready to send the result to the client.

To activate the zoom functionality, the user may use the drag and drop mouse technique, to select the first and last point in which he focuses. The corresponding values are sent along the next request as parameters to the server. The procedure is repeated under the condition that the sampling frequency is calculated based on the parameters distance. The plot which is



(A) A plot without zoom

(B) A plot with zoom

FIGURE 5.11: A plot before and after the zoom

presented to the user is focused in these specific points. Figure 5.11 presents the user interface result before and after the zoom.

5.4.2 Upload functionality

Figure 5.12 shows the procedure of uploading an HDF file to the framework. Firstly, the file uploads in the browser. Then, the file along with its metadata is sent to the server with a POST method, possibly in chunks of data. Next, the server checks for the file content, and then saves the HDF file in the filesystem, as explained in chapter 4.3.2.1. Finally, the file metadata with the unique new filename are saved as a Dataset instance inside the database.

5.4.3 Error Handling

A very important aspect of the framework is the way through which the system handles its errors. In general, exceptions [1] are divided in two major categories: those that are created by the client's wrong behaviour, and those that occur from poor software design, also known as bugs. The process by means of 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.

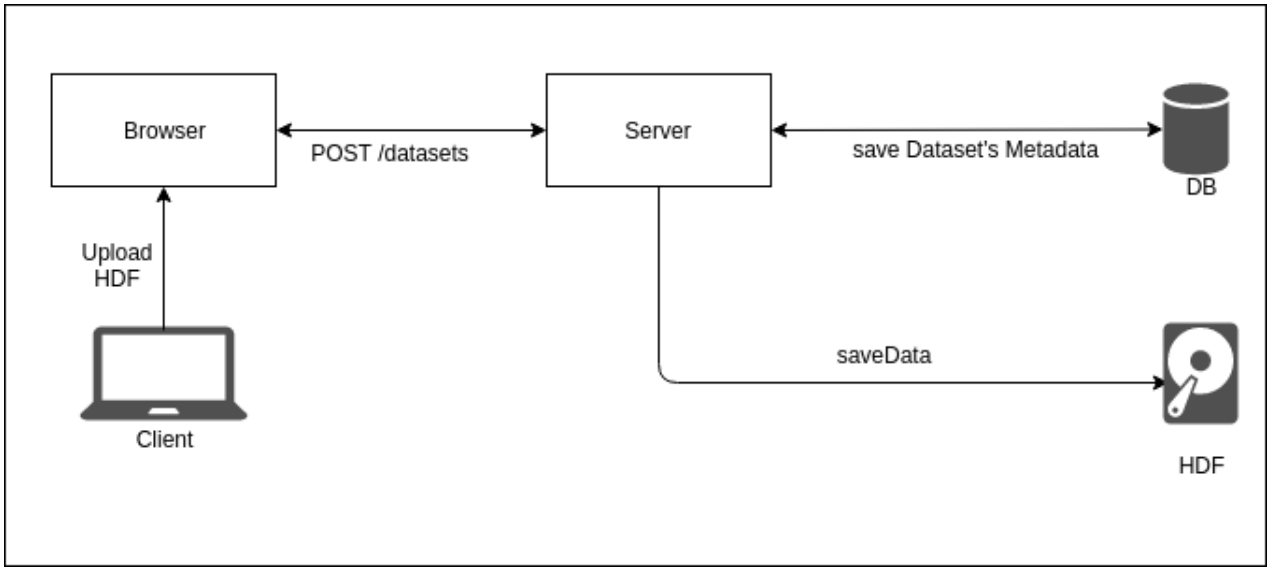


FIGURE 5.12: Upload mechanism

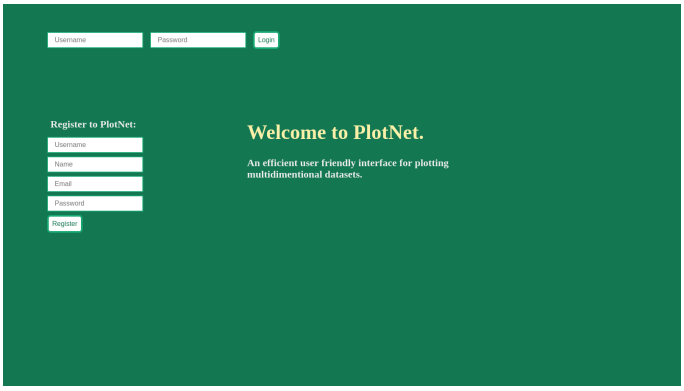
We designed the framework in a way that the handling of an exception happens in the regular flow of the code execution, and not inside a callback function. It is a bad practice to handle the error inside a callback function because the stack trace is lost. We handle the exceptions that occur in the routes and defender modules. All errors are sent with the corresponding status code to the front end.

Each time a request to the server is sent by the client, an error check is executed in the response object, in order to handle possible errors. During this check, a potential error is categorized based on the status code, and handled accordingly. In most cases a flash message shows up, but in case of a server error, the error page is presented.

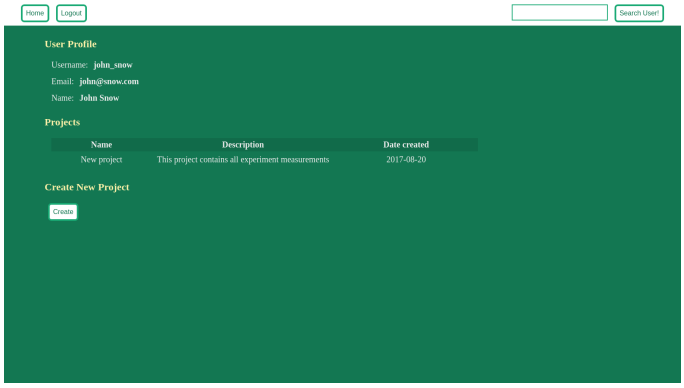
We pursued the minimization of the number of possible exceptions which occur from the client's wrong behaviour. In order to achieve that, we added in the front end specific conditions, validation restrictions etc, so that it becomes difficult to occur. Also, we implemented a similar logic in the back-end.

5.5 Application Presentation

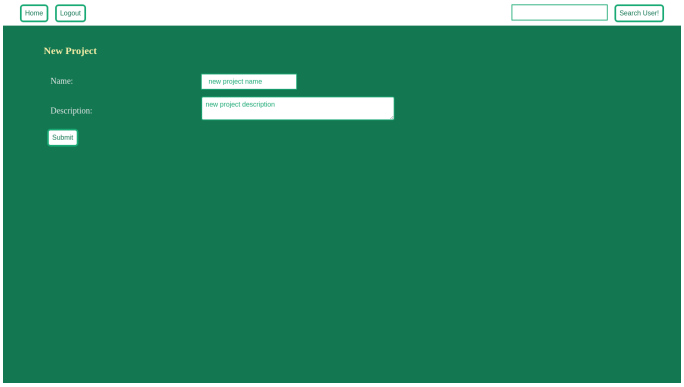
In this section we briefly present the demo application we created via our framework. Below we show some aspects of the application.



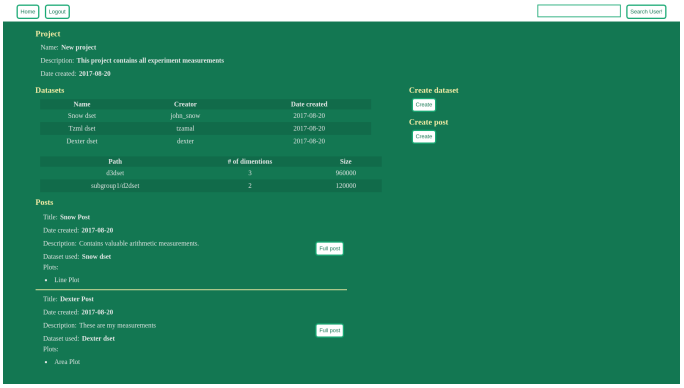
(A) Login page



(B) User home page

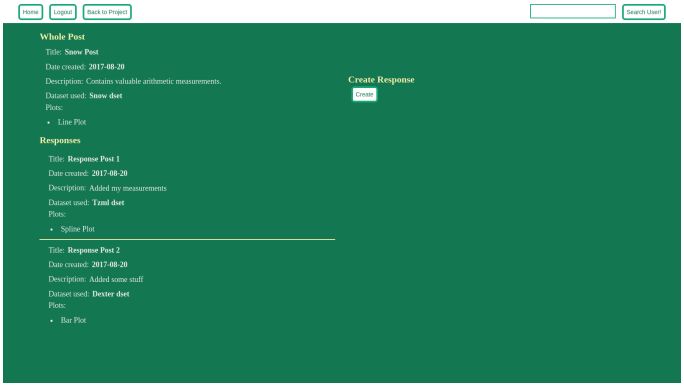


(C) Create new project page

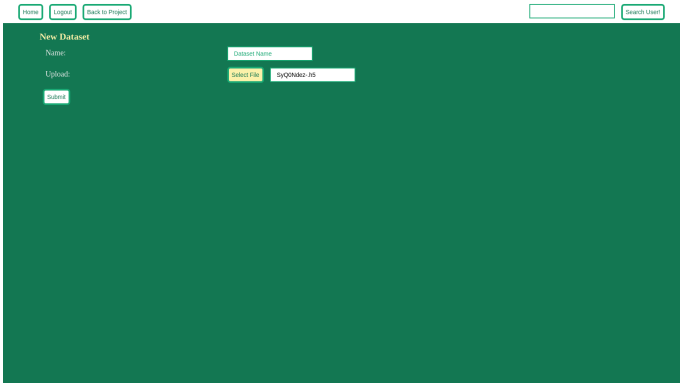


(D) Project contents page

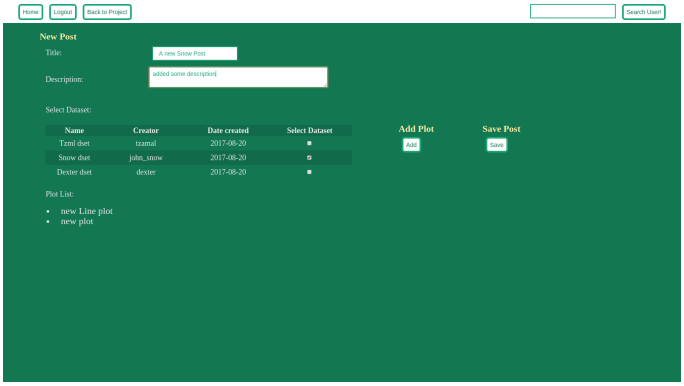
FIGURE 5.13: First pages of the application example



(A) Show whole post page



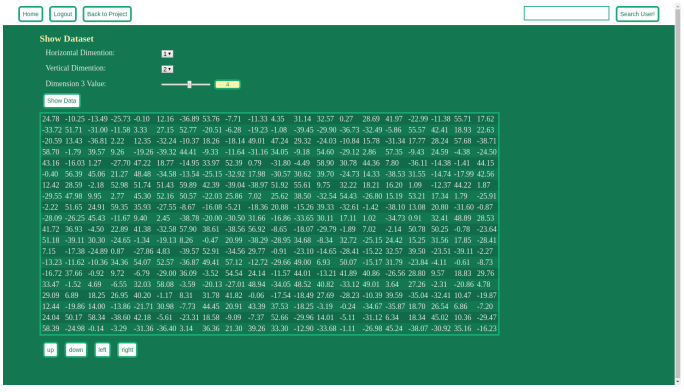
(B) Create new dataset page



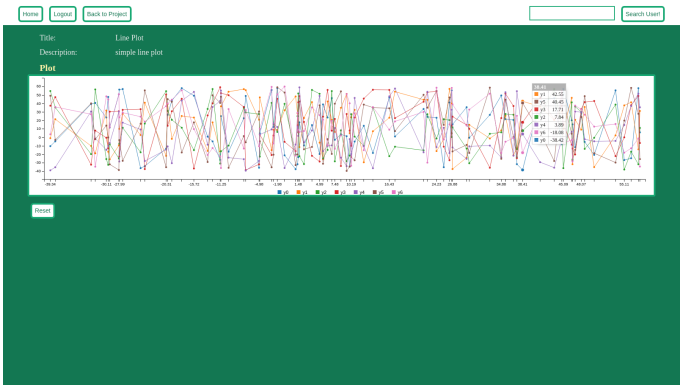
(C) Create new post page

(D) Create new plot page

FIGURE 5.14: Some more pages of the application example



(A) Dataset contents page



(B) Plot contents page

FIGURE 5.15: Last pages presenting dataset and plot contents respectively

Chapter 6

Performance Evaluation

In this chapter we describe the environment and the test scenarios we implemented in order to record the performance of our system. In table 6.1 we present the characteristics of our framework witch we use in our tests.

The capacity of the server's RAM is 8 GB. The operating system we selected for our tests is Debian 8 (Jessie) release. We used two different tools for the benchmarking of our system's performance. The first tool is the ab Apachee HTTP server benchmarking tool, that enables us to send multiple requests per second with fixed number of total requests. The second tool we used is the wrk, which allows us to send concurrent requests for a specific amount of time with timeout adjustment. With the aid of these tools we implemented two test scenarios, the results of which are presented below.

6.1 Testing Efficiency in Concurrent Requests

In the first test we selected some indicative server routes for measurement. In order to measure their performance, we select a predefined number of requests to be sent to the server, changing the number of concurrent requests in each iteration. The criterion of the server's performance is the total time

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Thread(s) per core	1
Core(s) per socket	4
Model	23
Model name	Intel(R) Xeon(R) CPU,X5450,@ 3.00GHz

TABLE 6.1: Server characteristics

for the response of all the requests. The smaller the time, the better. The results are presented in figure 6.1.

As we notice all the subfigures follow a specific pattern. Initially in all the subfigures the total time for the response of all the requests is considerably higher, compared to the following values of the diagrams. This is due to the low initial concurrency, which causes a delay until the requests are sent. That is, there is a lot of unused server-side processing power. In the case that "c=1", each request is sent after the completion of the previous sending, which explains the relatively high initial values.

Following, we observe in all the diagrams a stabilization of the total time of serving the requests, as the "c" increases. This is partially due to the adequate load the stress tool provides, so that there is no delay of the sending of the requests, but also because the server has enough time to respond without any delay.

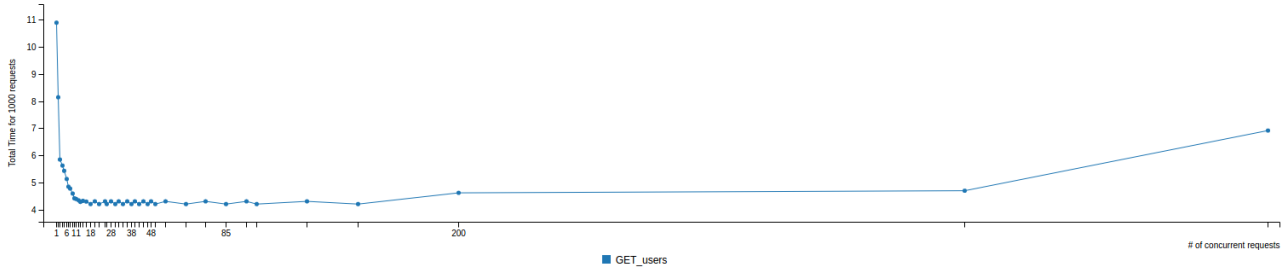
At the end of the subfigures we notice that the total time for serving the requests starts to increase gradually. The reason is that the number of concurrent requests sent by the ab tool is now too high to be served by the server, and this is the cause of the delay.

We observe significant differences between the values of the diagrams, the most important being in the routes where python is used. To execute these routes, new child processes must be spawned. As the number of the concurrent requests increases, the spawned child processes oversaturate the function of the server. This is how the higher time values in serving the requests are explained.

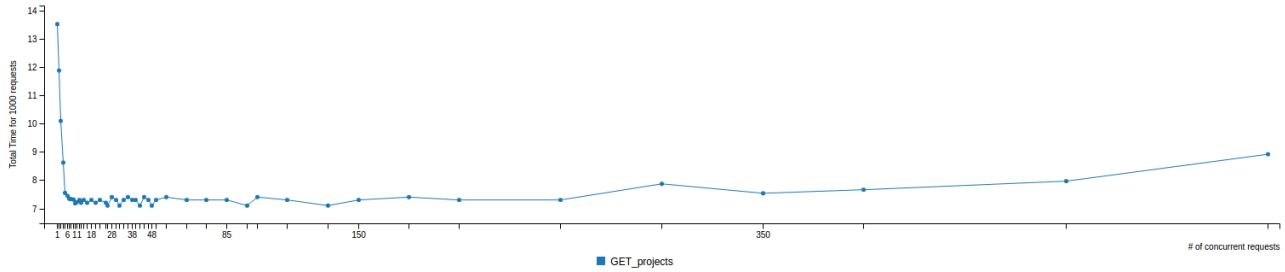
6.2 Testing Timeout Errors in Concurrent Requests

The basic characteristic of the second test scenario is the timeout value. A timeout occurs when the server fails to serve a request in a specific time period from the moment it was sent. In this test we set a relatively small timeout limit of two seconds and define each iteration in ten seconds per concurrency value. Gradually we raise the concurrency value and we calculate the number of served requests per second. The test scenario is finished when the first timeout error occurs. The results are presented in figure 6.2.

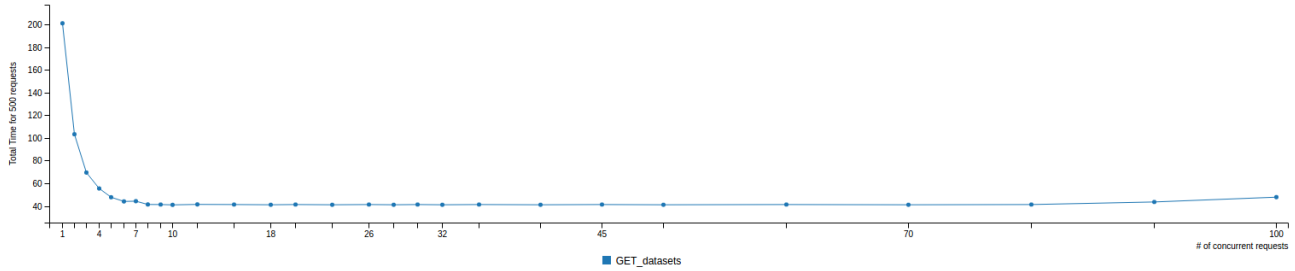
We notice that the routes that use spawned python child processes serve far less requests per second, as expected. Additionally, these routes cause far sooner timeout errors. On the contrary, the rest of the routes serve many more requests per second and delay the occurence of timeout errors.



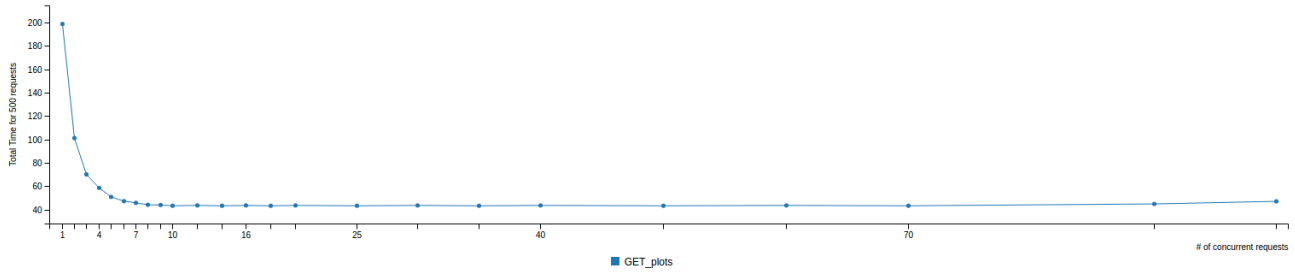
(A) GET users route experiment



(B) POST projects route experiment



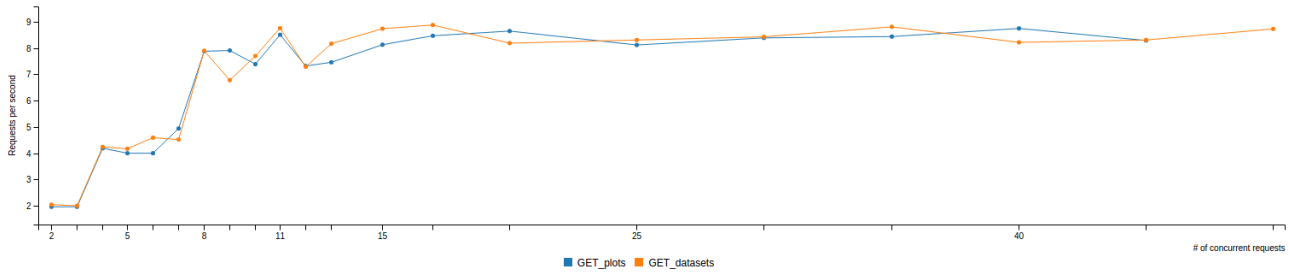
(C) GET datasets route experiment



(D) GET plots route experiment

FIGURE 6.1: Total time for all requests / number of concurrent requests figure

(A) Second experiment result



(B) Second experiment result

FIGURE 6.2: Requests per second / number of concurrent requests

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In the scope of this diploma thesis we designed and implemented a back and front end framework, globally developed in javascript, using the model driven software development approach. The framework provides entity to model mapping features, RESTful web service, data access object and CRUD operations for each model. The system utilizes a custom access control list module for user authorization. Also, the framework implements a method, which represents multidimensional HDF files in plots, in an efficient and integrated way. The front end tier of the framework utilizes the single page application design approach.

In order to present the framework, not only as an idea but also as an implemented concept, we designed and developed a demo application. The application's objective is the creation of user networks, in order to manipulate and visualize multidimensional datasets. Following we present the main advantages of our framework.

Extensibility The framework follows the principles of model driven software development architecture, so that any extra operation can be added, either within the context of an already developed service or as a new functionality, with a few lines of code. For example, an extra entity model or a route method may be easily added into the system.

No context switching The framework is developed mainly in javascript. Since only one programming language must be learned in order to evolve the current framework, the developer's productivity is increased. Especially the front end tier was developed exclusively in pure javascript from scratch, without the usage of a large framework.

7.2 Future Work

The way in which the framework is designed and implemented, offers us the potentiality of functionality extension in the future. Below we mention some possible extensions.

Server clustering A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, the user will sometimes want to launch a cluster of Node.js processes to handle the load. The node.js cluster module can be used to utilize this functionality, so that the system scales on full load.

Mongodb replication and sharding Sharding complications and replication fault tolerance will be studied in the future to optimize scalability.

Alternative methods in existing functionalities The framework functionality can be expanded in current operations. Upload file formats may be added (i.e csv files), in order to be converted and saved in the framework filesystem. Also, we can use different methods in plot sampling, such as mean values.

Image extraction for usage outside the system Right now the interaction between system users and no users is not possible. In the future, extra features, such as exporting plots in jpg format to share with users outside the application, may be added to solve this problem.

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] Douglas Crockford. "The application/json media type for javascript object notation (json)". In: (2006).

[4] Roy Fielding et al. "Hypertext transfer protocol–HTTP/1.1, 1999". In: *RFC2616* (2006).

[5] Jesse James Garrett et al. "Ajax: A new approach to web applications". In: (2005).

[6] HDF Group et al. "Hierarchical data format, version 5". In: (2014).

[7] W3C Working Group, W3C Working Group, et al. *Web services architecture*. 2004.

[8] Stephen W Liddle. "Model-driven software development". In: *Handbook of Conceptual Modeling*. Springer, 2011, pp. 17–54.

[9] David Sawyer McFarland. *CSS3: the missing manual*. " O'Reilly Media, Inc.", 2012.

[10] Michael S Mikowski and Josh C Powell. "Single page web applications". In: *B and W* (2013).

[11] Mark Pilgrim. *HTML5: Up and Running: Dive into the Future of Web Development*. " O'Reilly Media, Inc.", 2010.

[12] Ariel Ortiz Ramirez. "Three-tier architecture". In: *Linux Journal* 2000.75es (2000), p. 7.

[13] IZ Schlueter. "The node package manager and registry". In: *URL: https://www. npmjs. org* ().

[14] Robert W Shirey. "Internet security glossary, version 2". In: (2007).

[15] Inc Sun Microsystems. "Core J2EE Patterns - Data Access Object". In: (2001-2002).

[16] 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.