TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF INFORMATICS LABORATORY

Web-based Decision Policy Definition and Simulation Application for the Gorgias Argumentation Framework

DIPLOMA THESIS

of

Konstantinos Kostis

Thesis Committee:

Supervisor: Katerina Mania
TUC Associate Professor

Member: Nikolaos Spanoudakis
Researcher -TUC Laboratory Teaching Staff

Member: Michail Lagoudakis
TUC Associate Professor

MUSIC LAB
Chania, July 2019

Abstract

This thesis implements a web-based decision policy definition and simulation application for the Gorgias argumentation framework. The originality of this application lies in the use of state of the art web technologies alongside with Gorgias source code, which are combined together to deliver a user friendly and agile environment.

Moreover, the application utilizes the majority of the Gorgias-B features and the SoDA methodology and tries to hide their complexity, so that users can focus on adding the scenarios and its preferences intuitively. Furthermore, the application completely hides the argument definition process by creating default object level arguments and by implementing a custom scenario view as a table representation, having scenarios as rows and the available options for each scenario as columns.

For the front-end framework, in order to create the relevant HTML pages as well as navigation and basic functionality, the Angular 7 Typescript framework is selected for its robustness and the heavy amount of features that it offers. The cutting-edge Java Spring-Boot framework was selected as back-end framework, due to its capabilities which offer a very secure and fast server implementation which responds to the needs of every developer who wants to deploy applications to the cloud. Additionally, asynchronous scheduling technologies were used on the server, RESTful Web Services for access to the functionality from the client application, as well as NoSQL database technologies for storing and analyzing user's data. Finally, in order to have a more accurate picture of the results and the functionality of the developed application, evaluation with real users took place in two stages; one after the system design and the other after the system's implementation. During the first stage, the think aloud evaluation protocol was used, showing to users the paper prototypes created at the design process. At the latter stage, users were asked to use the developed application and interact with it. Results from both evaluation stages were taken into serious consideration and have been analyzed. Most of them have been implemented in the final version of this thesis application. After implementation, the same users were asked again and their feedback was very positive.

In conclusion, through the use of the system concerns and needs have been arise, which can be covered in future version of the application.

Keywords

Argumentation Theory, Argumentation Framework, Gorgias, Full Stack Web Development

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Concept

Argumentation is the interdisciplinary study of how conclusions can be reached through logical reasoning [3], or in other words, argumentation is a way to deal with contentious information and draw conclusions about it [4]. Argumentation is a relatively-new, fast-paced technology with applications ranging from simple games, in medical[5] area and network security[6], cognitive assistants[7] [8] business programs[1] [9]. For example some cars have an auto-pilot system integrated. Auto pilot recognizes outdoor conditions about weather, traffic info and driver's driving habits, combines them all together and adjusts car mechanics and speed according to these collected data, explaining also to driver why each choice was made. Also, modern cars can recognize possible collisions and warns driver in order to avoid it, or car takes action to avoid collision or a crash. An example of this is ABS and traction control that most cars integrate in our days. With the recent advancements in hardware and software, argumentation has entered the markets and it's expected to reach almost every household in the near future with the expansion of Internet of Things (IoT), such as personal assistants like Amazon's Alexa and Apple's Siri. Major companies are already developing argumentation frameworks, algorithms and applications in order to include them in their upcoming projects. Apparently, even small businesses and individuals can benefit from argumentation's features. There is a remarkable example [2] of an ophthalmologist in France, who mapped all of the symptoms a patient could have, with the maladies that can cause these symptoms, using argumentation software. After this mapping of symptoms and maladies that are caused from these, ophthalmologist's secretary as well as the ophthalmologist, can have a preliminary diagnosis about patient's condition. This could lead to avoidance of a possible delay of making an appointment, in cases where patient's condition is characterized as severe from the argumentation software. This is a huge boost at the productivity and the accuracy of this ophthalmologist's work. The software used by the ophthalmologist is Gorgias [1], which is a preference-based argumentation framework written in Prolog at 2003.

This thesis' goal was to implement an online application taking advantage of all Gorgias [1] [10] features emphasizing on the ease of use. Users should be able to model their real-life problems in a user-friendly design without lack of useful information giving them the opportunity to navigate to different view options. Thanks to Gorgias' underlying robustness, even users with no technical knowledge they will not face any difficulty to use it efficiently. Also, useful tips appear in each section to help even more the accomplishment of each application task. All applications stored are available in the cloud, so that they can be ready to be edited or executed in each case. Apparently, security issues regarding the Web Application are taken

into consideration so users who use the application feel secure to store their work as also their results. All of the above will, of course, be explained in greater detail in the following chapters.

1.2 Thesis Contribution

Over the last decades, argumentation has been greatly developed and it's used widely from home automation to cars, in order to make decisions and be able to explain to users why each choice was made. This thesis motivation, is to develop an online decision-making definition and simulation tool that will be comfortable for each user to use and interact with it.

After research on the internet, about other similar online tools, Spindle[11] was found. Spindle is a logic reasoner that can be used to compute the consequence of defeasible logic theories in an efficient manner. This implementation, covers both the basic defeasible logic and modal defeasible logic. Spindle also can be used as a standalone theory prover or as an embedded reasoning engine. Despite that, after extended use, we came up to the conclusion that Spindle is not handy to be used at everyday basis and without having expert knowledge of defeasible logic. Spindle's user interface can be characterized as too poor with only one text-area, where users define their arguments and defeasible facts. But, in order to define these, user has to know about defeasible logic and how-to syntax it. So, for naive users it's impossible to use this tool efficiently.

Another tool implemented is Gorgias-B [12] [1] [10]. Gorgias-B is based on Gorgias Argumentation framework which is written in Prolog. Although Gorgias-B is not implemented online, it can be characterized as more easier to use and understand than Spindle tool. Gorgias-B focuses on simplicity and ease-of use. It encapsulates all essential features of Gorgias framework, hiding from users the underlying confusing technology. So, users do not have to know about argumentation and Prolog. Gorgias-B allows users to create options, facts, beliefs and define priorities over scenarios.

Taking all these into consideration, this thesis application will focus on expanding Gorgias-B tool which is based on Gorgias framework. More specifically, all Gorgias-B main features such as option, fact and belief creation, argument definition, conflicts resolution and scenarios execution, were migrated into this thesis application as well as some new features that were introduced.

The main key feature introduced is Argue Table, where users can review their scenario preferences in a more responsive and clear way. This feature is expected to benefit users in creation of arguments and definition of option properties. Also, from the table view, users will be able to expand and refine their already created scenario preferences by adding new facts and beliefs that user would like to include into their new scenarios.

Furthermore, in Argue Table there are other new features, the Impossible scenario feature and the ability to define options in scenarios as default. When a scenario is generated from other already created scenarios, that have conflicting options between them, and this scenario supporting information is impossible to happen according to user, user is able to define this scenario as impossible and to exclude it from displaying in the Argue Table and from simulation of the scenario.

Subsequently, users can define priorities over options and especially define them as default at each particular scenario. This can be done easily in the UI with just hovering over each selected option at each scenario.

A disadvantage which occurred from using Gorgias-B, is that, due to the complexity of Gorgias framework and its quantity of features, naïve users would find it difficult to use and

understand all the features for everyday use. So, taking these into consideration, at the design process, two custom views were implemented to benefit all kind of users. To benefit naïve users, which are not familiarized with argumentation and argumentation frameworks, a Basic View is implemented. Basic view summarizes the key features that are included into this thesis application, but in a more simplistic way. Especially, in Basic view, at the option creation page as also at facts and beliefs creation page, an algorithm for natural language processing is introduced which analyzes the free-text that user has inserted into each of these forms, so users which have not enough knowledge about predicate creation and its context, to create options facts and beliefs faster.

At the other hand, in Advanced View, are included all features of Basic View plus some other features, like defining arguments for options or beliefs and resolving conflicting scenarios. Also, an option to view the generated Prolog file that will be fed to Gorgias framework in order to simulate the scenario and return the results, is implemented.

1.3 Thesis Outline

This thesis is divided into seven main chapters based on their content. The first two chapters (1, 2) include general background information and knowledge in order to better help the reader understand the subject of this thesis. Detailed development of the project follows along with how different parts of the application were put together and the final chapter (7) explains how we evaluated the system and application based on feedback from users. This chapter includes our conclusions and the suggested future work. Thesis structure is explained below in greater detail.

1.3.1 Chapter 2 - Background

Chapter begins with an introduction to Argumentation and it's definition as well as a brief description about the main frameworks that have been implemented about argumentation. An insight of Gorgias argumentation framework and Gorgias-B tool is presented. Concluding, we present the SoDA methodology which is used by Gorgias-B to define the requirements of each real-life argumentation application that will be developed by the users of the developed system.

1.3.2 Chapter 3 - Functional Specifications and UI Prototyping

This chapter begins with the definition of the functional requirements of the developed application. The main tasks of the application are described in order to clarify which are the key-features that will be implemented. After that, the personas of the users that were asked to test the application are shown. Then, the first storyboards that have been submitted through the design process of the application are briefly described and discussed. At last, paper prototypes that have been designed after the evaluation and the storyboard processes are presented and also the necessity of these in the application design process.

1.3.3 Chapter 4 - User Views

In this chapter, all user interface views that have been implemented in HTML are thoroughly described. Starting from the home page of the application, login and signup page. Then, Views are divided into two categories according to the view that are being displayed

(Basic and Advanced View). In Basic View, belong the Option, belief, fact page, Argue Table page as well as the execution/simulation Page. In advanced view, belong Argument for Options Page, Argue for Options page and Prolog File Page which displays the generated file contents that will be used from Gorgias to execute the scenario.

1.3.4 Chapter 5 - Application Design

In this chapter, the architecture, for both client and server, is presented. For the client-side of this thesis application, Angular framework is selected and for the sever-side, Spring Boot Java framework is selected. The architecture pattern used in front-end as well as in back-end is MVC (Model – View – Controller)

1.3.5 Chapter 6 - Implementation & Evaluation

This chapter contains a brief description about how each component communicates with other components. In front-end, is described how UI has been implemented using Angular's component structure which divides each page into more sections in order to be easier to manage and to maintain it. In back-end, is explained how each REST service (Scenario Service, Prolog Service, REST service, CoreNLP Service, Database Service) interacts with the developed system as well as a demonstration and explanation of algorithms developed for argument generation is provided. Lastly, the think aloud evaluation protocol is explained. There are used three people to evaluate the developed application. Each user's evaluation results are stated and analyzed.

1.3.6 Chapter 7 - Conclusions and Future Work

In this chapter, conclusions about development of application follows as well as some suggestions for potential future work that could expand and upgrade the developed application.

Chapter 2

Background

2.1 Argumentation

Argumentation theory, or argumentation, is the interdisciplinary study of how conclusions can be reached through logical reasoning. It includes the arts and sciences of civil debate, dialogue, conversation, and persuasion. It studies rules of inference, logic, and procedural rules in both artificial and real world settings.[4] Also, according to [3] Argumentation is defined as "a verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener or reader, by putting forward a constellation of propositions intended to justify (or refute) the standpoint before a rational judge". It is the field of study in which rhetoric, logic and dialectic meet.

2.1.1 Argumentation Frameworks

Over the last decades, argumentation is an important domain in the field of Artificial Intelligence (AI)and much research has been done on this area and many methodologies and frameworks have been developed to achieve reliable logic-based results. The most widespread framework is Dung's abstract argumentation. An abstract argumentation framework, as defined by Dung [13], represents the knowledge as a set of vertices, representing arguments, and the edges, representing the attacks among arguments (fig 2.1). A more complex form of argumentation is called structured argumentation, where an argument can derive from the support of another argument, hence it is common to represent a single argument as a tree.

The abstract framework of has been instantiated to several particular cases. Some of them are:

1. Abstract argumentation framework
2. Value-based argumentation framework
3. Preference-based argumentation framework

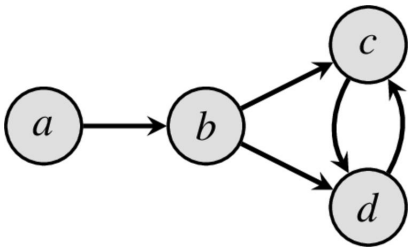

Figure 2.1: Graph representation of an argument by Dung

This senior thesis, utilizes the robustness of Preference-based argumentation and it's struc-

ture is described below.

2.1.2 Preference-Based Argumentation Framework

A variety of application problems which can be analyzed using argumentation theory, are in essense decision problems. Argumentation is best applicable when the application environment is incomplete and dynamic. This means that there is a lack of information in order to deploy a strict decision policy, so the use of argumentation can provide a way to manage all the possible alternative decisions that can be taken. The language description of argumentation is consisted in three parts:

Options

In a decision making problem, options are defined as the set of the various results. The solutions of an application problem can be characterized fully by options. Options are in fact actions, and the basic problem is to decide which action to take, e.g. to decide if a user will buy pork or chicken. In most cases, options can be explanations to classify a situation. Explanation is an intermediate step towards achieving the overall objective of deciding the course of action.

Scenario Information

Scenarios, are defined as a set of relations, needed to depict the possible states of the application environment. These relations encapsulate all the necessary information needed from the application environment when it's asked to solve particular instances of the problem.

Scenario-Based Preferences

A scenario-based preference is defined as a pair of scenarios, S, alongside with a preferred subset of options, O, that appear in this pair of scenarios S. These preferences can be described as high-level requirements in order to help find a satisfactory solution in any particular circumstance of the given problem. Scenarios, S, that appear in each preference constitute the minimal set of information about application environment to express preference among options. In human-like AI applications, such as cognitive personal assistants, these scenario-based preferences arise from personal preferences of a human user e.g. the preference for red meat, or the avoidance of chicken in the context of a shopping assistant. These scenarios can be grouped in hierarchies of increased specificity.

In order to define a decision maker's theory, information can be divided into three levels. At the first level, the rules that are defined refer directly to the subject domain, and are called Object-Level Decision Rules. At the second level, the rules that are defined are priorities over the first level rules. These priorities can be based on the specific roles that agents can assume or to generic and specific conditions. At the third level, the rules that are defined are priorities over these rules based on generic or specific contexts. Also, priorities over third level rules based on preferences between different contexts can exist.

2.1.3 Illustrative Example

Below, an example is presented[1], in order to comprehend the function of this argumentation framework. The example captures the guidelines of a human user for an on-line shopping

personal assistant. The available options of this problem are to buy, or not, various products in a supermarket.

$$OPTIONS = \{buy(lamb), buy(pork), buy(chicken), buy(fish)\} \tag{2.1}$$

The application problem is to decide which buy options to select. For easy of presentation, it is assumed that user can only buy one type of these foods when shopping. Firstly, user defines the Object-level Rules. This is defined as described below (2.2)

$$SP^1_m = \langle S^1_m = main\_shopping; O^1_m = \{buy(lamb), buy(pork), buy(chicken), buy(chicken)\}\rangle \tag{2.2}$$

This means that all the options are enabled/available under this basic scenario S^1_m. At this point, user can express preferences on these enabled options depending on further scenario information that is sufficient for a preference to be expressed.

$$SP^2_{m,c} = \langle S^2_{m,c} = S^1_m \cup cheap(pork), cheap(chicken); O^2_{m,c} = buy(pork), buy(chicken)\rangle \tag{2.3}$$

This scenario, expresses that there is a preference between buying pork or chicken if they are cheaper than the other products. At the scenarios below (2.4), (2.5), user indicates that, may have a preference amongst the cheaper options, e.g. a preference for pork in the winter and for chicken in the summer. These preferences, are refinements of the scenario $S^2_{m,c}$, in a same way that $S^2_{m,c}$ is a refinement of the initial scenario S^1_m.

$$SP^3_{m,c} = \langle S^3_{m,c,w} = S^2_{m,c} \cup winter; O^3_{m,c,w} = buy(pork)\rangle \tag{2.4}$$

$$SP^3_{m,c} = \langle S^3_{m,c,s} = S^2_{m,c} \cup summer; O^3_{m,c,s} = buy(chicken)\rangle \tag{2.5}$$

In refinements of scenarios, user is able to focus further on his preferences amongst preferred options of any parent scenario. In example below, user has a preference amongst cheaper options for the locally produced foods.

$$SP^3_{m,c,l} = \langle S^3_{m,c,l} = S^2_{m,c} \cup local(chicken); O^3_{m,c,l} = buy(chicken)\rangle \tag{2.6}$$

So, now there is a preference for chicken when it is cheap and it is produced locally. Argumentation can also support combination of scenarios, e.g. a new scenario made up of the union of two scenarios such as when the refinements of winter and local(chicken) can hold together:

$$SP^4_{m,c,w,l} = \langle S^4_{m,c,l} = S^3_{m,c,l} \cup S^3_{m,c,w} = S^2_{m,c} \cup winter \cup local(chicken); O^4_{m,c,w,l} = buy(chicken)\rangle \tag{2.7}$$

Which expresses the preference for the locally produced chicken even in the winter time, where pork is generally preferred amongst the cheap options. Lastly, user defines a scenario that takes into consideration the preference for locally produced foods that applies irrespective of the price (2.8).

$$SP^2_{m,l} = \langle S^2_{m,l} = S^1_m \cup local(chicken), local(lamb); O^2_{m,l} = buy(chicken), buy(lamb)\rangle \tag{2.8}$$

A principled approach to capture the preference requirements, or guidelines, of an application problem, would involve identifying possible combinations of the scenarios expressed directly by the user, which contain conflicting preferences, and prompting or learning from the user further preferences under such combined scenarios and their refinements.

2.2 Gorgias

Gorgias is a system based on preference-based argumentation that has been used for years by a huge variety of users for developing real-life applications. As described in [14], Gorgias is a preference-based argumentation framework written in Prolog where theories may be composed at different levels. Arguments are represented as rules which heads are either options or priorities over existing rules. When pairs of arguments derive to contrary conclusions, these pairs of arguments are named conflicting arguments. A conflicting argument is described as admissible if it counter-attacks all arguments that attack it. First level arguments (i.e. regarding options) must take priority (or higher level) arguments and be at least as strong as their counterarguments.

A useful tool named Gorgias-B [10] (fig 2.2) has been developed to support the development of applications of argumentation under Gorgias using the SoDA Methodology. Main feature of it, is that allows users with little or no knowledge of argumentation to model their applications. This tool, generates automatically source code in the form of an application argumentation theory in Gorgias framework. Conflicting Arguments that occur can be resolved using this tool, thanks to the graphical environment that guides users through each conflict and the available solutions to resolve these conflicts. Another feature is that it allows implemented scenarios to be executed. The results of the execution are presented at the user interface of the tool, as long as the admissible arguments that support them.

Although Gorgias-B tried to make argumentation more user friendly for users that have no knowledge of argumentation, it still remains difficult to understand it's operation unless they are experts in the field of argumentation or they are the developers of the tool.

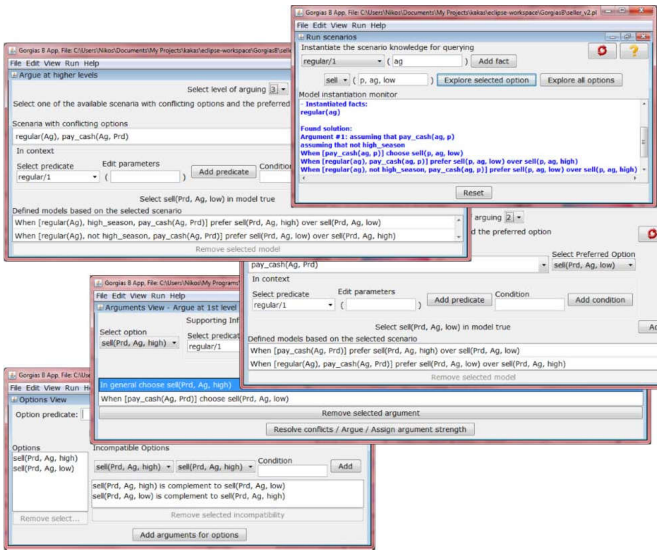

Figure 2.2: Gorgias-B Graphical User Interface (GUI) [1]

2.3 The SoDA Methodology

Software Development for Argumentation *(SoDA)* (fig 2.3) is a high level process, which requires from the developer the consideration of questions about the requirements of each problem at various scenarios. This methodology [2] is defined as a set of tasks that produce Work Products. These tasks are the definition of the different options of the application problem (T1), the identification of the knowledge needed to describe the environment (T2), the separation of information into information that exists always and circumstantial information (T3), the sort of the circumstantial information from general to more specific in levels (T4), the capture of the application requirements (T5) and the last task iteratively defines sequences of more specific scenarios and considers how options might win over others (T6).

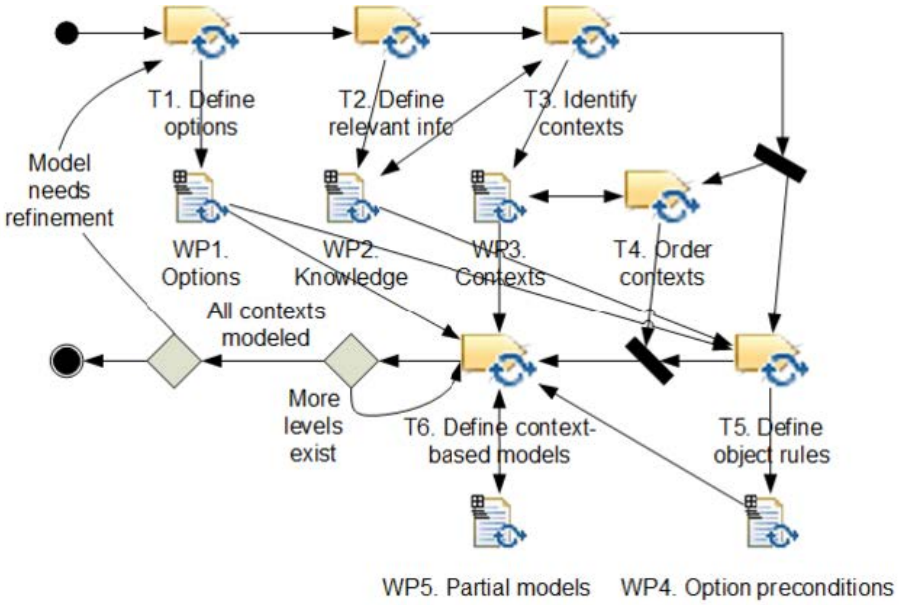

Figure 2.3: Graphical Representation of *SoDA* Methodology[2]

2.4 MVC Pattern

2.5 RESTful Web Services

Representational State Transfer (REST) is a software architectural style [15] that defines a set of constraints to be used for creating web services. REST conforms to the web standards such as using HTTP verbs and URI's. The principles that are bound to them are that resources are identified by the URIs, can have multiple representations and can be accessed/modified/created/deleted by standard HTTP methods.Also, RESTful services are bound by the principle of statelessness, which means that each request from the client to server must include all the details to understand the request.This improves visbility, reliability and scalability for requests. HTTP verbs inform the server for what action has to be done in each request. The main HTTP verbs or HTTP methods are:

1. **GET:** This method enables a user to get access to a resource. When a client click a URL in the browser, it sends a GET request to the address specified by the URL.

2. **POST:** This method is mainly used to create a resource. Multiple invocations of the POST request can create multiple resources.

3. **PUT:** This method is used to update a resource. Multiple invocations of the PUT request can update multiple resources.

4. **DELETE:** This method is used to delete a resource. When a resource is deleted, calling DELETE method multiple times, the same response will be sent to client.

HTTP Method	Resource URI	Description
GET	/library/books	This gets a list of books
GET	/library/books/isbn/12345678	This gets a book identified by ISBN "12345678"
POST	/library/books	This creates a new book order
PUT	/library/books/isbn/12345678	This updates a specific book identified by ISBN "12345678'
DELETE	/library/books/isbn/12345678	This deletes a book identified by ISBN "12345678"

Table 2.1: HTTP methods and descriptions for the actions taken on the resource with a simple example of a collection of books in a library.

2.6 Technologies

2.6.1 NodeJS

NodeJS is a software development platform (mainly for servers) built into a JavaScript environment. Its goal is to provide an easy way to create scalable online applications. Unlike most modern network deployment environments, a nodeJS process is not based on multi-threading but on an asynchronous input / output communication model. This type of operation model aims to improve the processing capabilities of web applications with many in-/ out functions, as well as real-time web applications (real-time communication programs, browser games).

Figure 2.4: NodeJS Logo

The platform architecture brings Event-driven programming to servers, enabling the development of fast JavaScript servers [16]. Event-driven programming is a model in which the

flow of a program is determined by events such as user actions (mouse clicks, push buttons), sensor outputs, or messages from other programs / threads. It is the predominant programming model used in graphical user interfaces and applications focused on executing specific actions in response to user input. In event-driven applications, there is usually a basic loop that awaits event occurrence, and then trigger a callback function when an event occurs [17].

Using the model described above, the developer can create large-scale servers without the use of multi-thread, but with the exploitation of a simplified model. The program-driven simplified model uses iterations to mark the completion of a process [16]. The Node Platform was created because parallels are difficult to implement in many server programming languages and often lead to reduced performance. Node deployment is based on Google's open-source V8 JavaScript engine [18], has excellent speed and is easy to use with HTTP, DNS, and TCP key Internet protocols. Finally, the platform platform, the JavaScript language, is so widespread that it makes it directly accessible to the entire community of web developers. Main advantages of NodeJS are:

1. Increased speed. As mentioned above, Node is a software development platform that uses the V8 engine, built by Google to incorporate Chrome into its browser. This machine translates and executes JavaScript at great speeds, mainly because of the fact that its compiler directly converts JavaScript into machine code.

2. Loop of events. The event loop is a single thread that performs asynchronously all input/output functions. Traditionally, input/output functions are performed synchronously, blocking each other, or asynchronously utilizing parallel threads. This approach tends to be overcome due to the increased memory it requires and its reputation in programming difficulty. Conversely, when a Node application requires execution of an input/output operation sends an asynchronous event loop to work, together with a re-call function and continues the normal flow of the program. Finally, when the asynchronous process is completed, the event loop returns to the process and performs its recall.

3. The fact that it was based on an already popular language, JavaScript. The most prominent client-side application development frameworks are based, if not exclusively, on the logic of JavaScript. Node is no longer required to translate the client-side logic to that of the server side as it is common. It is also not necessary to translate HTTP data sent to different objects on the server side.

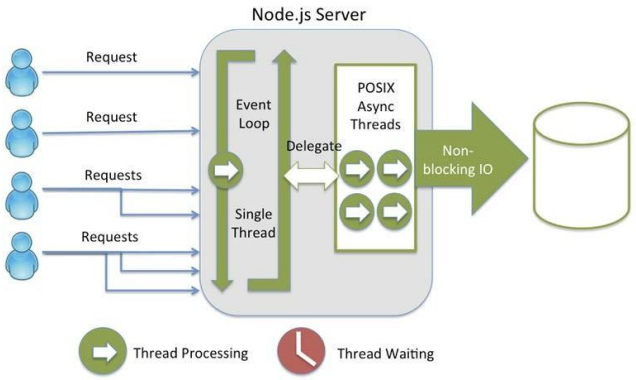

Figure 2.5: NodeJS Architecture

In this thesis, NodeJS was used for achieving robust communication between front-end and back-end. So, NodeJS using proxy can pipe requests from Angular to Spring and then return the responses from Spring to Angular.

2.6.2 Webpack

Web browsers have been designed to consume HTML, JavaScript, and CSS. The simplest way to develop is simply to write files that the browser understands directly. The problem is that this becomes unwieldy eventually. This is particularly true at development of web applications.

Webpack [19] allows the developers to treat their projects as a dependency graph. They could have an index.js in their projects that pulls in the dependencies each project needs through standard import statements. Developers can refer to their style files and other assets the same way. Webpack does all the preprocessing for them and gives them the bundles they specify through configuration. Webpack uses a config file with the default filename webpack.config.js to define loaders, plugins, etc., for a project. This declarative approach is powerful, but it is a little difficult to learn.

Figure 2.6: Webpack Logo

In this thesis, Webpack was used for packing all of the HTML and CSS pages created, also for preprocessing these pages, so the loading time in browser will be reduced to provide to users a fast and efficient navigation experience.

2.6.3 Spring Framework

The Spring Framework is an open source Java framework based on Rob Johnson's "Expert One-on-One J2EE Design and Development"[20] code designed to facilitate the development of Java applications. Spring has a modular architecture and has been divided into independent packages which, however, can work independently with other frameworks, offering their functionality with minimal adjustments. It reduces the effort and cost to develop an application, while providing facilities for the effectiveness of testing.

Figure 2.7: Spring Framework Logo

The Spring framework has about 20 sections to organize its features, some of which are shown in figure 2.8. The main services offered are:

- Inversion of Control container (IoC)
- Aspect-oriented Programming (AOP)
- Data Access
- Transaction Management
- Model-and-View Controller
- Remote Access Framework
- Batch Processing
- Authentication & Authorization
- Remote Management
- Messaging
- Testing

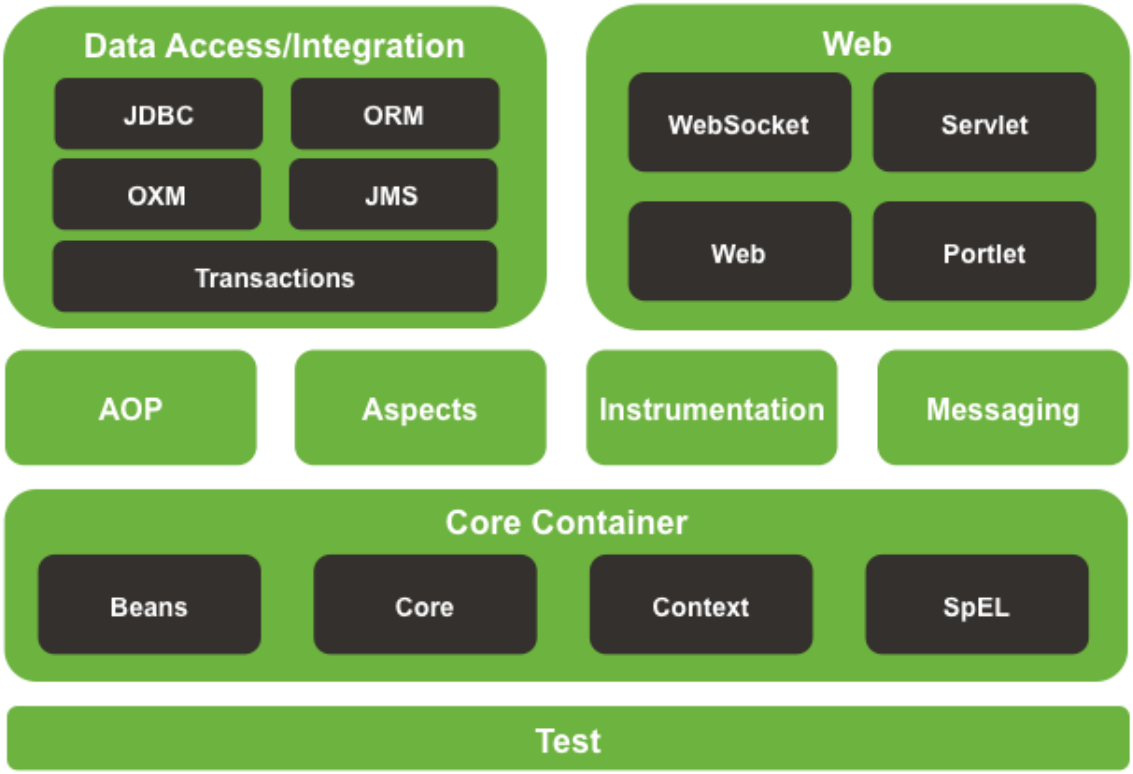

Figure 2.8: Spring's Architecture

The most important features of the Spring Framework are Inversion-of Control (IoC) and AOP (Aspect Oriented Programming). The first one is provided as a set of techniques in which the control flow is reversed in relation to the conventional method performed without the use of the framework. The framework calls procedures/methods created by the developer rather than the developer calls the framework processes/methods. The second offers a flexible solution to the implementation of critical operations, such as transaction management.

Inversion of Control Container and Dependency Injection

One of the most important features of Spring is IoC (Inversion of Control) or Dependency Injection (DI). Every major application [21] consists of classes that work together. The objects therefore undertake to acquire their own links with the objects with which they collaborate (ie to determine their dependence on them). This is a kind of pull configuration (Rod Johnson, 2005), that is, the object attempts to extract the dependencies from its environment.

Unlike DI, the dependencies are inject to the objects that need them when they are created. This task is assigned to the container, which translates the names of the objects into other objects through the constructor, methods, or "factory" methods. This approach creates a push configuration. In this way the dependencies are explicit and the classes are documented by themselves. Therefore, it is not necessary for the programmer to develop the code for the composition of a class, as this is done by the framework easily and quickly.

Spring Boot

The Spring Boot programming framework is an open source library of Java libraries that provides a particularly comprehensive infrastructure support for the development of robust applications deployed in Java EE (Enterprise) in a fast and easy way. The Spring Boot Framework was originally written by Rod Johnson and was released under Apache 2.0 license in June 2003. Spring Boot's key features are:

1. Auto-composition: The composition of the various application features is automatically done. Automatic composition allows for higher levels of security, easier development of the Spring MVC model, Java Persistence API, etc.

2. Dependency builder: Depending on the functionality the developer declares to be necessary for his work, Spring Boot undertakes to include the appropriate libraries. For example, for the implementation of an online application, the "web" launcher must be added. Similarly, if a Mongo persistence application is deployed, the mongo initiator must be added, and so on.

3. Command Line Interface (CLI): is a command line tool that is optionally used for fast deployment with the Spring framework. Allows scripts to be executed, which are approximately similar to the corresponding Java code.

4. Activator: Allows the monitor of the processes running within an application as it works. The corresponding information is presented through web endpoints or through a shell interface.

In this thesis, for the design and implementation of the server, Spring Boot Java framework was used. Also, Spring's Security features were used to reinforce application's security and robustness.

2.6.4 Angular

Angular [22] [23] is a client-side Typescript programming framework developed by Google. It belongs to the category of MVC (model-view-controller) frameworks.

The MVC model is implemented in the Angular programming framework with HTML and Typescript. The View layer uses HTML, while Model and Controller uses Typescript.

Figure 2.9: Angular's Logo

Views

A view on Angular is defined by appropriate tagging with the <**router-outlet**> tag. In a sense, the overall HTML document is the view. However, given that Angular applications are designed as single-page applications (SPA), only a portion of the page represents the current view each time. Therefore, it can be assumed that the contents of the BODY tag represent the view and the HEAD and HTML tags create the container for the individual views. An example of marking an HTML page area as a view in Angular's framework is listed in the following section of code:

```
<!DOCTYPE html>
<html>
<head>
<title>Angular Demo</title>
<meta charset="utf-8" />
</head>
<body>
<h1>Angular Demo</h1>
<router-outlet></router-outlet>
</body>
</html>
```

Figure 2.10: Angular's router-outlet implementation

In order to start using Angular [24], the development environment has to:

1. Include Node.js® and an npm package manager
2. Install the Angular CLI using

   ```
   npm install -g @angular/cli
   ```

3. Create a workspace and initial application

   ```
   ng new my-app-name
   ```

4. Serve the application

   ```
   cd my-app
   ng serve --open
   ```

Controllers

An Angular controller is a Typescript constructor that is used to enhance the functionality of the page. The controller is declared as defined below. Angular creates an object of the Controller class using this particular constructor. A controller is used to respond to user input and to interact with views to make changes to user interface. In addition, they are used to maintain the model and modify it. The following code shows a simple JavaScript controller.

```
@Component({
selector: 'app-movie-list',
templateUrl: './movie-list.component.html',
styleUrls: [ './movie-list.component.css' ],
})
```

Models

The model that corresponds to a controller includes the data to be displayed on a page, as well as the data that is collected through forms. Additionally, models can include functions that are triggered by user input or other activities, such as a button press or data change. Angular has the ngModel class, which if integrated into the controller can be used directly as a model. Based on the above, the code that follows declares and set values to model.

```
import {Component} from '@angular/core';

@Component({
selector: 'example-app',
template: `
<input [(ngModel)]="name" #ctrl="ngModel" required>

<p>Value: {{ name }}</p>
<p>Valid: {{ ctrl.valid }}</p>

<button (click)="setValue()">Set value</button>
`,
})
export class SimpleNgModelComp {
name: string = '';

setValue() { this.name = 'Nancy'; }
}
```

Based on the MVC model, Angular is consisted by some structural elements (fig 2.11) which make the application robust and easy to understand. Below we describe in more detail the structure and purpose of some of these structural elements.

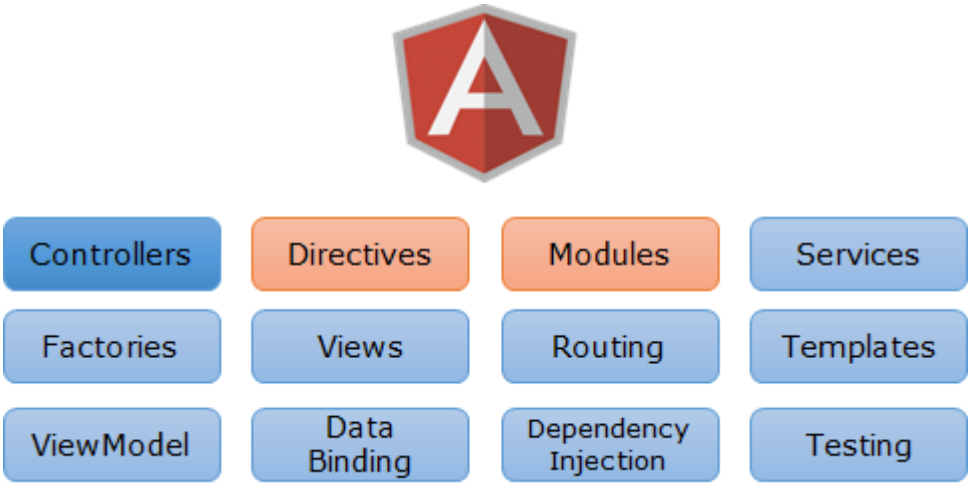

Figure 2.11: Angular's modular architecture

In this thesis, the whole design ad implementation of front-end was made with Angular 7 framework. All web pages in HTML were created using Angular component system.

2.6.5 MongoDB

MongoDB is an open source software [25] designed to store and manage document-oriented information, ie data that is semi-structured. Developed and maintained by MongoDB Inc. It can run on different platforms and belongs to the family of NoSQL databases, ie it is non-relational. Unlike the relational SQL databases, MongoDB does not have the meaning of rows, dependencies, tables, nor joins and foreign keys can be found . What we encounter is JSON type documents with flexible shapes, resulting in many applications to have a faster and easier merge of data.

Figure 2.12: MongoDB Logo

Documents & Collections

MongoDB, as mentioned above, stores JSON type documents, which are represented by keys and values. Usually documents with multiple keys and values come across. Figure 2.13 presents such a document.

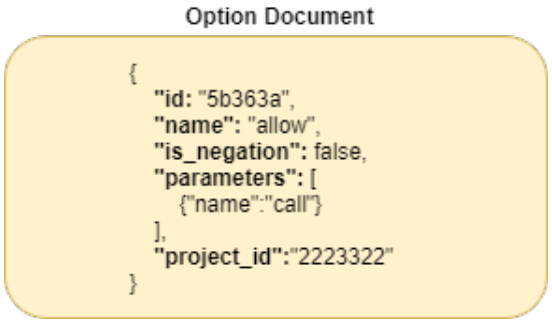

Figure 2.13: MongoDB document representation

As in relational databases we have tables, respectively in MongoDB we have collections. Documents relative to each other (eg same index documents) are stored in them. Figure 2.14 shows a collection of many documents.

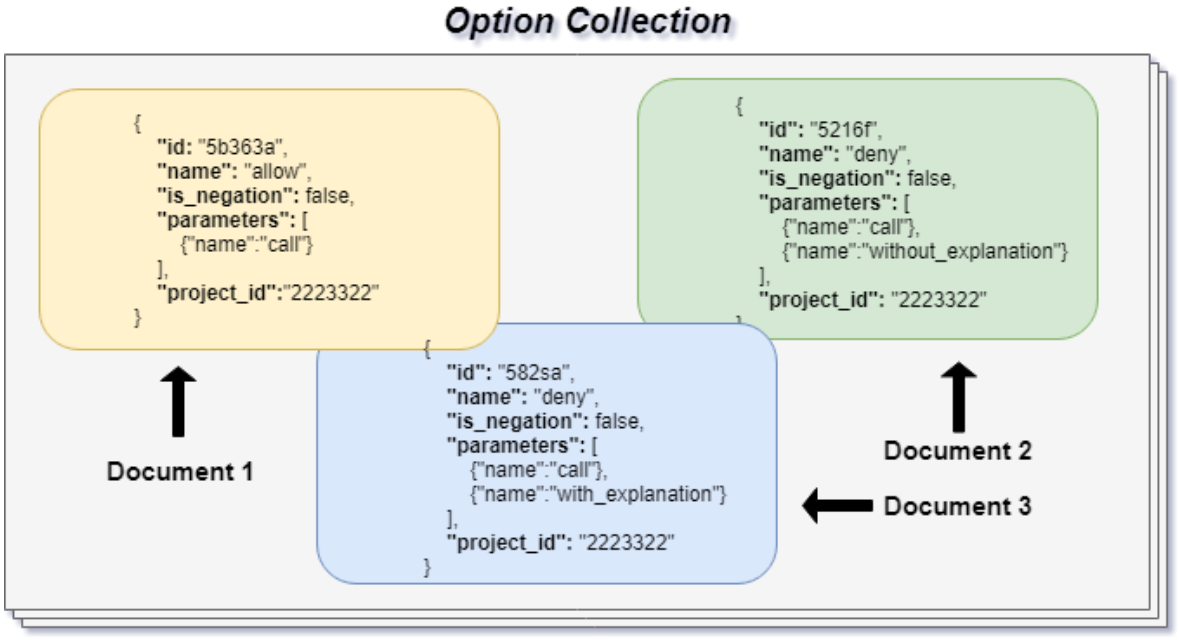

Figure 2.14: MongoDB collection representation

Collections have not definite structure. Therefore, documents may have different structure, while they are in the same collection.

Querying

```
> db.users.find({"username" : "joe", "age" : 27})
```

Indexing

To improve query performance, MongoDB uses indexes. In the absence of indexes, searching in a collection is carried out by scanning all the documents of a collection one by one.

Aggregation

Aggregations are a set of functions that allow the manipulation of data returned by MongoDB in response to a query. The merge functions group the values from the multiple documents, and by performing some functions on them, they return a compact result.

MongoDB provides three aggregation methods: the aggregation pipeline, the map-reduction function, and the simple-purpose merge functions.

In this thesis, the database selected is MongoDB due to it' s speed and structure of data that will be stored.

2.6.6 Prolog & Prolog JPL

Prolog is a logic programming language. Prolog is highly associated with artificial intelligence and computational linguistics. Unlike other programming languages, Prolog is intended primarily as a declarative programming language. Logic is expressed as relations which are called as facts and rules. It was one of the first logic programming languages implemented, and remains since today the most popular among such languages, with a number of free and commercial implementation accessible.

Figure 2.15: SWI-Prolog Logo

SWI-Prolog is a free and open-source implementation of Prolog from 1987. The development has been driven by the needs of real-world applications. SWI-Prolog is widely used in the field of education and research. A huge benefit for using SWI-Prolog, is the existence of JPL. JPL is a java interface to Prolog, which offers functions that are called in Java and that enables the communication between Java and system's installed Prolog Engine. It contains two API's, one Java API for controlling Prolog from Java, and one Prolog API, to control Java and call Java functions from Prolog. JPL is also available online as a Maven package.

In this thesis, Prolog is used in order to use the argumentation framework Gorgias, which is written in Prolog. Spring boot establishes a connection with Prolog, using SWI-Prolog's JPL plugin.

2.6.7 Docker

Docker [26] is an open platform for developing, shipping, and running applications. Docker enables the developer to separate his applications from his infrastructure so he can deliver software quickly. With Docker, it is possible to manage the infrastructure in the same ways to manage applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, developers can significantly reduce the delay between writing code and running it in production.

Figure 2.16: Docker Logo

Docker provides the ability to package and run an application in a loosely isolated environment called a container [27]. The isolation and security allow the developers to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. This means developers can run more containers on a given hardware combination than if they were using virtual machines. Developers can even run Docker containers within host machines that are actually virtual machines.

Docker Architecture

Docker uses a client-server architecture. The Docker client communicates with the Docker daemon, which is responsible for building, running, and distributing the Docker containers. Docker client and daemon can run on the same system, or there is an option to connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Docker architecture as well as differences between virtual machines and Docker are shown in figures 2.17 and 2.18.

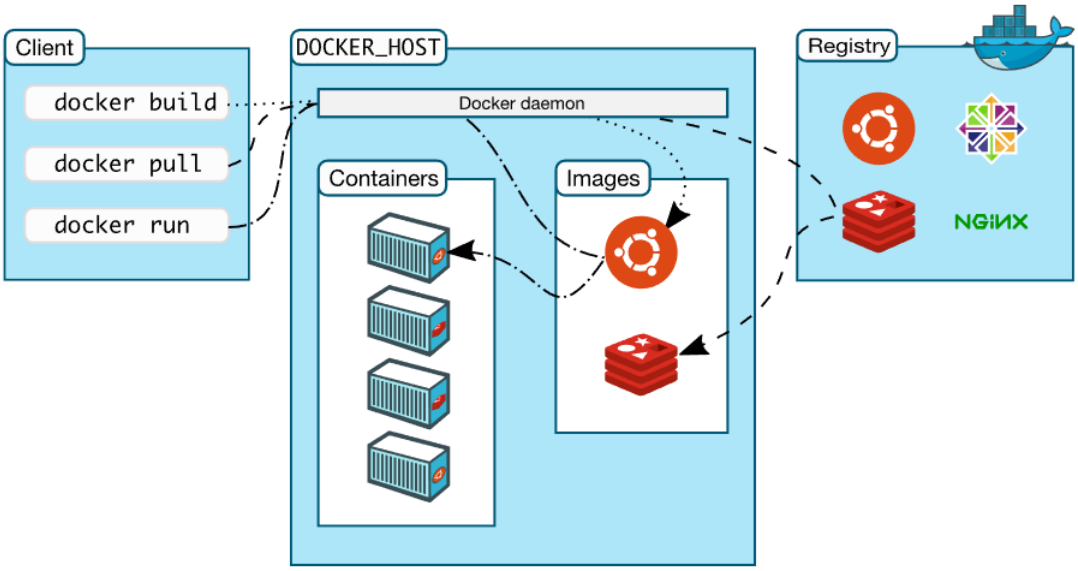

Figure 2.17: Docker Architecture

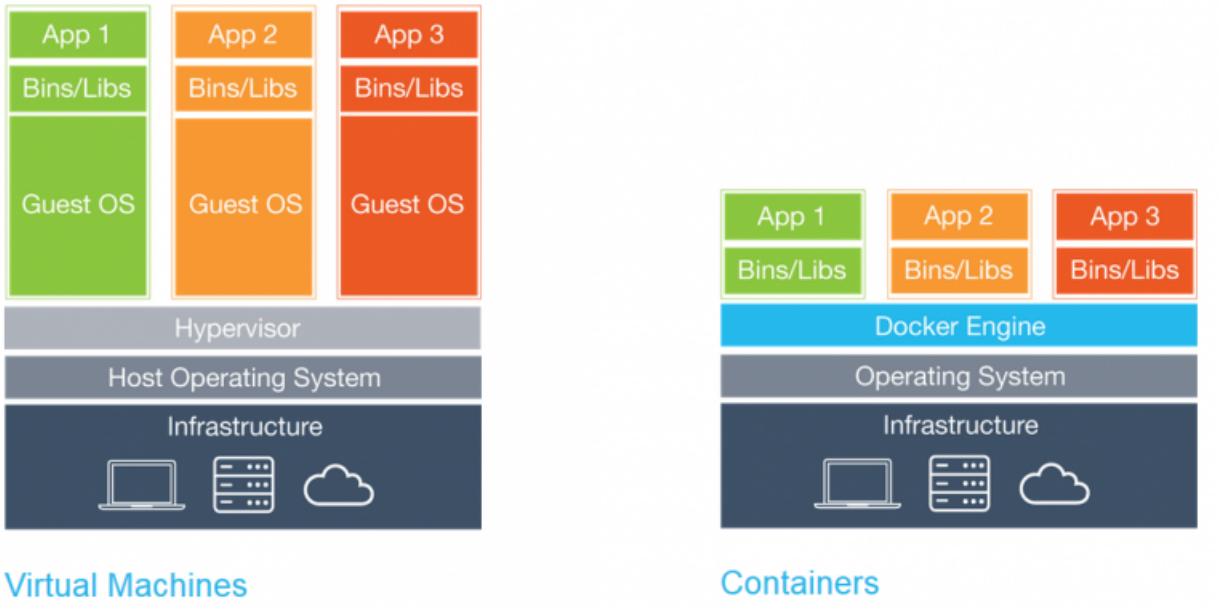

Figure 2.18: Docker Container Vs Virtual Machines

In this thesis, Docker was used, in order to deploy the whole application in the cloud. So, the developed application is packed in a container, which is independent from the host operating system that is running on server. This makes the application able to be deployed in every system that has Docker installed.

2.6.8 Development Methodology

To develop our system, the work methodology of Christodoulakis[28] is followed:

1. Paper Prototypes: Paper Prototypes are sketches in paper, that illustrate the pages that will be implemented.
2. Storyboards: Storyboards are a set of sketches that illustrate the navigation between pages and basic functionality
3. Discussion with users: Paper prototypes are given to users to interact with them and is asked from them to execute some basic functionality of the designed system.
4. Application Implementation: Taking into consideration the feedback from users that occured from discussion, the application is being implemented.
5. Evaluation: After the implementation, developed application is given to users to evaluate it. Their feedback is being analyzed and discussed.
6. Evaluation results implementation: The feedback from users, and their proposals about the developed application are being encapsulated into the application.
7. Feedback from users: At the final stage, final deliverable of the developed system, is given again to same users to test it.

Chapter 3

Functional Specifications and UI Prototyping

For the development and ultimate implementation of an online application, it is necessary to divide the software into its structural modules as well as to co-operate and coordinate between the teams that undertake to do so. Initially, the needs of the user are analyzed to give a full picture of the application's requirements and final goal. Through teamwork, functional and non-functional requirements are planned and organized, the constraints and timing of the project are set. Then, based on all of the above, designers undertake to design the user interface (UI) by continually evaluating it. Once finished, developers begin to build the application, again according to the user's requirements and needs.

In this chapter we describe the functional specifications of the Gorgias argumentation web application and the tools that will be used to implement the user interface, the functionality of the client and its servers.

3.1 Functional Requirements

The application described in this diploma thesis aims to create an online decision policy authoring tool to help develop and solve real life application problems using argumentation. Its main feature is its user-friendly and responsive graphical environment (UI). It is aimed at users of all ages with elementary even with little familiarity with computers and concepts like artificial intelligence or argumentation. For this reason, it is necessary to carefully design the user interface so that it is fully understandable and user friendly, since it supports both desktop and portable computers as well as mobile units (tablet, smartphone). The application should provide ancillary messages that guide the user, so he can understand and use the different options provided by the system.

Requirements from the interface can be summarized as follows:

- Fully responsive user interface
- Smooth application navigation
- Handy and user-friendly UI
- Simplified and accurate interface design to make application easy to be used, not only by professionals but also by users who are not familiar with such applications.

- Developed application should give the opportunity to a user to create an account to the system, by giving a unique username, his e-mail and a password. Also user should be able to manage his account, such as change his name or his password.

- A registered user should be able to create a new project, by naming it or to edit/delete an existing project.

- At each project created, a user should be able to do the below:

  - Insert the available options for each real-life problem that desires to model.

  - Insert all the available knowledge needed in order to describe the different application environments which can arise in the application problem domain.

  - Capture the application requirements. This should be done by creating the initial scenarios, accompanied with all the available options or beliefs that are triggered in each scenario instance.

  - Specify the partial models or scenarios and can consider how options/beliefs might win over other options/beliefs at each argue level.

  - Preview all the scenarios that have been created at all levels. User should be able to edit, delete, expand a scenario to make it more specific, mark an option as default, or mark the scenario as impossible to happen.

- Finally, user should be able to test each developed decision model through an execution engine using Prolog in the background. User can instantiate the scenario at execution time by adding facts and beliefs.

- Compatibility with all modern mobile phones and devices that can be connected to the internet

3.2 Personas

Personas are used to help designers and developers determine the requirements and needs of the system and proceed with the application design. In fact, these are virtual users with attributes and roles that correspond to real users who are interested in the application.

3.2.1 Antonis, 55, Professor

Mr. Antonis, is a professor in Computational Logic and Artificial Intelligence. He is also one of the co-founders of Gorgias framework, which makes him expert in argumentation. Due to his research domain in argumentation, Mr. Antonis uses Gorgias often to model and review decision policies that have been defined at each of his research projects.

3.2.2 Takis, 20, student

Takis is a student in Technical University of Crete at School of Electrical and Computer Engineering. Takis is very organized in his everyday life, thus he likes modeling his real-life tasks and problems, in a modern way, in order to make choices about them. So, he uses artificial intelligence decision assistants in an everyday basis.

3.2.3 Nikos, 35, Web Designer

Nikos, is a web designer and also an owner of a company that creates web applications and websites. He is very pressed in work, and his company employs many workers. All these, makes everyday decisions more difficult for Nikos, who wants to have the full control of his company, to track the delivery of their current projects (milestones) and also to assign tasks to his workers driven by their workload and their already assigned tasks.

3.3 Storyboards

The design of system interfaces with the user was based on the storyboards technique. These are virtual representations of the graphics on the pages of the application and represent the functionality of the system as well as the relationships between the pages. In more detail, they show the navigation on the application pages, support the designer to evaluate the interface and to add, modify, or remove functionality quickly and easily.

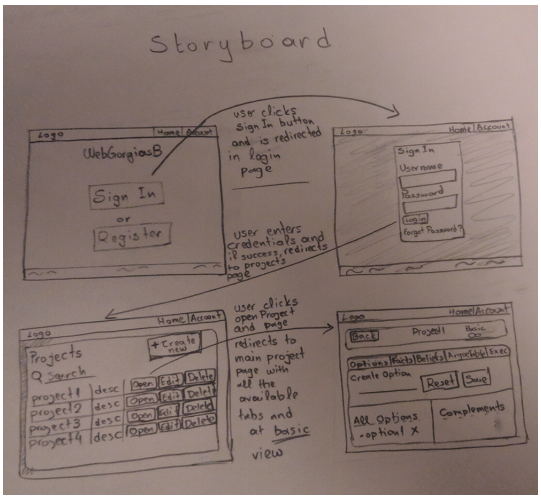

Figure 3.1: Application's Basic Storyboard. Transition from home page to login page is described in first two pages. The other pages represent transition from successful login to projects list page and then to the main page of the selected project.

3.4 Paper Prototypes

Paper prototyping [28] is a process design method that developers use to design and test user interfaces. Following the implementation of the storyboards and the evaluation that takes place in them, designers advance with paper prototypes. A paper prototype is a drawing of the user interface of the application with all the functionalities and the elements they contain, with the final result being the screens of the application.

With the help of this method, developers are able to test the interfaces at an early stage and thus save development time. This is because rescheduling of an interface due to an error or due to some new functionality is easy and fast. As is well understood, the early stage in the implementation and nature of paper prototypes, often even on paper, encourages criticism of interfaces by potential users who are studying the originals designed as prototypes. This method assists in assessing and defining requirements before even developing the application code. The following illustrations show the paper prototypes that were created before the

development of the application. They are divided into four categories, according to the view that is selected. Home Page, Basic View, Advanced View and illustrations that are common in both views.

Home Page

Below are illustrated the starting pages of the web application. In figure 3.2 is illustrated the Home Page. The homepage adopts a minimal design in order to be easier for the user to focus on the important elements of this page, such as the Login and the Register button. In figure 3.3 the Home Page when a user has already logged in or registered is illustrated. This page describes some key features of this applications and prompts the user to enter the projects menu.

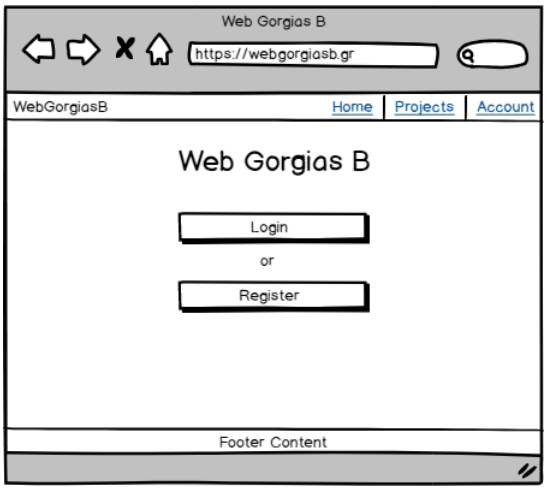

Figure 3.2: Home Page Wireframe

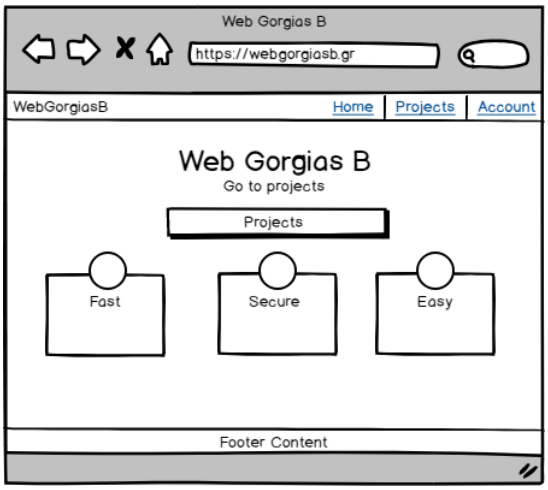

Figure 3.3: Home Page After Login Wireframe

Basic View

Application's basic view aims to help non-expert users who use the application, model their real-life problems using a clean and simple user interface with all the elements of this interface to be easy to understand. Each different functions of the application is divided into tabs. In the figures below, are illustrated the tabs and their contents that are exclusively visible at Basic View.

Figure 3.4 below, illustrates Option page, where user can create an option by expressing it in natural language. A proprietary service runs and does natural language processing at the input sentence, where the results are transformed into an appropriate form in order that the argumentation framework can manage. Also, user can review already created options and define whether different options are complements.

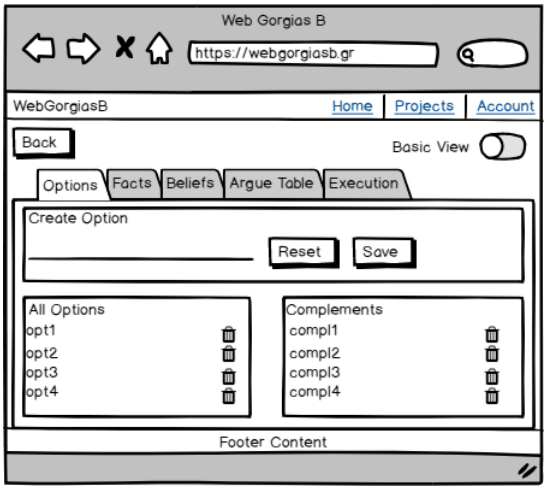

Figure 3.4: Option Page (Basic View) Wireframe

In figure 3.5 Facts Page is presented. In this page user can insert new facts to the system using natural language. Another service is responsible to handle this request, and to transform this sentence from natural language to the desired form. User can also delete the existing ones facts. Lastly, in figure 3.6, there is a creation form for Beliefs. As in previous two pages 3.4,3.5, user would has the ability to enter a desired sentence in natural language in order to be transformed, as also can delete already inserted beliefs.

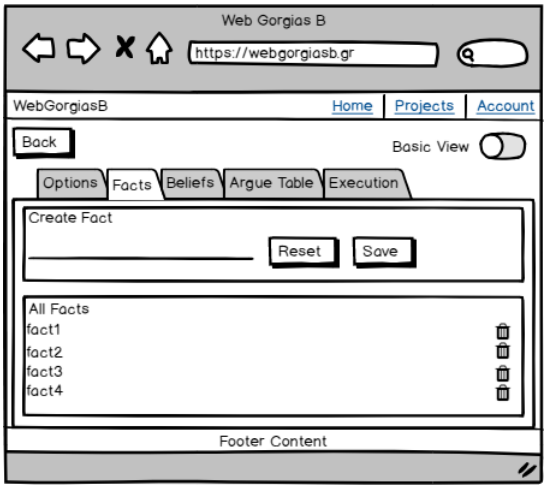

Figure 3.5: Fact Page (Basic View) Wireframe

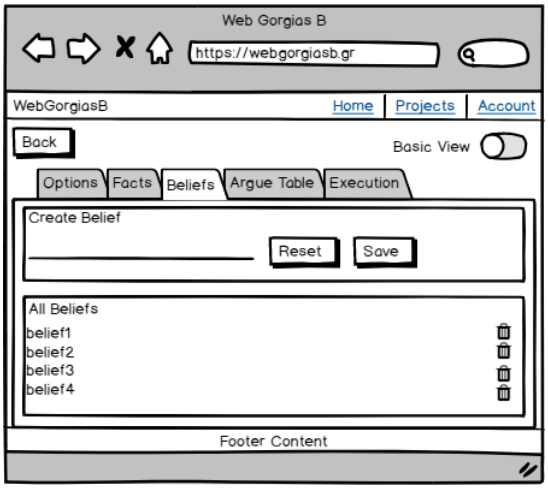

Figure 3.6: Belief Page (Basic View) Wireframe

Advanced View

Advanced view includes all the available tools that an expert user will need in order to benefit from argumentation's full potential. In the first figure 3.7, advanced user can create an option by inserting the name of the desired option and at least one (1) parameter that describe this option. A choice to add the negation of this option is also provided by system. User can also delete any of the created options. Last but not least, users can define all the created options as complements.

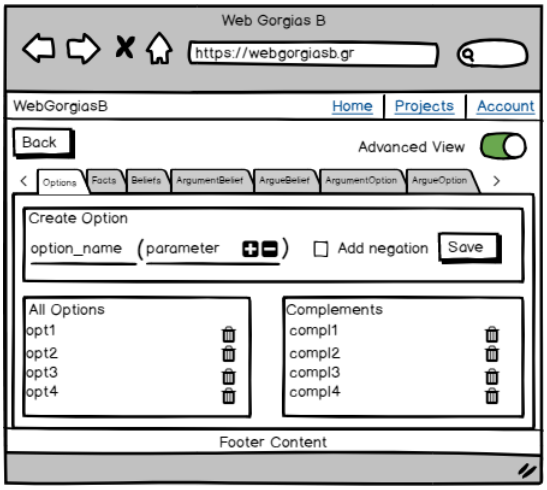

Figure 3.7: Option Page (Advanced View) Wireframe

In figures 3.8 and 3.9, advanced user can create a fact/belief by inserting the fact/belief name and optionally can insert describing parameters for each one. In Belief page, user can also mark an individual belief as Abducible. Furthermore, like in Basic View, user can review all created facts/beliefs, and there is an option to delete unnecessary facts/beliefs.

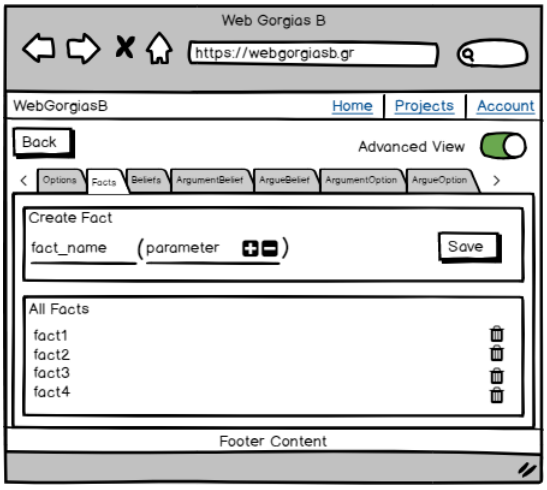

Figure 3.8: Fact Page (Advanced View) Wireframe

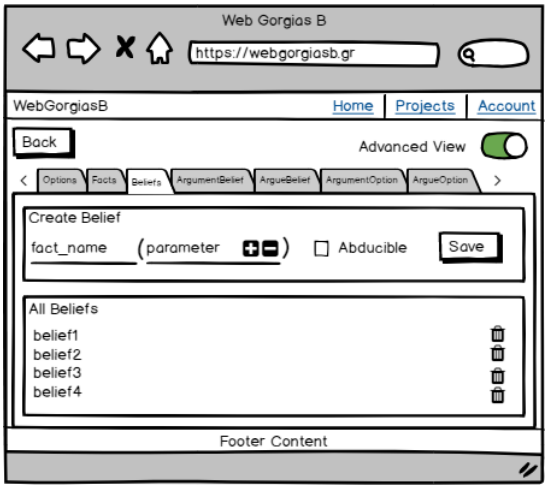

Figure 3.9: Belief Page (Advanced View) Wireframe

In figures 3.10 and 3.11 below, Argument for Beliefs and Argument for Options Page are presented. User can select from a drop-down list a preferred belief/option and then select from another drop-down list, the corresponding facts/beliefs that user desired to include, in order to define a preference over this selected belief/option. In these pages, user can also inspect already created preferences and delete them if it's necessary.

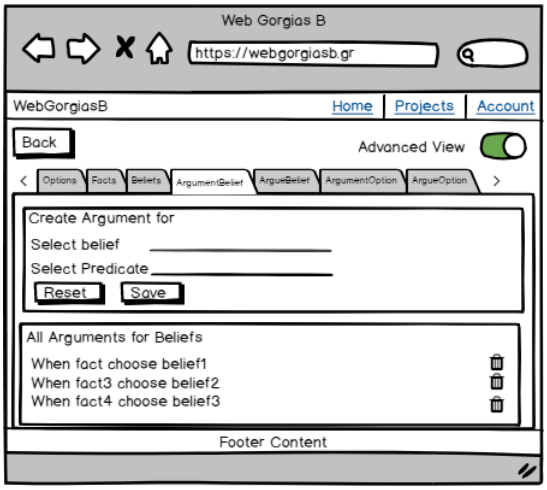

Figure 3.10: Argument for Beliefs Page (Advanced View) Wireframe

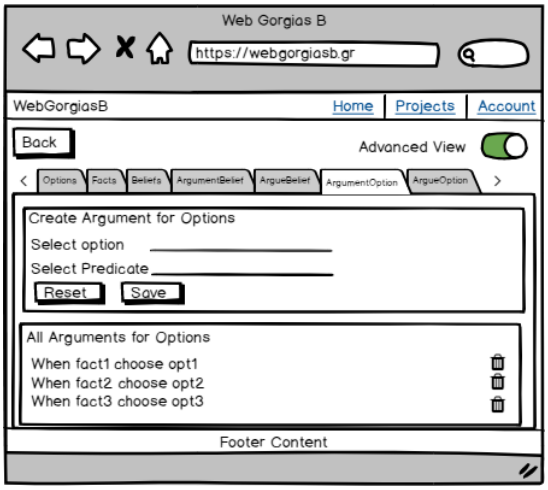

Figure 3.11: Argument for Options Page (Advanced View) Wireframe

In figures 3.12 and 3.13, Argue for Beliefs/Options pages are described which role is to define preference between options. In order to inspect or create new argue rules for an existing belief/option, firstly user has to select the argue level in which the desired rule belongs. After initializing the argue level, user has to select from a drop-down list the conflicted scenarios that have been arise from lower level rules. Subsequently, user has to select the preferred belief/option from a list and the necessary scenario information that reflects that specific argue rule. User can also review or delete the scenarios that correspond to the selected level and the conflicting scenario.

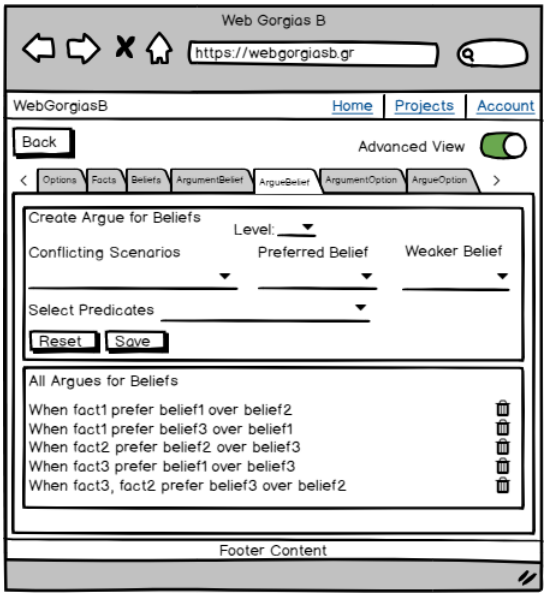

Figure 3.12: Argue for Beliefs Page (Advanced View) Wireframe

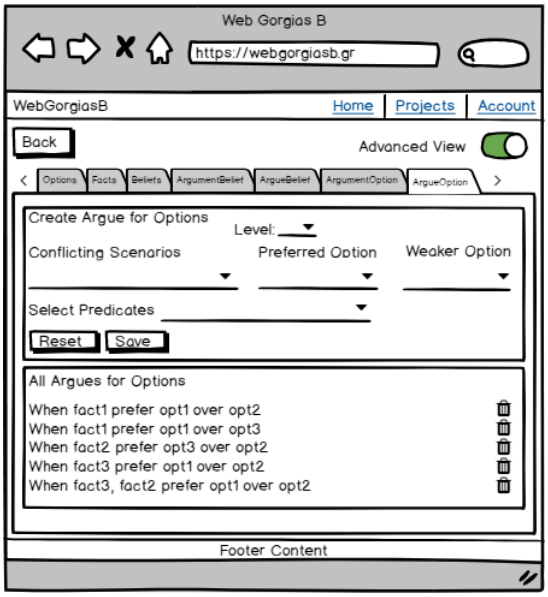

Figure 3.13: Argue for Options Page (Advanced View) Wireframe

Basic & Advanced View

Argue Table Page and Execution Page that are illustrated in figures 3.14 and 3.15 show up also in Basic and Advanced View. In Argue Table Page 3.14, all the created scenarios from all levels are listed in a table view. A check-mark visualization is used in order to indicate which of all the available options are selected in each scenario. User has the option to expand the existed scenarios with additional scenario information and more specific options. It is also available to mark a scenario as impossible if user decides that this scenario is impossible to happen or even delete one.

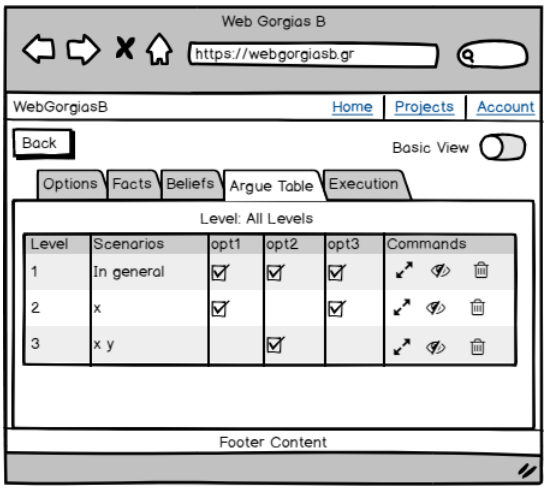

Figure 3.14: Argue Table Page (Basic & Advanced View) Wireframe

In figure 3.15 below, is illustrated the Execution page. In this page, user can test/simulate the scenarios. User can instantiate as many facts or beliefs as needed and then either search for specific options, or select the "Explore all Options" button to see which of the options can be valid. After execution process, system presents to user the execution results in a list form grouped by available options for that instantiated scenario.

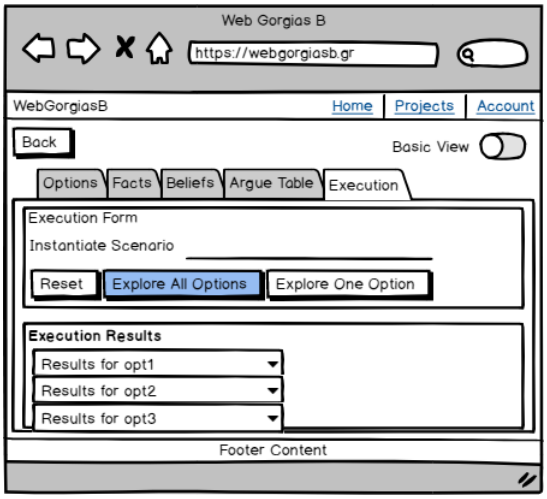

Figure 3.15: Execution Page (Basic & Advanced View) Wireframe

Chapter 4

User Views

The purpose of this senior thesis is to define and simulate decision policies for real life problems quickly and efficiently. The application was designed from its earliest stages with the design of pages optimized for user experience and usability. It is aimed at users of all ages, with good or relative familiarity with web applications.

Moreover, given the needs born through the use of smart mobile phones, application pages could not have been developed without being fully responsive and functional.

Subsequently, some of the features that have been set in order to achieve the above are analyzed.

Simplicity

The simple, fast and complete user experience in web design is one of the most difficult points facing designers and developers. The main feature of modern applications is simplicity. The user must be able to navigate the application with confidence and comfort. Thus, the goal of the implementation that has been implemented within this thesis application, is to have an experience that will be psychologically comfortable and visually relaxing. That is why the pages were designed in such a way that they are easy to understand, simple, with distinct content and few colors that do not tire or confuse the user.

Visual hierarchy

The elements of a page must be organized and positioned at the appropriate points so that once users open the application, they can first focus on the most important of them. A big role in this is played by the order in which data are displayed, their size etc.

Navigability

Another very important feature that a web application should have is easy navigation. It must be fully understood by the user, which means that the user must be able to quickly understand what needs to be filled in, when he opens the application, then which button will press, how will return to the previous page etc. Having all of the above, as a basic axis, the application pages have been configured with the help and evaluation made by users so that they behave ergonomically and correctly in both desktop and mobile devices (tablets, smartphones).

4.1 Home Page

Home page (fig. 4.1) is the first page that appears to a user. It contains two buttons that prompt the user to either log in with his credentials or to create a new account.

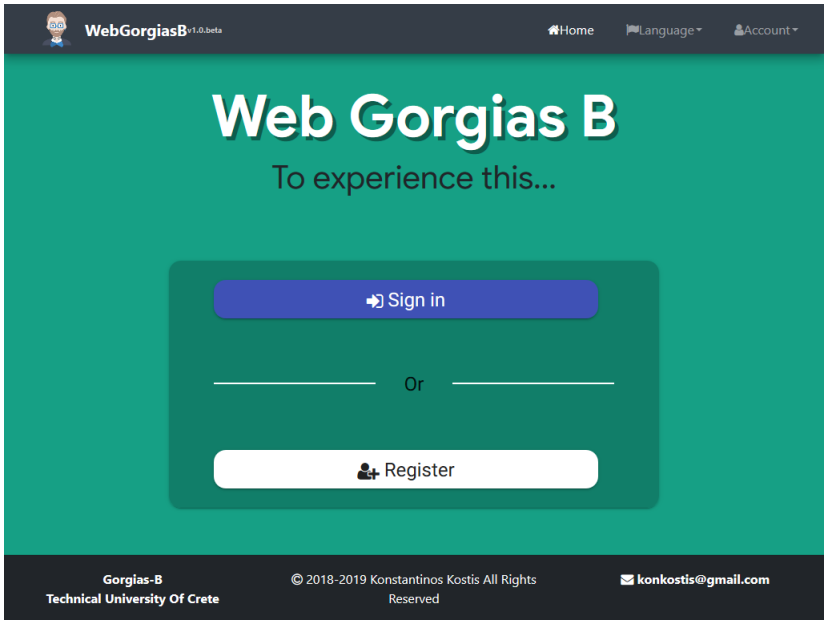

Figure 4.1: Home Page View

4.2 Registration Page

This is the page where a user creates his new account (fig. 4.2). This page contains an input form with user's account information in the fields.

More specifically, form's fields are the below:

Username Here user, inserts his username, which must be unique for each user. With this username, he will login to the system.

Email User inserts his email address which must be valid, in order to get the confirmation email to authenticate his new account.

Password & password confirmation Here user inserts his preferred and he's asked to reinsert it for confirmation. Also a meter showing how strong the password is, is displayed between password field and confirmation.

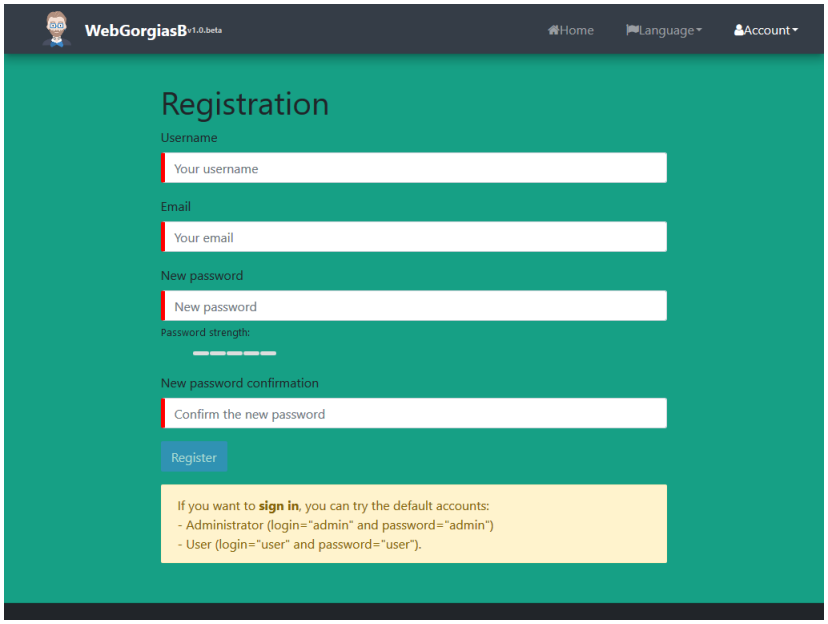

Figure 4.2: Registration Page View

4.3 Login Page

This page (fig 4.3)shows up when user clicks the login button. This page contains an input form with two fields. The credentials that are needed to enter the application. There is also an option if user forgets his password, to reset his account password after following the instructions from the sent email.

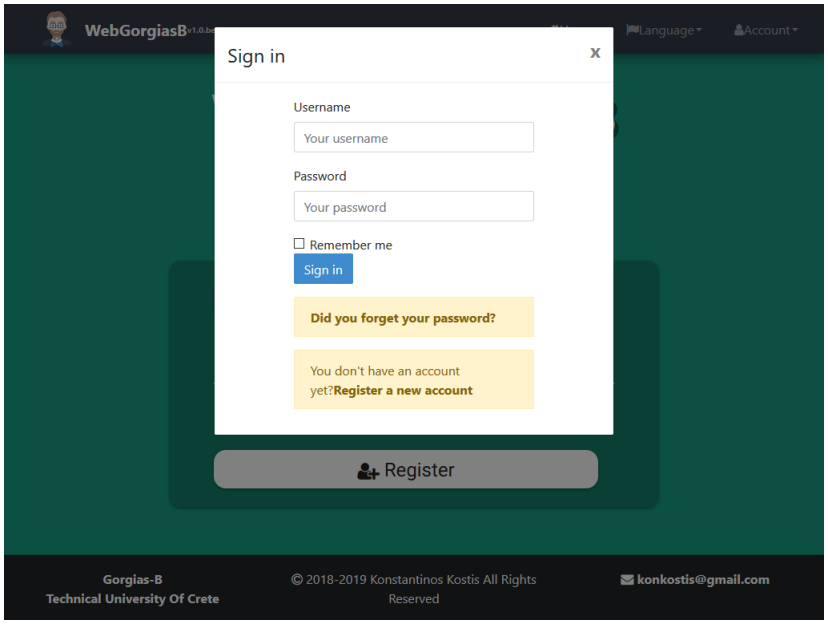

Figure 4.3: Login Page View

4.4 Projects Page

This page (fig 4.4) is the projects page. In this page, user can create a new project by clicking the "Create new Project" button which is located to the upper right corner of the page. If there are already created projects, they are all listed in a table with their corresponding info and their action buttons needed to open it, to edit it or to delete it.

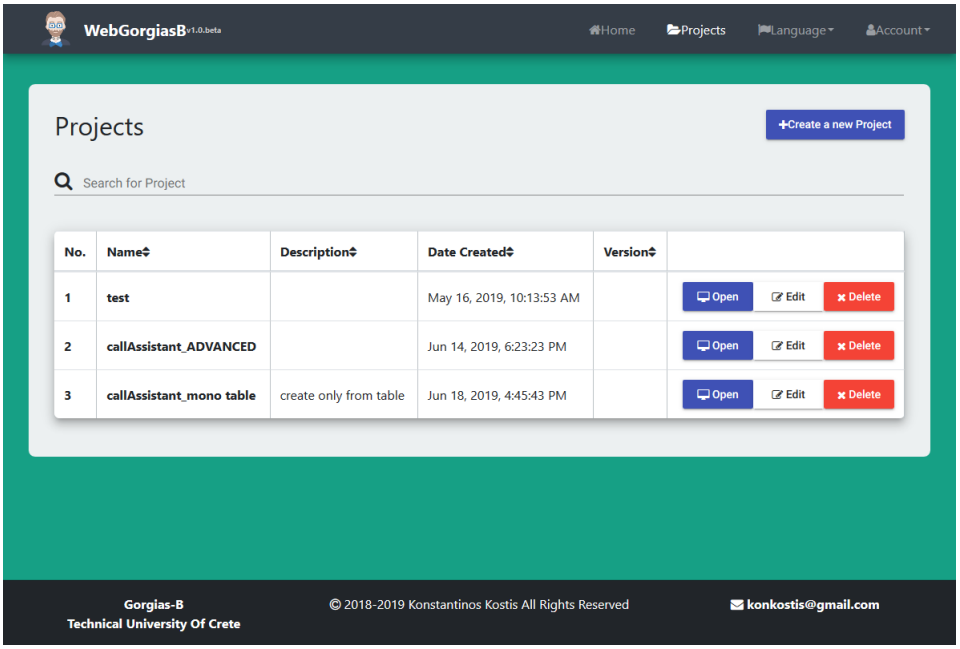

Figure 4.4: Projects Page View

4.5 Basic View

In this section are presented the views that belong to basic view, as also as the views that are common in Basic and Advanced View. The common views are Argue Table (fig. 4.8) and Execution Page (fig. 4.8).

4.5.1 Options Page

The Option page (fig. 4.5) can be divided into three parts. In the first part, there is the input form to create a new option. This contains only one field which user fills it with free text and the option is generated automatically after natural language processing in the background. The second part of page, is the list of all created options, where can review or delete them. In the last part, there is the complement section. In this section, user can define from the available options, which of them are complements and add them at the complements list that there is below or can click the button "Generate Complements" that automatically generated complements for all options defined.

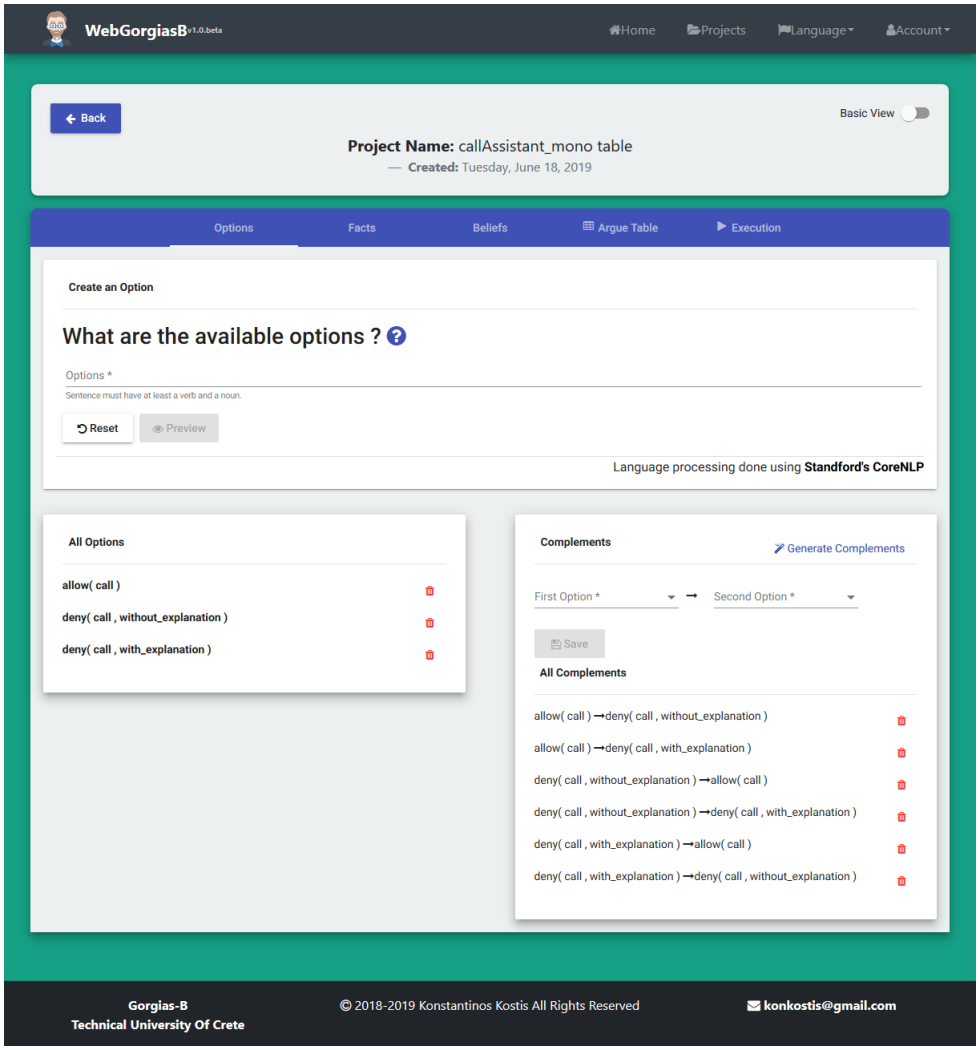

Figure 4.5: Options Page Basic View

4.5.2 Facts Page

The fact page (fig. 4.6) in Basic is consisted by the input form to create a new fact in free text and the list that contains all the facts, where user can review the inserted facts or delete them.

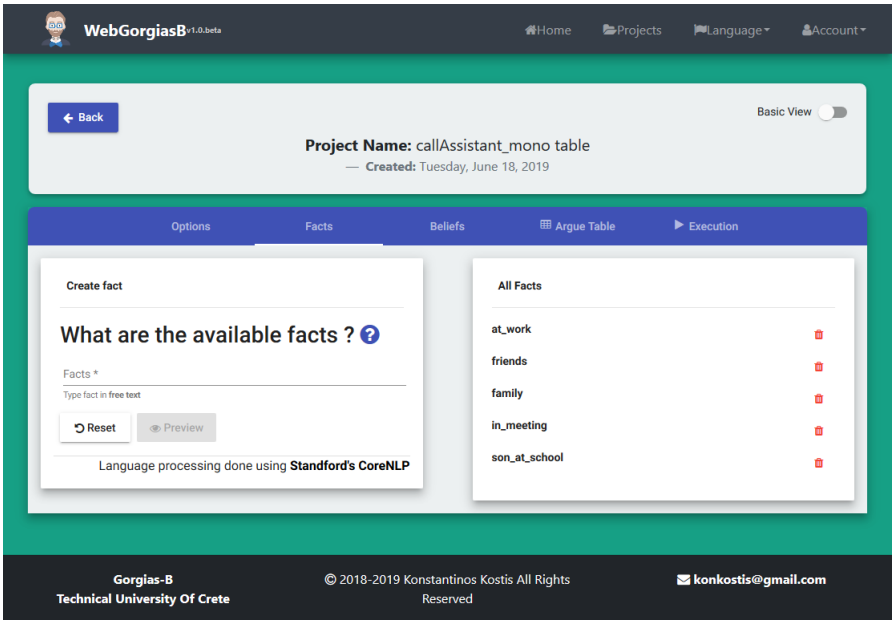

Figure 4.6: Facts Page Basic View

4.5.3 Beliefs Page

The belief page (fig. 4.7) is consisted by the input form to create a new belief in free text and the list that contains all the beliefs, where user can review the inserted beliefs or delete them.

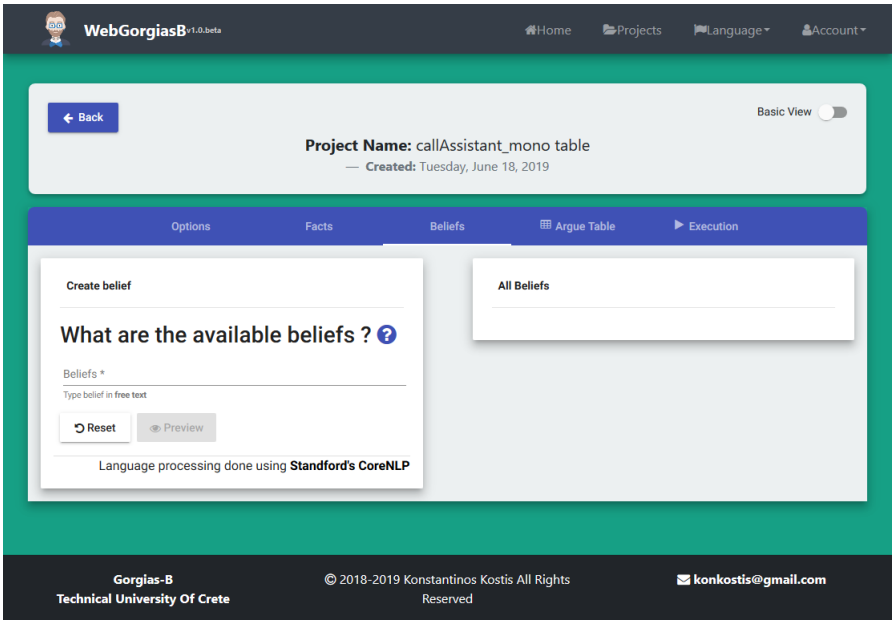

Figure 4.7: Beliefs Page Basic View

4.5.4 Argue Table Page

In this page (fig. 4.8), there is a table where user can expand existing scenarios, delete them or overview them.

More specifically, argue table has as rows each different scenario, and as columns it has the level of each scenario, the scenario name, the available options and the command column which contains buttons that refer to a specific action.

At the columns that correspond to available options, if some options are selected, a checkmark is appeared at this option. Also, user is able to define a selected option as default by clicking the default button, that is appeared when mouse pointer hovers over that option cell.

The last column of the Argue Table, encloses all actions in buttons that correspond to each scenario. These actions are the below:

Expand: When user clicks the expand button, this row collapses and it shows up a small window, which prompts user to select beliefs and facts that will participate as also as the desired options that will be available for the next level scenario that will be created.

Impossible: When user clicks this button, this scenario is removed from this view and it's added ti impossible scenario view.

Delete: When user clicks the delete button, it prompts user to verify whether he wants to delete selected scenario or to cancel delete.

To toggle if the impossible scenarios will be visible or not at the Argue Table, a slider toggle is located at the top of the table, which alters the view of the impossible scenarios.

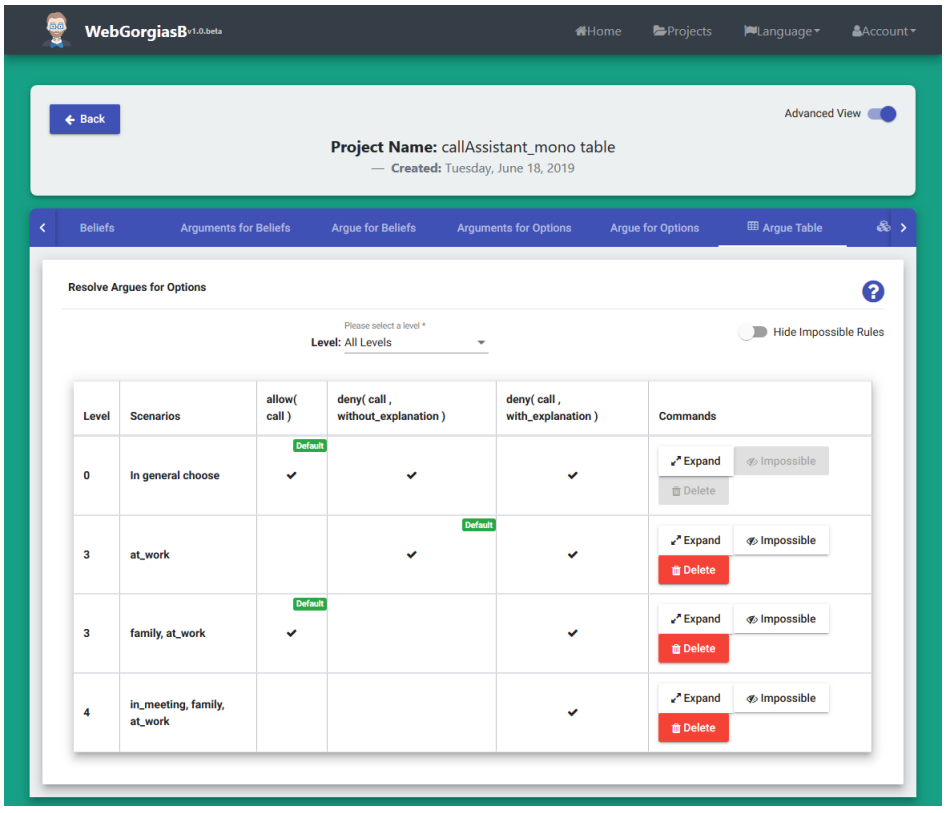

Figure 4.8: Argue Table View

4.5.5 Execution Page

This page (fig. 4.9) contains a input form that user has to fill it to instantiate scenario for execution/simulation and a collapse caret list with header each available option and as body the explanation that has been returned from Gorgias framework if it exists.

The input form contains one field and two buttons. At this field, are listed in a multi select box, all available facts and beliefs that have been created at the working project. After selecting facts and beliefs, user has to select for which options the simulation of the scenario will execute. The buttons, refer to, whether a user would like to simulate the instantiated scenario for all created options, or for a single option.

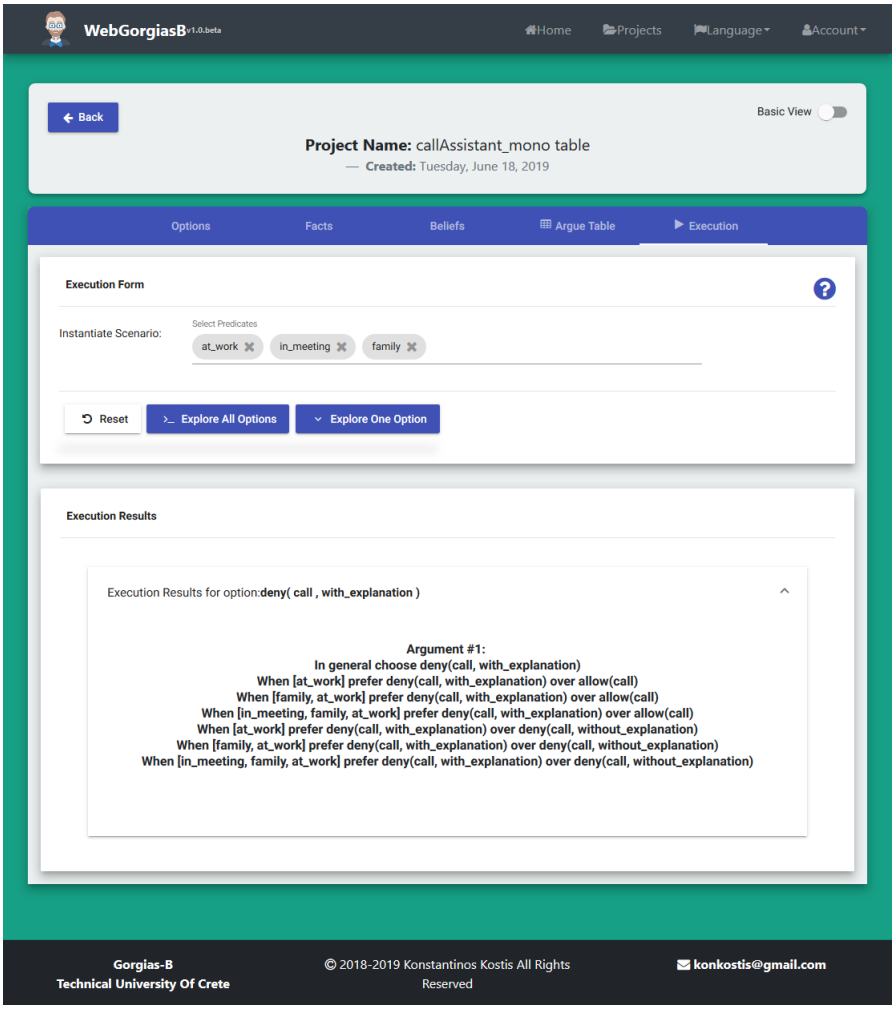

Figure 4.9: Execution Page View

4.6 Advanced View

4.6.1 Option Page

The Option page (fig. 4.10) can be divided into three parts. In the first part, there is the input form to create a new option. This contains three fields:

Option Name: In this field, user fills it with the desired option name. This field is required.

Option Parameters: This field, is the parameter field. This field requires at least one input. So, user should add some context to the option and he can add as many parameters as he wants that describe the option better. There are two buttons aside each parameter field, with the first button to insert a new parameter and the other button to remove selected parameter.

Option Negation: With this checkbox, if user checks it, an identical option like the one he just created, will be created also but it will be the negation of it. For example eat(food) and not_eat(food).

The second part of page, is the list of all created options, where can review or delete them. In the last part, there is the complement section. In this section, user can define from the available options, which of them are complements and add them at the complements list that there is below or can click the button "Generate Complements" that automatically generated complements for all options defined.

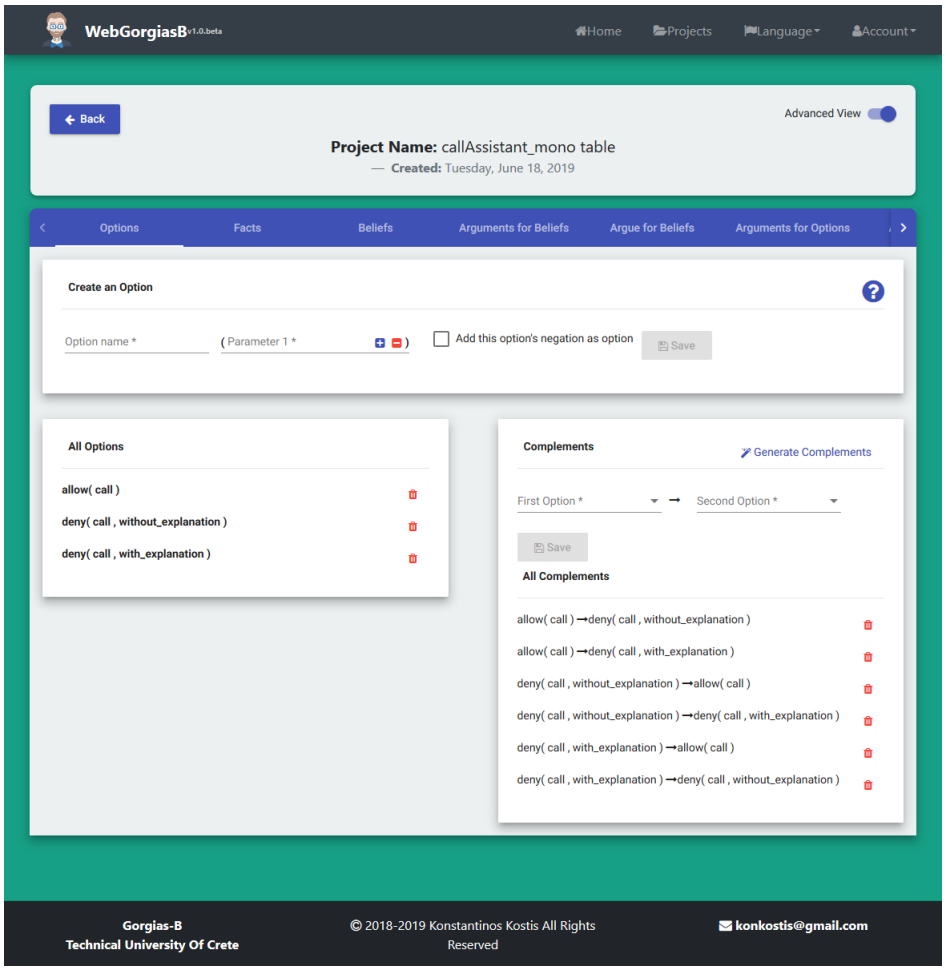

Figure 4.10: Home Page View

4.6.2 Facts Page

Fact Page in Advanced view, (fig. 4.11), is divided into two main components. The first component is an input form for belief creation, containing three fields:

Fact Name: In this field, user fills it with the desired fact name. This field is required.

Fact Parameters: This field, is the parameter field. This field is optional. So, if a user wants to add some context to the fact, he can add as many parameters as he wants. There are two buttons aside each parameter field, with the first button to insert a new parameter and the other button to remove selected parameter.

The other component, contains the list of the created beliefs, with their name and an option to delete them per will.

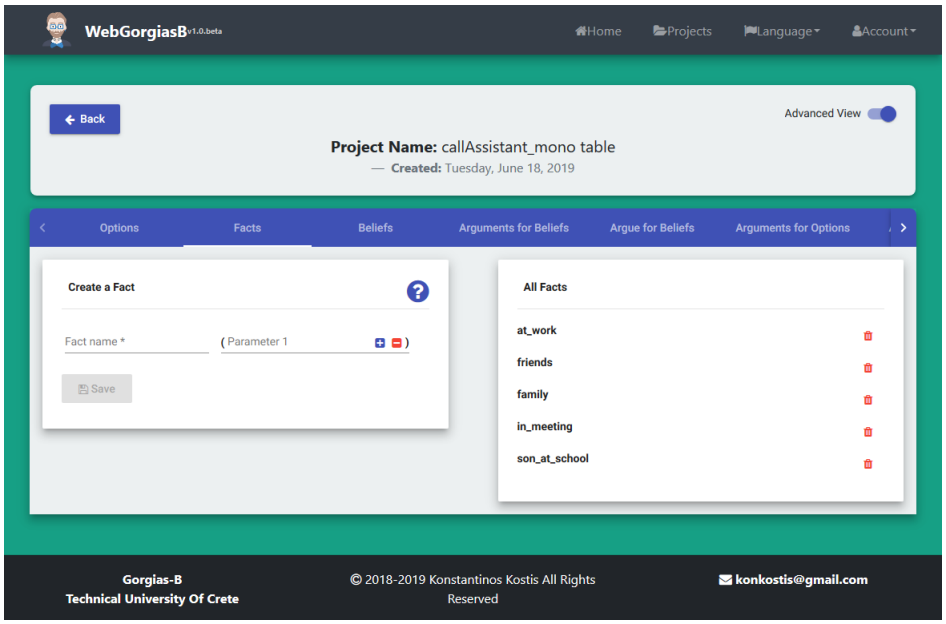

Figure 4.11: Fact Page Advanced View

4.6.3 Belief Page

Belief Page in Advanced view, (fig. 4.12), is divided into two main components. The first component is an input form for belief creation, containing three fields:

Belief Name: In this field, user fills it with the desired belief name. This field is required.

Belief Parameters: This field, is the parameter field. This field is optional. So, if a user wants to add some context to the belief, he can add as many parameters as he wants. There are two buttons aside each parameter field, with the first button to insert a new parameter and the other button to remove selected parameter.

Abducible belief: Last form field, is a checkbox. By clicking this checkbox, user can define the new belief as Abducible.

The other component, contains the list of the created beliefs, with their name and an option to delete them per will.

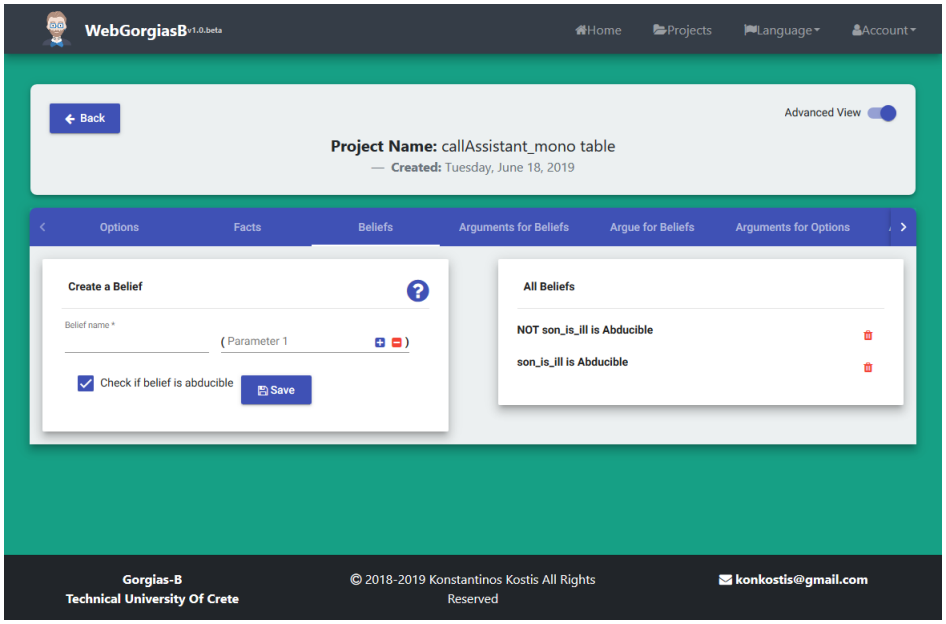

Figure 4.12: Belief Page Advanced View

4.6.4 Argument for Options Page

In Argument for Options Page (fig. 4.13), user can define the Object-Level Arguments of the scenario, or in other words, the first level Scenarios Preferences. There is an input form with two fields. The first field is a select menu containing all options that have been created and the other field is a multi select menu which contains all the facts and beliefs created. When user selects one element from this field, this is listed below with an option to remove it.

At the other half of the page, are listed all Arguments for Options created, in order for user to review them or to delete them.

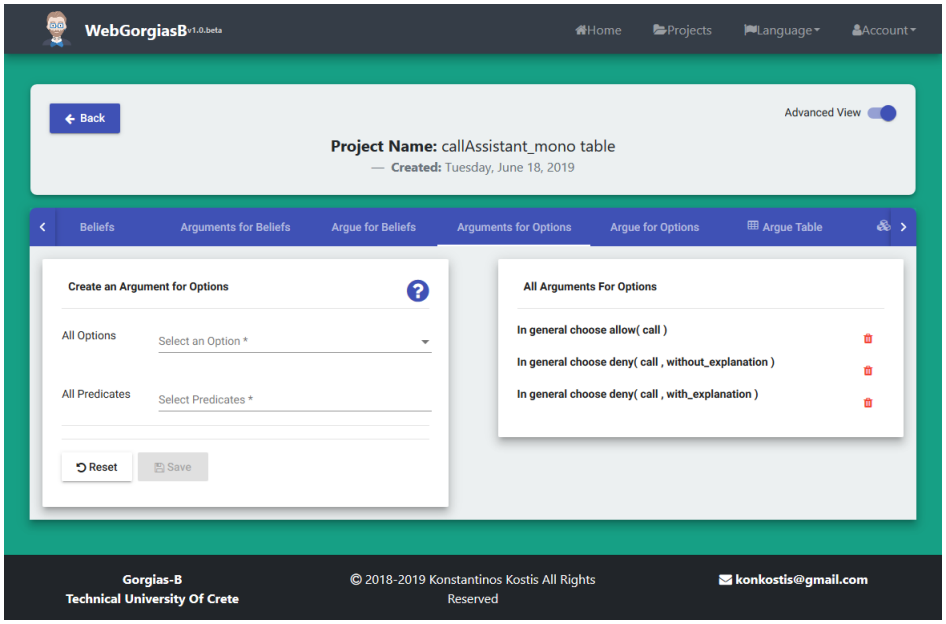

Figure 4.13: Argument for Options Page View

4.6.5 Argue for Options Page

In this page (fig. 4.14), are located all the necessary elements to review delete and define scenarios preferences. To define scenario preferences, the input form is used. The fields of the form are the following:

Level: This drop-down menu, contains all the available levels that already defined scenarios are. This field is required.

Scenarios: When user selects a level, then the Scenarios drop-down list is filled with the conflicted scenarios name, if any exists. This field is also required.

Preferred Options: After selecting scenario that user will refer, available options are calculated and drop-down list is filled with them.

Weaker Options: Weaker options drop-down list contains also all the available options for this conflicting scenario selected in Scenarios field. It has also an option "All Others".

All predicates: In this form field, there are all the created by user facts and beliefs at this project. When user selects one element from the list, this is listed below with an option to remove it.

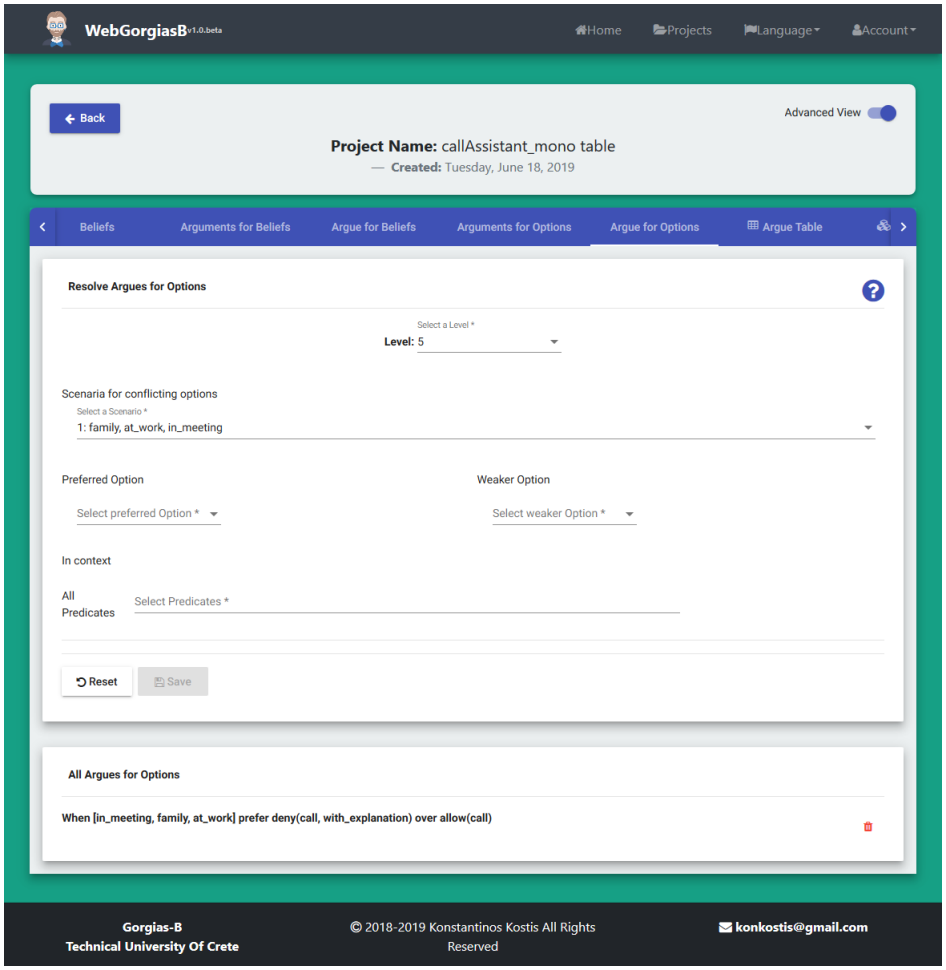

Figure 4.14: Home Page View

4.6.6 Prolog File Page

Lastly, in the Prolog File Page (fig. 4.15) in Advanced View, is displayed the Prolog source code that will be sent to Gorgias framework to process it. The source code is presented well formatted to user, in order to be clearer and easier to read.

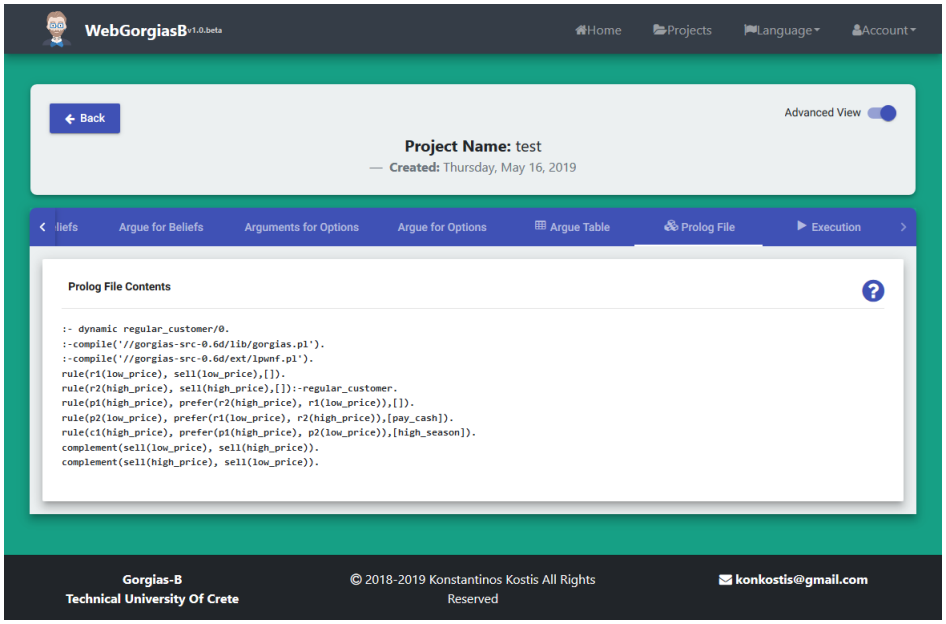

Figure 4.15: Prolog File Page View

Chapter 5

Application Design

The design was based on the idea that the data resulting from user actions will be analyzed by the application and stored in a NoSQL database. Subsequently, using a suitable process such data are converted to a format which can identify the Prolog and stored these as a file in NoSQL base. Finally, this file will be used by the Prolog engine to execute and export the result. Main design model of the application is shown in figure 5.1

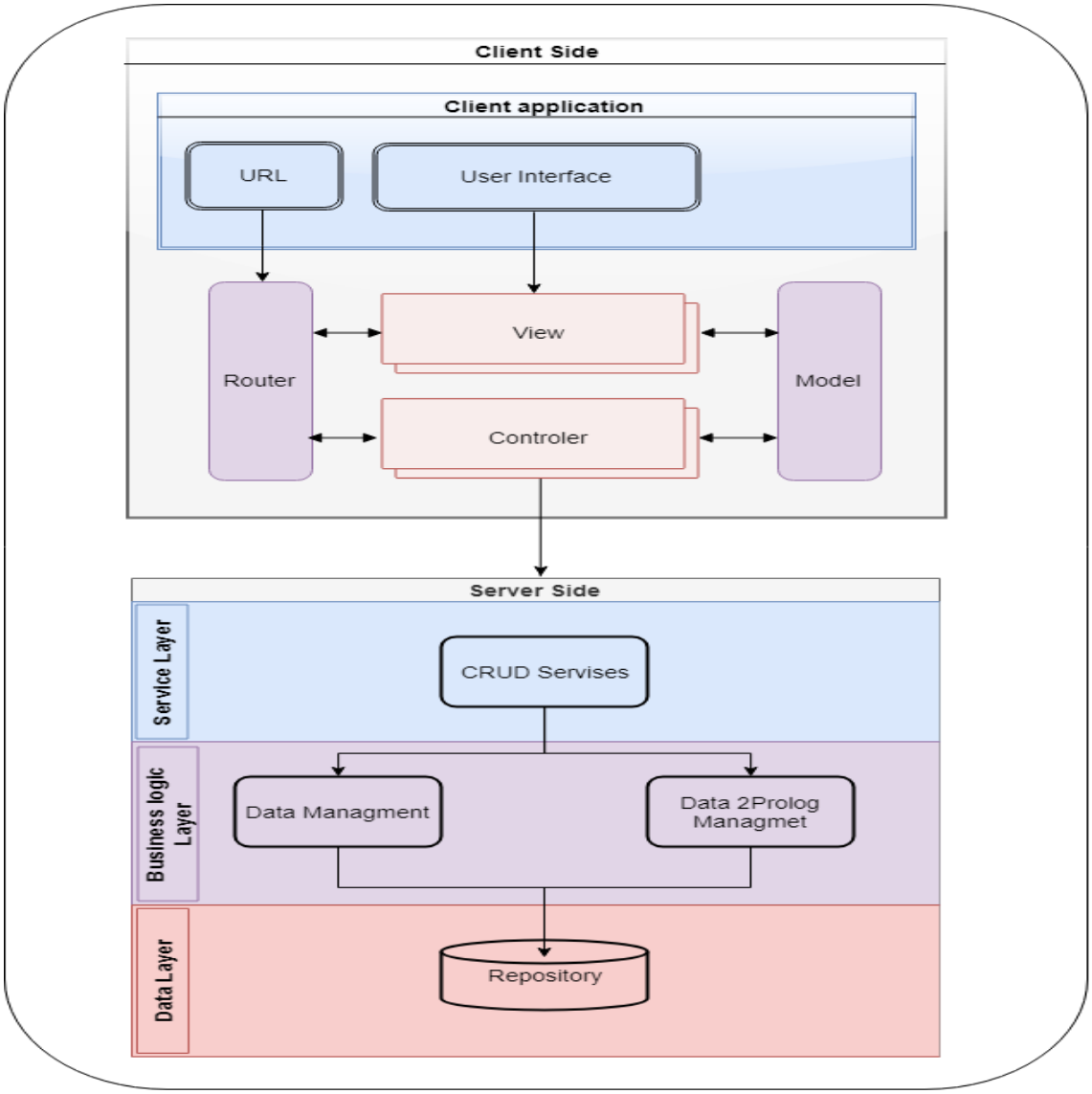

Figure 5.1: Application's main design model

The overall application was designed to take advantage of the principles and benefits of the Model-View-Controller (MVC) design model. This means that distinct modules will be created to control the presentation of the data, filtering it according to the user's criteria and managing it in a data model.

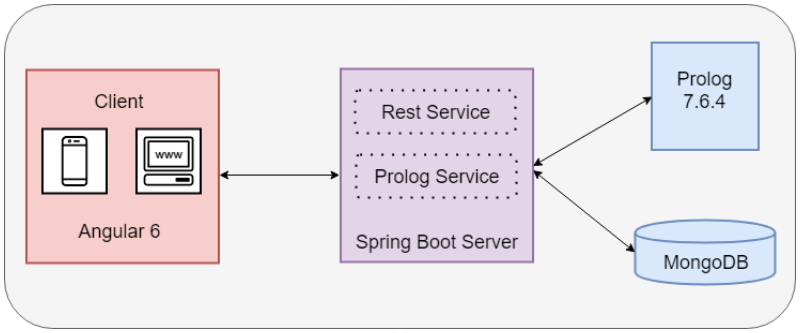

Figure 5.2: Application's distinct modules design

From the above figure 5.2, it appears that generated data are transmitted through a REST service at system Model level, where they are stored for use in further questions that will result in their execution in Prolog environment and the return of the result. At a second level, the Model is specialized in each module to match the technologies that meet the corresponding requirements mentioned above.

5.0.1 Client Side

At the client level, the form input environment is created for each different project that a user wants to create. Also, the browser displays the results that resulted from the execution of the defined project-specific scenarios, in a Prolog environment.

5.0.2 Server Side

At server side, application implements the services below:

- Communication with Prolog and conversion of the data into a format appropriate for their recognition by Prolog
- Storage service of the created data in the database
- Filtering data based on user criteria
- Conversion of free text to Prolog Predicate using Stanford's Natural Language Processing software CoreNLP[29].

Client's software is based on the Model-View-Controller model, as well as the server model. This is implemented through restful services that perform the tasks of:

- Prepare and transmit queries to the data model so they can be sent to the Prolog engine
- Send the data resulting from execution in Prolog to the data model so that it is displayed to the user

Simultaneous use of MVC model both for the client and the server offer specific advantages that are critical to an application in which significant volumes of data (in a production environment) arise. Such advantages are [30]:

Sparse communication and data exchange between client and server, as data functions transmitted to the client can be managed locally, minimizing delays in the application response.

Improving the response of the application, since processing focuses on a smallest set of application-related data on the client side.

Load distribution from a single server, which implements the model controller in multiple browsers.

5.1 Client Side MVC Pattern

The client of the application has been based on Model-View-Controller (MVC). This implementation ensures that the server transmits the data application only. The data is processed by the customer according to the standard MVC model, so that their presentation format is finally displayed in the client browser. An important role in this implementation has been played by the existence of several programming frameworks, which, with the use of JavaScript, implement it in a relatively simple way.

Client access to an MVC programming framework makes it easy to develop applications that manage dynamic data. More specifically, due to the fact that the MVC implementation to the client minimizes the need for exchange of formatted data with the server, it is offered to develop basic CRUD services, which manage raw, unformatted data. CRUD services (create, read, update, delete are functions that aim to store, refresh, find and delete data from a database.) The basic principle on which MVC implementation is based on the customer is the implantation of labels in the code HTML, which trigger a corresponding JavaScript language code In this context, the browser loads the HTML page and creates the Document Object Model for it Then the DOM model translates and gathers all the tags in the programming box Finally, through the labels the associated JavaScript functions are activated and the page is reloaded in its final form.

5.2 Server Side

Server implements these main services:

1. Communication with Prolog and conversion of the data into a format appropriate for their recognition by Prolog **(PrologService)**
2. Business Logic Management Service **(REST Service)**
3. Conversion of free text to Prolog Predicate using Stanford's Natural Language Processing software CoreNLP[29] **(CoreNLPService)**
4. Storage service of the created data in the database **(Database Service)**

5.2.1 PrologService

PrologService can be divided in two main subcategories.

Convert to Prolog Service

In this service, the data entered by the user into the forms in the application's interface, is collected and analyzed. Source code in Prolog programming language results from the analysis process. After analysis process, this source code is encoded in Byte array and stored in the database after it has been assigned the appropriate id to be recognized by the application. An example of this conversion is shown in figure 5.3.

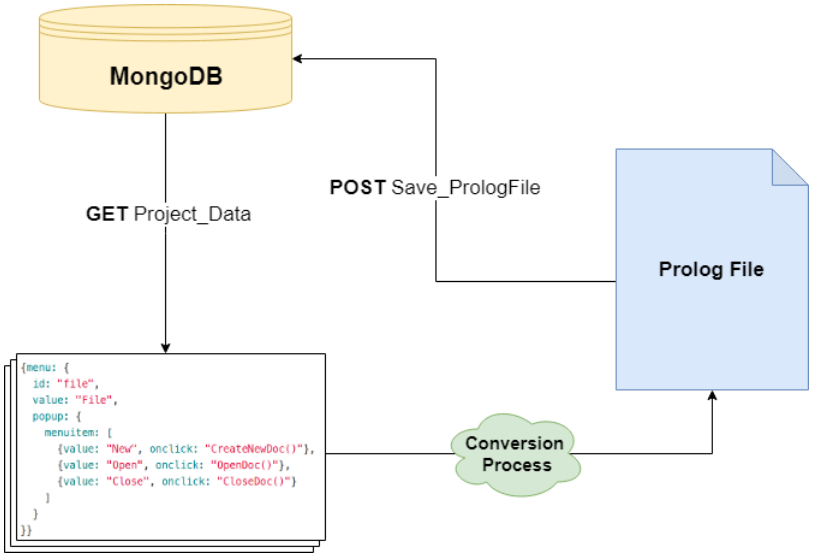

Figure 5.3: Data Objects to Prolog conversion

JPL Service

This service, handles the whole communication between JAVA and prolog, using JPL dependency. JPL is an important dependency for the developed application's architecture. [31] JPL is a set of Java classes and functions providing an interface between Java and Prolog. JPL uses the Java Native Interface (JNI) to connect to a Prolog engine through the Prolog Foreign Language Interface (FLI), which is more or less in the process of being standardized in various implementations of Prolog. JPL is not a pure Java implementation of Prolog; it makes extensive use of native implementations of Prolog on supported platforms. Main tasks of this service are to initialize the Prolog Engine, and send queries to it, which include the paths for the binary files that have to be loaded in Prolog in order to execute it. Also, the results that Prolog exports during execution, are handled by PrologService and are transmitted to client's Model in order the user to be able to view it.

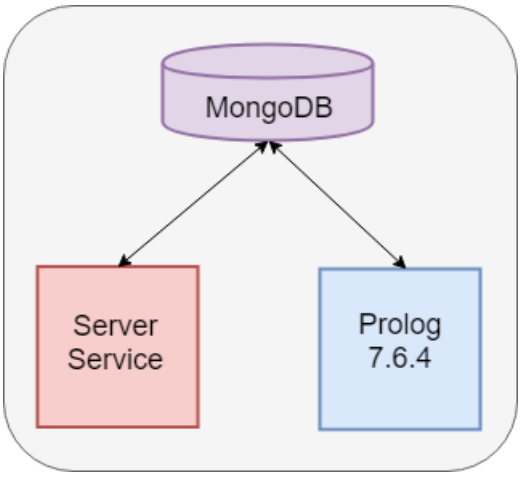

Figure 5.4: Prolog execution procedure

5.2.2 REST Service

The Representational State Transfer (REST) architecture creates an interface between systems by using the HTTP protocol for data traffic and activating functions on those data using formats such as XML and JSON.
The main features of REST architecture are:

1. This is a client / server protocol that does not maintain the status of the communication channel. This means that neither the client nor the server needs to "remember" any previous status of the communication channel in order to communicate again.

   Objects in the REST architecture are managed through the URI. Any resource in a REST system is addressed by its URI. The URI allows access to the information for modification or deletion.

2. Uniform interface. In order to transmit data via REST, the system applies specific operations to the available resources (POST, GET, PUT and DELETE) provided that they are addressed with a URI. This mechanism creates uniformity in the management of information managed by a server.

3. Layered Architecture: It is possible to create hierarchical layers between the modules of an application. In this context, the programming interface may be available from server A, data storage is on a B server, and authentication requests are handled by a third C server.

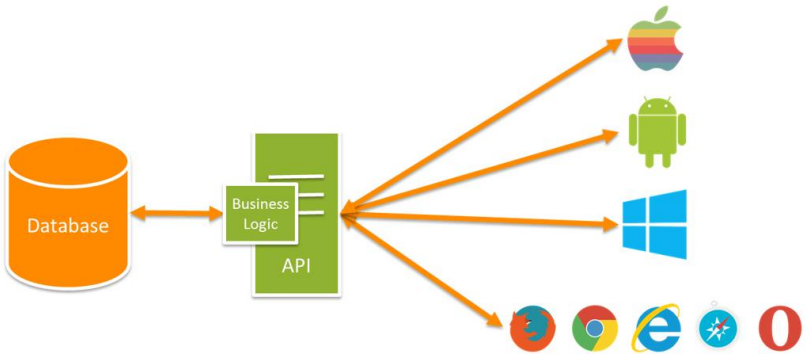

Figure 5.5: REST API architecture

5.2.3 CoreNLPService

Stanford's CoreNLP [32] provides a set of human language technology tools. It can give the base forms of words, their parts of speech, whether they are names of companies, people, etc., normalize dates, times, and numeric quantities, mark up the structure of sentences in terms of phrases and syntactic dependencies, indicate which noun phrases refer to the same entities, indicate sentiment, extract particular or open-class relations between entity mentions, get the quotes people said, etc.Stanford CoreNLP's goal is to make it very easy to apply a bunch of linguistic analysis tools to a piece of text. CoreNLP is available in English language.

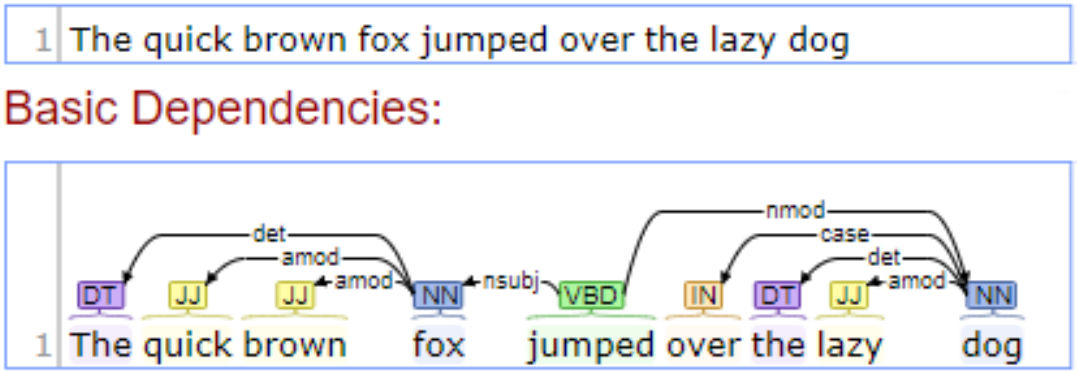

Figure 5.6: Usage of Core NLP software

5.2.4 Database Service

The NoSQL technology was chosen to implement the storage and analysis functions of data from user input processes. NoSQL means those databases that have a data storage and retrieval mechanism different from that of relational databases (SQL), which is based on the storage of semi-structured objects. The data types used by NoSQL databases (key-value, documents, etc.) are different from those of relational databases, which makes them more efficient in various functions, mainly data analysis [33]

5.3 Server Side MVC Pattern

The three services described above, implemented on the side of the server, work together on the basis of the MVC model. The server's MVC model implements three building blocks.

1. The View section serves to display the pages that contain the argumentation data such as options, facts and scenarios preferences that correspond at each user created project.

2. The Controller section coordinates the flow of data to and from the Model section and View. The Controller reacts to the actions of the user and retrieves information from the Model in order to channel them into the View section. Controller consists of the REST services, that define the actions that have to be done after each request as also the return of the responses from requests.

3. The Model section represents the data stored in the database. The implementation of this section matches the database data to objects. The items to be stored are transmitted to the System Service Database for service.

In summary, the MVC model is illustrated in Figure 5.7

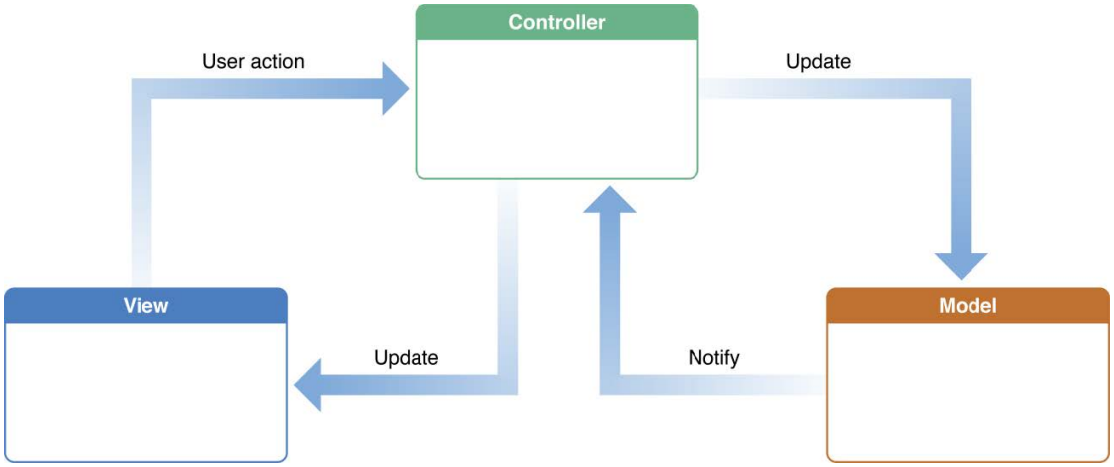

Figure 5.7: Model-View-Controller model representation

Chapter 6

Implementation & Evaluation

6.1 Client Side

In order to implement client-side applications, only technologies that can be run on a standard browser without the need for any additional software (such as a runtime) have been used. Given that an application client who consumes the server services is a browser only, the design decision to implement it has led to the adoption of the following technologies and programming tools:

- HTML5
- CSS3
- Angular

The choice of Angular programming framework was based on the simplified implementation of the MVC model on the client side by embedding additional HTML tags in the code that triggered the execution of functions through the frame. The advantage of Angular lies in the fact that HTML remains the page description language while synchronizing data from the User Interface with the JavaScript objects created by the page using a 2-way binding technique. Angular approach is the implementation of "event handlers" and their activation when the programming framework decides that one of them should be activated. Using Angular programming box, a complete distinction is made between DOM model, models and functionality (as developed in the controllers).

MVC Pattern

The design model on the client side is Model-View-Controller, which was implemented with Angular. On the basis of this model, building blocks were created, which can be reused on the same or different pages. These units derive data from the data model and feed the level view. The use of Angular as a client-side programming framework implements the MVC model as follows:

View It is implemented with the HTML description language and additional Angular tags, so that the Document Object Model created by the page loaded in the browser can be checked.

Controller Controller sends requests to the REST Service and assigns the information that is required for the View level to the RxJS object of the programming frame. It also implements callback functions, which are triggered as a response to specific events.
It is charged with validation of data so that it is processed by the client and does not burden the server with such tasks.

Figure **6.1** and **6.2** shows MVC model's entities and the link between them.

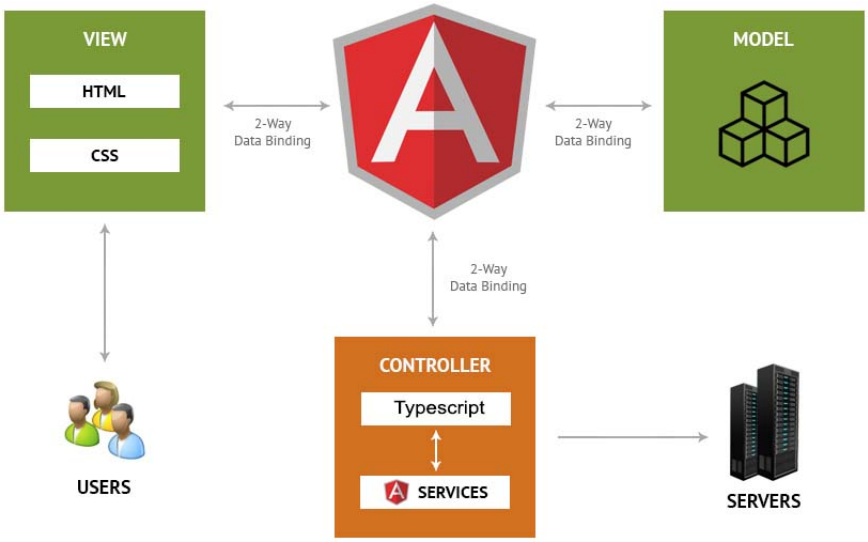

Figure 6.1: Angular's MVC Workflow with 2-way data binding

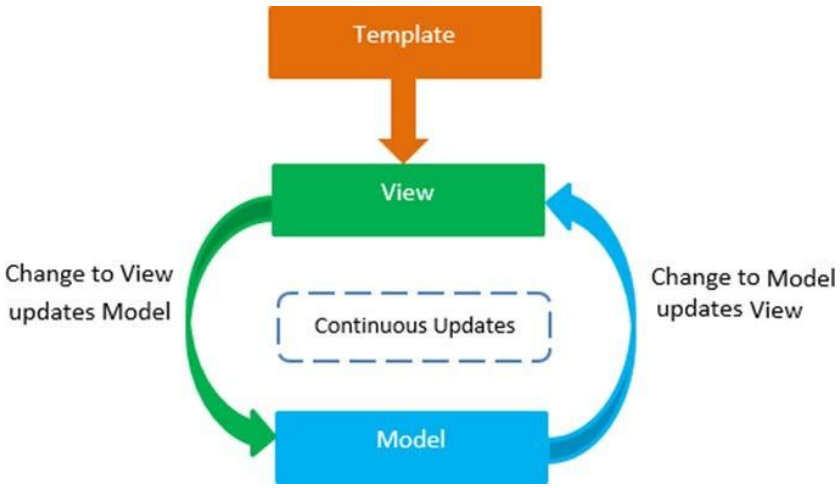

Figure 6.2: Angular's 2-way data binding technology

Components

Based on Angular's architecture, client consists of different components. Each component is implemented according to the MVC model and has its own Controller, View and Service. Angular facilitates the reuse of a building unit by using the label:

```
<component_name></component_name>
```

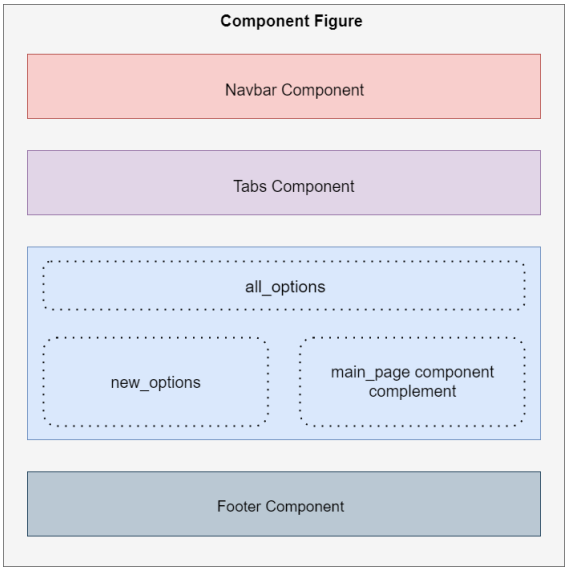

Figure 6.3: Client's page structure, using Angular's components

6.2 Server Side

6.2.1 ScenarioService

ScenarioService is implemented in Spring Boot framework. All the different functionality of the developed application that refers to scenarios, is implemented in this service. To be more specific, this functionality is summarized with the following functions.

First of all, scenarioService includes the functions needed, to group all the scenarios created, by their name, in a Table View (table 6.1). So, all the scenario preferences that reflects each scenario are grouped together and then it's packed as JSON array in order to return it to the Client Side and then to graphical user interface (GUI) of the application to be presented in the Argue Table custom view.

In addition, there is the function to create a scenario based on selected beliefs and facts, accompanied with the appropriate option for each scenario.

	Scenarios	*Options*		
		allow(call)	*deny(call, without_explanation)*	*deny(call, with_explanation)*
1	In general choose	**X**	**X**	**X**
2	at_work	**X**	**X (Default)**	**X**
3	at_work, family(call)	**X (Default)**		**X**
4	at_work, family(call), in_meeting			**X**
5	at_work, family(call), in_meeting, from_son(call)	**X**		**X**
6	at_work, family(call), in_meeting , from_son(call), son_is_ill	**X**		
7	at_work, family(call), in_meeting , from_son(call), not_son_is_ill			**X**

Table 6.1: Argue table for Call Assistant example

In order to make scenarios more specific and express preferences for an option over another, expand function has been implemented. This function is illustrated in alg.1. This algorithm represents how system can manage user's inputs, to generate arguments in a specific scenario over an argument in more general scenarios. So, once we set up, for each option o in O where O contains all the options that users have selected to create a preference for them. Algorithm

accesses all arguments in that scenario that has as preferred option that selected options and also the arguments that have the selected options as a non-preferred option and then checks if the preferred options of these arguments are complements. If they are, then algorithm generates the priority arguments at a higher level than the previous arguments. Due to the possibility of the existence of default options in the scenarios examined, to avoid conflicting preferences, a correction algorithm is executed to correct possible mistakes about hierarchies.

Function `expandScenario`(SP^{lvl}, o)**:**
1 **for** *each option* $o \in SP^{lvl}$ **do**
2 **for** *each* $arg^{SP^{lvl-1}}_{o\_over\_o'}$ **do**
3 **for** *each* $arg^{SP^{lvl-1}}_{o'\_over\_o}$ **do**
4 **if** $complements(o, o')$ **then**
5 add $arg^{SP^{lvl}}_{o\_over\_o'} = (S^{lvl} - S^{lvl-1}) \rhd (arg^{SP^{lvl-1}}_{o\_over\_o'} > arg^{SP^{lvl-1}}_{o'\_over\_o})$
6 call *autoCorrectArgs*()

Algorithm 1: Central Algorithm For Scenario Refine

Function `autoCorrectArgs()`**:**
1 **for** *each option* $o \in O'$ **do**
2 **if** $isDefault(SP, o)$ **then**
3 $insertPreference(SP, o)$

Algorithm 2: Central Algorithm For Correcting Arguments after Refine

Multiple scenarios may have common elements and can also be combined, which results in possible conflicts between the available options of each one. These conflicts get analyzed and are presented to user in Argue Table, in order to make a decision for these.

As stated at previous chapters, one of the innovations that this implementation offers, is the function for users to give to an option higher priority over all others at a scenario. This is the central algorithm alg.3 for defining an option as default. Algorithm searches for arguments of a specific scenario S, which their lower priority option is this that user wants to define as default. Then, at each of these arguments, algorithm creates a counter-argument at a higher level, by setting higher priority for the default option and the context to true.

Function `insertPreference`(SP^{lvl}, o)**:**
1 **for** $arg^{SP^{lvl}_x}_{o'\_over\_default}$ **do**
2 add $arg^{SP^{lvl+1}}_{default\_over\_o'} = (true) \rhd (arg^{SP^{lvl}}_{default\_over\_o'} > arg^{SP^{lvl}}_{o'\_over\_default})$

Algorithm 3: Central Algorithm For Inserting a Preference for an Option

Function `isDefault`(S, o)**:**
1 **for** *each* $arg^{S^{lvl}_x}_{o\_over\_o'}$ *where* $S^{lvl}_x \in S$ **do**
2 **if** $(S^{lvl} - S^{lvl-1}) = \emptyset$ **then**
3 **if** $lvl > 1$ **then**
4 result = true
5 **if** $supp\_information(arg^{S^{lvl}}_{o\_over\_o'}) \neq supp\_information(arg^{S^{lvl-1}}_{o'\_over\_o})$ **then**
6 result = false;
return result

Algorithm 4: Algorithm for Identifying whether an option (o) is marked as Default in a specific scenario (S)

Furthermore, there is a possibility that a user would like to express a priority for an option o at a scenario (S^{lvl}), or in other words a scenario preference (SP), and that option to be unavailable. So, an auto-complete algorithm is implemented, to search whether this option was available in a previous scenario $S^{lvl'} where lvl' < lvl$. If this information is found, a preference for this option in that scenario is generated, as also a default preference for all the other options that where selected before. Therefore, the initial preference that user asked, can now be completed successfully. This whole procedure described above, is illustrated in alg.5 below.

Function `autoCompletePriorities`(S^{lvl}, o)**:**
1. call $expandScenario(S^{lvl-1}\ where S^{lvl-1} \in S^{lvl}, o)$
2. **for** $arg^{S^{lvl}_x}_{t\_over\_t'}$ *where* $t \neq o$ **do**
3. call $insertPreference(S^{lvl}, t)$
4. call $expandScenario(S^{lvl}, o)$

Algorithm 5: Central Algorithm For auto completing priorities for unavailable options

Last but not least, this service contains the newly introduced functionality of defining or undefining a whole generated scenario as Impossible and excluding it or reincluding it from the Argue Table custom view. An example of an Impossible scenario is that it's winter and its summer simultaneously.
$a(allow(call)) = \{in\_general\} \rhd allow(call)$
$a(deny(call, without\_explanation)) = \{in\_general\} \rhd deny(call, without\_explanation)$
$a(deny(call, with\_explanation)) = \{in\_general\} \rhd deny(call, with\_explanation)$

Given the example shown in Argue Table 6.1,and having created the object level rules for each option when a user tries to expand the newly created scenario to produce the second line (2) of the table, expandScenario(SP^{lvl}, o) function will be called like this:

expandScenario($SP^2_{at\_work,}, o$)
$o = [allow(call), deny(call, with\_explanation), deny(call, without\_explanation)]$

and the arguments that will be generated are the following:

$p_1(allow(call)) = \{at\_work\} \rhd (allow(call) > (deny(call, without\_explanation))$
$p_2(allow(call)) = \{at\_work\} \rhd (allow(call) > deny(call, with\_explanation))$
$p_3(deny(call, without\_explanation)) =$
$\{at\_work\} \rhd (deny(call, without\_explanation) > allow(call))$
$p_4(deny(call, without\_explanation)) =$
$\{at\_work\} \rhd (deny(call, without\_explanation) > deny(call, with\_explanation))$
$p_5(deny(call, with\_explanation)) = \{at\_work\} \rhd (deny(call, with\_explanation) > allow(call))$
$p_6(deny(call, with\_explanation)) =$
$\{at\_work\} \rhd (deny(call, with\_explanation) > deny(call, without\_explanation))$

In order to define the second option $deny(call, with_e xplanation)$ as default the insertPreference(SP^{lvl}, o) function is called like this:

insertPreference($SP^2_{at\_work}, deny(call, with\_explanation)$)

and these arguments are generated:

$c_1(allow(call)) = \{true\} \rhd (deny(call, without\_explanation) > allow(call))$
$c_2(allow(call)) = \{true\} \rhd (deny(call, without\_explanation) > deny(call, with\_explanation))$

With the previous same functions are generated the arguments that refer to $at\_work, family(call)$ scenario where $allow(call)$ is the selected option:

$c_3(allow(call)) = \{family\} \rhd (allow(call) > deny(call, without\_explanation))$
$c_4(allow(call)) = \{true\} \rhd (allow(call) > deny(call, without\_explanation))$
$c_5(allow(call)) = \{family\} \rhd (allow(call) > deny(call, with\_explanation))$

Subsequently, the next scenario that will be created is line 4. The function to be called is

autoCompletePriorities($S^4_{at\_work, family(call), in\_meeting}, deny(call, with\_explanation)$)

because the option $deny(call, with_e xplanation)$ is not available with the current selected options in the previous scenario preferences. The arguments that will be generated are:

$c_6(deny(call, with\_explanation) =$
$\{family\} \rhd (deny(call, with\_explanation) > deny(call, with\_explanation))$
$c_7(deny(call, with\_explanation) =$
$\{family\} \rhd (deny(call, with\_explanation) > deny(call, without\_explanation))$
$c_8(deny(call, with\_explanation) =$
$\{true\} \rhd (deny(call, with\_explanation) > deny(call, without\_explanation))$
$c_9(allow(call)) = \{true\} \rhd (allow(call) > deny(call, with\_explanation))$

6.2.2 PrologService

This service is implemented by the Spring-Boot programming framework. Prolog Service contains all the functions needed to successfully compile and simulate each project. Specifically, the whole procedure from compilation to simulation can be summarized in the following 4 steps.

As soon as the user chooses to run his project and after initializing of the execution with the facts and beliefs that will participate in it as well as the option or all options, PrologService is called. Initially, the service reads from the database all the data that correspond to this particular project, and then the data is compiled into Prolog language. This process is as follows.

The first step is to convert all the project data to Prolog file. All different facts and beliefs are stated in the file with the Dynamic term. Each belief or fact is transformed into "name/num_parameters" as described below.

```
:-dynamic fact_name/number_of_parameters, .... .
```

Following, is the definition of the libraries to be used. These are the libraries of Gorgias that are in a directory in the root folder of the application. Then all the different rules are converted one by one into Prolog Rules, paying great attention to their priority over

other rules as well as the identifying name they will take to avoid which conflicts. The data that accompanies each rule such as options and facts/beliefs are also converted. Also the complements for the different options that are defined in this project are also inserted in the compilation process and take the form such as below.

```
complement(allow(Call), deny(Call)).
```

Finally, if some beliefs are defined as abducible, then they are also introduced in this form.

```
abducible(son_is_ill, []).
```

After the transformation of the file is done successfully, this information is stored in a byte array to the database.

Then, the second step is, to establish a connection between Prolog and Spring Boot. This is achieved by using the JPL library provided by SWI-Prolog. To initialize connection, JPL.init() method is called. Once it's successful, all HTTP libraries and protocols are declared in order to permit to Prolog the handling of HTTP requests (GET, POST).

```
use_module(http/http_open).
```

Afterwards, the next step is to load the generated file from first step to Prolog and simulate it. So, system makes a method call "http_open" in Prolog to load the file as a Stream. After the completion of loading procedure, service uses user's inputs from the execution page of GUI. These user inputs contains info about the instantiation of the scenario that is going to be simulated. This info is the selected beliefs or facts that take part in the instantiated scenario and the options that will be examined. Therefore, to execute and to return the execution results, Gorgias command "prove" is called. The produced results accompanied with a minimal are returned eventually as HTTP response to service.

Finally, the last step is to lush the File from Prolog engine's memory as also to close the connection between Prolog and Spring-Boot.

6.2.3 REST Service

REST services implement two functional requirements:

1. Storage of the data generated by the user's actions in the corresponding forms in the UI of the application. Each different entity is stored in the appropriate MongoDB collection.

2. Recover data from the database, based on client queries. Queries are served by the Database Service, where they are disseminated via the corresponding REST service.

6.2.4 CoreNLPService

CoreNLPService was implemented using the Spring-Boot framework. Proprietary libraries are included from Maven Repository and compiled. In order to achieve the goal that was defined is functional requirements, to create user-friendly UI and to be easy for non expert users to use the application, difficult and complex Prolog elements, such as Predicates should be visualised in another way. So, the basic idea is to guide the user to write a free form text, which after language processing, it will be transformed into Prolog's argument's structure predicate form. Or in other words to extract predicate from free text i a way that should ensure almost complete acquisition of predicate-argument structures from text. We consider

extraction of predicate-arguments, structures from a single text with a substantial narrative part.

Verbs play a fundamental role in Natural Language Processing (NLP), so verb information in lexicons is essential. A verb as a predicate identifies a relation between entities denoted by the subjects and complements. So, coreNLPService, utilizing the power of Stanford's coreNLP tool written in Java, processes the give sentence, with the restriction that is written in well-English, and analyzes the entity relation of the sentence by verb. When syntactical analysis completes, entities are transformed into word functions as presented in below figure 6.4. An abstract form of representation would be $verb(subject, object, nouns)$ with minor changes per input. This transformation is presented to user to approve it or to adjust it.

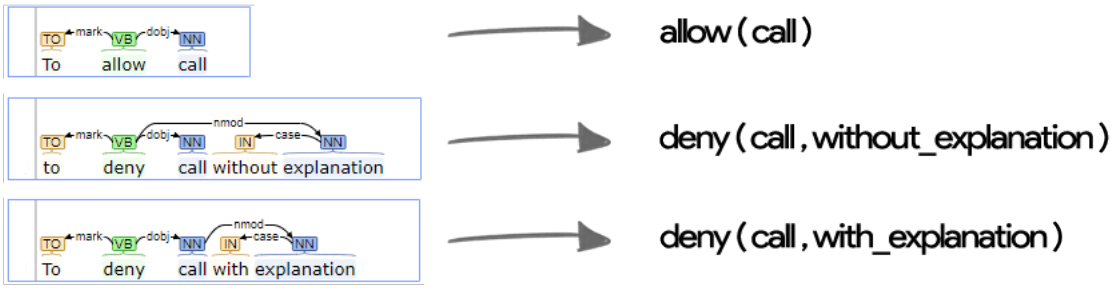

Figure 6.4: Core NLP text transform

6.2.5 Database Service

6.3 System Evaluation and User Feedback

6.3.1 Think aloud evaluation

The think aloud protocol is a protocol used to collect data from usability testing. Involves users who undertake to perform certain defined actions that have been asked for them, while at the same time they must express aloud any thought. More specifically, users must say what they see, what seemed strange to them, what they feel, and that it is captured (captured?) By observers in the form of notes, or even better in the form of videos . In the latter case, developers can use the video to see how users reacted to what they were asked to do and what they saw during the test. In this way, a more complete picture is formed on the functionalities to be developed, on the design of the pages and on the organization and editing of the project.

6.3.2 User Feedback

Primary Stage

Once the design of the pages is over, we apply the think aloud protocol. For the application's needs, an evaluation was made by 3 users whose choice was not random at all. The two users were designers of web sites and the third one was a student. The first two focused on the simplicity that the interfaces designed on the paper had to show more. They advised on the navigability of the application and the position of the side column with the results and prompted a change of position. The student emphasized on the lack of help section and proposed to add at each page a help button containing help tips accompanied with simple small examples, in order to guide the users on how to use the application and each different aspect

of it. Student also indicated that the complement section in options page was too complex and time-consuming because of the iteration of the same procedure over and over until all combinations are over. So, he indicated to add an "auto-generate complements" button on top of the complement section, so that is clearly visible by users. The functionality of this button, will be to automatically create for all the combinations of the options, the pairs of complements accompanied with a message. The message informs users about the successful or not completion of the task.

Lastly, student, due to his lack of experience on argumentation and argumentation frameworks, he insisted that for a naïve user like him, the information and the tabs at the application were many enough and it was a bit distracting for him. Therefore, a solution for this problem, would be to create two custom views. One for naïve users, and one for expert users. The naïve user view, or as called "Basic View" would contain only the basic needed tabs to create options, beliefs, facts, define scenario preferences in Argue Table and to execute/simulate scenarios. The advanced view would include all basic tabs as well as tabs where the definition of the arguments, resolving conflicts between arguments take place and the tab that contains the content of the Prolog file generated.

Final Stage

After the implementation of the web pages, the real-time evaluation was also done by browsing the web pages. Rating was made by three users. And in this case the choice was not accidental. The first evaluation was made by a user interface developer with extensive experience in web design and development. His thoughts moved around the aesthetics of the pages, their colors and functionality. In particular, he highlighted the need for a new "Back to Projects" button, which after clicking it, will redirect to the projects page in order to navigate to another user created project. Also, he reported that the collapsed input that appears when a user clicks to expand an already defined scenario is not visible enough and confusing to focus. So, he proposed to add an outline with a color that will help users to focus on it and not distract them. The outline will cover both the scenario that will be expanded and the preview of the scenario that will be created.

The second evaluation was made by a student who focused on the application at mobile phone. His thoughts moved around some changes in the layout of the pages, so that page content to fit exactly on mobile phone screens. Furthermore, he insisted to correct the alignment of the text input forms, so that to be relaxing and easy for each user to fill them.

The third and last rating was made by an expert user in argumentation and Gorgias framework. His thoughts however, helped to a large extent to examine the simplicity of implementation and the effectiveness of its functionality. He proposed to add one more functionality to the system. After extended use of the developed system, he observed that in some cases the scenarios that are generated automatically from the combination of other scenarios that have conflicting options, may have in their context facts or beliefs that re objectively impossible to happen. To overtake this situation, he proposed to add a button to define such scenarios as impossible. After the definition of a scenario as impossible, this scenario should be hidden from the Argue Table View, as well as from execution page. But, in order to remove a scenario from the impossible state, a custom view to display only the impossible scenarios should be implemented.

Furthermore, last user noticed that some help messages and some labels on form inputs were misleading by mistake and thus he tried to give advice to make them as clear as possible. Last but not least, he proposed to add all pages of the developed application floating

notifications that will inform users about the progress of their requests or about actions that they perform. Thus, in every page were added notification messages in floating boxes where each box, according to the progress of each action/request have different colors. Green color for successfully completed requests and red color for unsuccessful requests or errors. All these notifications windows include a custom explanatory message according to the progress and the page that belongs to.

After the necessary corrections were made and at this stage of implementation they were again asked if they were satisfied and the answers were all positive. The developed application's database service was implemented with a NoSQL family database system. The open-source MongoDB database was selected. The MongoDB storage unit is the Document. A record on MongoDB is equal to a Document. Document is a data structure consisting of pairs of fields and corresponding values. It is similar to a JSON object, and their field values can contain all supported JSON types, other Documents, tables or Document Tables as well as some other types, such as Date, Timestamps, ObjectId.
The advantages of a Document approach are the below:

The advantages of a Document approach are the above:

- Documents can be easily represented in each programming language with the data types available to each one.
- Integrated Documents and Tables reduce the need for expensive joins.
- Documents are not static and can be changed at any time , thus flexibly supporting polymorphism and giving developer flexibility in creating the database structure.

Database Structure

MongoDB is used to store 8 types of Collections:

- **User Data** In this Collection, User Data are stored which arised from the application's signup process. The data that are stored are the username that must be unique, the password which is stored encrypted, the name of the user, as well as user's email which is used to activate new user's account or if the user forgot the password.
- **Project** Each Project is defined by it's unique ObjectID, by it's name, description, date created and modified and by the userID that created it.
- **Option** Each option has a unique ID, a name, parameters and the project's ID which belongs.
- **Complements** Complements are defined as a pair between options and the project's ID which belongs.
- **Fact** Fact's basic fields are the name of the fact, it's parameters and the ID of the project it belongs.
- **Belief** Belief share the same fields with Fact, with an addition of a boolean field whether a belief is considered as abducible.
- **Rule** Rules are the most important part of our structure. Rules are composed of their unique ID, the project to which they belong, optionHead or beliefHead accordingly which of these two the rule refines, the lowest priority rule that defines it, the level and the highest rule in priority.

- **Impossible Scenarios** In this Collection, info about the scenarios that have been defined by Rules which are impossible to take place in a real-life application, is stored.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The objective of this senior thesis is to implement web-based decision policy definition and simulation application for the Gorgias argumentation framework in a modern programming language that performs well to users and to create a web interface of this so it can be accessible by the public. The software was designed to incorporate cutting-edge technologies and programming frameworks, handle a large volume of transactions and be compatible with both desktop and smart mobile phones and mobile devices.

This senior thesis demonstrated the novelty of a decision policy definition and simulation in Spring-Boot and Angular in with a simple-to-use web interface. At the web interface, user can model real-life problems, create scenarios, expand scenarios or refine them. Visualization of the scenarios is also available in the web interface. The implementation specifics of this project, including front-end, back-end, unit testing and deployment, were also discussed in this thesis.

Moreover, usability testing has taken place during the evaluation process, which uncovered usability issues to the web's interface user friendliness. These issues that are described in evaluation process, firstly have been analyzed and then have been resolved according to user's feedback.

Furthermore, particular emphasis was given at the design of the application pages, having as a goal that each page is fully understandable and simple for the users, regardless of the age and the experience in such web applications. This entails creating a simplified web interface for inserting data, as well as for reviewing the results.

Developed application differs from the other corresponding implementations, such as Gorgias-B, because of the integration of 4 basic and innovative functions.

- Visualization of all the available scenarios in a table view (Argue Table). The user is given the option to display all the available scenarios that have been created accompanied with the corresponding options at each scenario. In this way, user can expand the scenarios, edit his choices or delete specific scenarios.

- Priority definition between the options of each scenario. User has the ability to define selected options as default, giving that bigger priority on these default options over the options at each particular scenario.

- Definition of a scenario as impossible. User, at will, can set a whole scenario except from the basic scenarios as Impossible. In this way, these impossible scenarios are not

displayed at the argue table with the rest of non-impossible scenarios. Anytime, user can also review the scenarios that have been set by him as impossible and set them again as possible.

- Implementation of a natural language processing tool at form inputs. During the insertion of the available options, user has the ability to insert them in free-text form to the system. In background, system makes natural language processing with syntactical and word analysis. To achieve it, Stanford's Core NLP language processing tool. SO system (takes) user input from the form and converts it to a form that's appropriate for the background system.

In conclusion, through the use of the system concerns and needs have been arise, which can be covered in future version of the application. These are described thoroughly in the future work section below.

7.2 Future Work

The system developed in the context of this senior diploma thesis, is a product of both systematic design of the multiple units that comprise it and of its intense implementation at programming level. In this context, it was not possible to use the system as a productive tool than to present it as a prototype base on which to base future work and future extensions.

Systematic work with this system and its operation in decision policy definition and simulation, set the stage for some thoughts that contribute to its extensions, both functional and technological.

7.2.1 Natural Language Processing expansion

Natural language processing and conversion to Prolog predicate[34], could be expanded at future implementations of this thesis to support multi line sentences at inserting options, facts and beliefs, or to enter free-form text to describe scenarios and then the system to map it to appropriate system's data structure. This whole process could be more efficient and attractive, if user could also insert information via voice input. To be more specific, user could insert not only options, facts and beliefs as a free text but also define scenario preferences and priorities between options.

7.2.2 Automatically recognition of complimentary contexts

Another useful and time saver expansion of the system, would be the existence of a database filled up with complimentary contexts (day -> night), so that combinations of already created scenarios that have these complimentary contexts combined, can be calculated as Impossible scenarios automatically. An example of this is that a combined scenario refer to summer and winter at the same time which is impossible to happen.

7.2.3 Custom Scenarios View

Argue Table view was highly rated by users who evaluated the developed application. Despite that, in order to be handier and more configurable, an additional view could be implemented. A custom view where each subset scenario, to be listed with its father scenarios, or another view which lists the initial scenarios, and each one has as sub list their children scenarios.

7.2.4 Execution Results explanation

Gorgias framework, right after execution provides an output, which results from the given input file. This output, is an explanation why each specific option is available and which defined arguments take part at each decision. The current explanation, is based on listing the translated arguments from Prolog. A possible future expansion of this, could be the display of the explanation in a more user-friendly way, maybe in English sentences, grammatically and syntactically correct. This could lead to make the explanation clearer, more comprehensive and without any doubt about its meaning.

7.2.5 Collaboration with other users

Last but not least, in many cases users work together as a group their projects or share their works. It would be nice feature to allow groups to collaborate at each project, to edit it simultaneously and interact with it in real time.

References

[1] A. C. Kakas, P. Moraitis, and N. I. Spanoudakis, "Gorgias: Applying argumentation," *Argument & Computation*, no. Preprint, pp. 1–27, 2019.

[2] N. I. Spanoudakis, A. C. Kakas, and P. Moraitis, "Applications of argumentation: The soda methodology." in *ECAI*, 2016, pp. 1722–1723.

[3] N. K. Janjua, *A Defeasible Logic Programming-Based Framework to Support Argumentation in Semantic Web Applications*, 1st ed., ser. Springer Theses. Springer International Publishing, 2014.

[4] F. H. Van Eemeren, R. Grootendorst, and F. H. Eemeren, *A systematic theory of argumentation: The pragma-dialectical approach*. Cambridge University Press, 2004, vol. 14.

[5] N. I. Spanoudakis, E. Constantinou, A. Koumi, and A. C. Kakas, "Modeling data access legislation with gorgias," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2017, pp. 317–327.

[6] E. Karafili, A. C. Kakas, N. I. Spanoudakis, and E. C. Lupu, "Argumentation-based security for social good," in *2017 AAAI Fall Symposium Series*, 2017.

[7] F. Cloppet, P. Moraitis, and N. Vincent, "An agent-based system for printed/handwritten text discrimination," in *International Conference on Principles and Practice of Multi-Agent Systems*. Springer, 2017, pp. 180–197.

[8] N. Spanoudakis and P. Moriaitis, "Engineering an agent-based system for product pricing automation," *Engineering Intelligent Systems*, vol. 17, no. 2, p. 139, 2009.

[9] K. Pendaraki and N. Spanoudakis, "Portfolio performance and risk-based assessment of the portrait tool," *Operational Research*, vol. 15, no. 3, pp. 359–378, 2015.

[10] N. I. Spanoudakis, A. C. Kakas, and P. Moraitis, "Gorgias-b: Argumentation in practice." in *COMMA*, 2016, pp. 477–478.

[11] H.-P. Lam and G. Governatori, "The making of spindle," in *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2009, pp. 315–322.

[12] "Gorgiasb tool." [Online]. Available: http://gorgiasb.tuc.gr/

[13] P. M. Dung, "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games," *Artificial intelligence*, vol. 77, no. 2, pp. 321–357, 1995.

[14] A. Kakas and P. Moraitis, "Argumentation based decision making for autonomous agents," in *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. ACM, 2003, pp. 883–890.

[15] B. Mehta, *RESTful Java Patterns and Best Practices*. Packt Publishing, 2014.

[16] P. Teixeira, *Professional Node.js: Building Javascript Based Scalable Software*, 1st ed. Wrox, 2012.

[17] A. Mardan, *Practical Node.js: Building Real-World Scalable Web Apps*, 1st ed. Apress, 2014.

[18] "V8." [Online]. Available: https://v8.dev/

[19] J. Vepsalainen, *Webpack and React: From apprentice to master*, 2017.

[20] R. Johnson, *Expert One-on-One J2EE Design and Development*. Birmingham, UK, UK: Wrox Press Ltd., 2002.

[21] C. Walls, *Spring Boot in Action*. Manning, 2015.

[22] C. D. Brad Dayley, Brendan Dayley, *Node.js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications (Developer's Library)*. Addison-Wesley Professional; 2 edition (November 2, 2017).

[23] A. Freeman, *Pro Angular 6*, 3rd ed. Apress, 2018.

[24] [Online]. Available: https://angular.io/guide/quickstart

[25] M. D. Kristina Chodorow, *MongoDB: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.

[26] "Docker overview," Jan 2019. [Online]. Available: https://docs.docker.com/engine/docker-overview/

[27] V. S. Pethuru Raj, Jeeva S. Chelladhurai, *Learning Docker*. Packt Publishing, 2015.

[28] C. Christodoulakis, "A rich media mobile web application for visitors and the community of the technical university of crete," Master's thesis, Technical University of Crete, 2012.

[29] "Stanford corenlp – natural language software." [Online]. Available: https://stanfordnlp.github.io/CoreNLP/

[30] H. J. La and S. D. Kim, "Balanced mvc architecture for developing service-based mobile applications," in *IEEE International Conference on E-Business Engineering*. IEEE, 2010, pp. 292–299.

[31] F. Dushin. [Online]. Available: http://www.swi-prolog.org/packages/jpl/java_api/index.html

[32] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P/P14/P14-5010

[33] M. Madison, M. Barnhill, C. Napier, and J. Godin, "Nosql database technologies," *Journal of International Technology and Information Management*, vol. 24, no. 1, p. 1, 2015.

[34] T. Mitsikas, N. I. Spanoudakis, P. S. Stefaneas, and A. C. Kakas, "From natural language to argumentation and cognitive systems." in *COMMONSENSE*, 2017.