university of groningen

university college groningen

Reengineering and Extending a Node-Based Web-Based Communication System for Summer Schools and Other Short Duration Conventions: A Case Study

Bachelor Thesis

Nikolas Hadjipanayi

Faculty of Mathematics and Natural Sciences
University of Groningen

19. July 2018

Supervisors:

Dr. Mircea Lungu

Dr. Vasilios Andrikopoulos

Abstract

Most short duration conventions and schools have no convenient and centralized way of communication between attendees and the coordinators. A system attempting to solve this was developed last year by a team of software engineering students on the request of the department of Summer and Winter schools of the University of Groningen. When systems like this get larger and they lack a clear architectural design, they degrade to an unmanageable state and adding more functionalities to them becomes challenging. The already existing solution to the problem was lacking some critical functionality according to initial pilot tests that run with one summer school. During the period of our research, we dealt with the maintenance cycle of the existing system, refactoring parts of the code that were overly complicated and extracting a clear architectural design that will help with maintainability. Research on software maintainability revealed that a high-level architecture for a system improves maintainability allowing new developers that are tasked with maintenance to be more efficient[1]. We validated our work by observing user interaction with the system and their feedback.

Contents

1 Introduction

Short-lived social events are becoming increasingly more popular, companies often choose to send their employees to seminars or other work-related events. In addition to that, universities and schools often organize events, such as summer schools, where they invite people from all over the world to participate. The large scale of these events on top of the fact that often times the attendees are unfamiliar with each other leads to a gap in communication. This creates a need for an application that will enable group communication for people without them needing to share any personal information. Larger institutions choose to develop their own applications that are proprietary to solve this. In contrast events with not as many resources end up depending on social media groups like the ones Facebook and Google+ provide or even e-mailing lists. The above solutions are not ideal since they require all of the participants to either have a social media account or an e-mail they are willing to share. In the current era of constant security threats and information leaks, people are not as willing to do so.

During the 2017 course of Software Engineering, the department of summer and winter schools of the University of Groningen brought the aforementioned problem to the attention of the administrators of the course. They requested the creation of a communication application that will enable them to have a centralized place where their attendees can refer to when they need information regarding their school or get to know each other.

As per the external client's request, such a system was created by a team during the course. The system was built using many different technologies and tested with a

single school during the summer of 2017 summer school period. The system included a back-end service that would serve as a RESTfull API on which mobile application and web applications can be based upon. Besides that, the system included a management web application that provided the administrators with a user-friendly way to manage the content of schools. The pilot testing mentioned, revealed a lot of flaws in the system. The initial elicitation of the requirements was inaccurate resulting to the clients needing extra functionalities that were not implemented. This was proven to be extremely hard to achieve due to the lack of a clear architecture of the system. Although a software design is supposed to be used as a blueprint for systems this was not the case for the system in question. It is possible though to extract the architecture of a system based on its implementation in order to benefit from the increase in maintainability[1].

Structure of this Thesis The structure of the remainder of this document is the following:

In Chapter 2 (Related Work), the general research field is described together with descriptions of how others tried to solve similar problems.

In Chapter 3 (The problem statement), the problem statement is clearly defined and why it is important to be solved.

In Chapter 4 (Reverse engineering), the process followed to document and extract the architecture of the existing system is described.

In Chapter 5 (Refactoring and extending), the work on refactoring and extending the system is described.

In Chapter 6 (Evaluation), the feedback from the users of the system is presented.

In Chapter 7 (Discussion and future work), the future work on the project is described, followed by discussion and some concluding remarks.

2
Related Work

2.1 Software maintenance

Software maintenance is the process of the software lifecycle that is responsible for fixing errors or extending the system. Early work on the subject of software maintenance has shown that, although this part of the cycle is often overlooked, it consumes a considerable amount of the resources of a project[2].

2.2 Reverse Engineering

Reverse engineering in the context of software is a transformation process that is inverse to the traditional course of software development. That is, given an implementation of a system the design and functional requirements can be deduced[3]. The goal of reverse engineering is to improve the maintainability of a legacy system, thus making it cheaper to maintain and extend[4]. There has been considerable research and advances in the field of reverse engineering in the past years. The majority of the research focuses on retrieving the code of a deployed application and security, with little to no work done in redocumenting systems build only with JavaScript within NodeJS.

2.3 Software architecture

Software architecture is a subcategory of software engineering that is responsible for the documentation of the different parts of a system and their relations. The elements can be data objects or processes that transform them[5]. The obtained structure, referred to as the architectural design, illustrates the high-level design decisions of a system and their interactions. In addition to that, the architectural design should include sufficient information to allow high-level analysis of the system. That improves its maintainability and by extension the ability to expand the system with new functionalities[6]. In the past, there have been approaches towards automatic architecture recovery but none of them work for JavaScript and Node based systems[7][8].

3
The Problem Statement

3.1 System functionalities

The legacy system consisted of the following functional requirements:

- Announcements: the ability to post, edit and delete announcements. Mandatory information is poster name, date, title and text of the announcement.
- General information: the ability to post, edit and delete general information. This type is usually added before the school starts and contain information like accommodation and payments. Mandatory information is date, title and text of the general information.
- Lecturers: the ability to post, edit and delete lecturers. Lecturers are the people that are in charge of activities or give talks during the school. The mandatory fields are name, a description an image of the professor and a link to a personal website.
- Schedule: a schedule for adding events. An event needs a title, a location, the details and a starting and ending date.
- Forum: a forum where attendees can discuss. The forum has 2 components threads and comments to those threads.
- School specific: Schedule, announcements and the forum are school specific. This gives the ability to run multiple schools on a single server instance.

- School login code: Each school needs a login code. The login code is used by the mobile/web applications that are built on this system to delegate users to the appropriate school.
- User types with privileges: Multiple user types with different privileges. This allows us to restrict school coordinators from performing operations like user management.
- School management: Add edit and delete a school. A school has a name and a starting and ending date.
- Web application: The server, that all of the above are implemented on, must serve a web application that helps administrators and coordinators to manage content.
- RESTful API: All of the above resources must be accessible through a RESTful API. This will enable mobile applications and web applications to be build based on this system.

Besides the functional requirements, there are also some important non-functional requirements.

- Security: The system must be secure keeping passwords encrypted.
- Privacy: The personal information stored must be kept at a minimum.
- Scalability: The technologies used must be scalable.
- User-friendliness: The content management web application that comes with the server must be intuitive to use.

3.2 Problem Statement

The system in its current state is unfit to be used in practice due to multiple issues discovered during the pilot testing, including critical functional requirements that were not considered in advance. Besides that, the whole system is missing a clear architectural design leading to the low maintainability of the system. To address this issues the system needs to be:

- Reverse engineered: A high-level architectural design of the system needs to be extracted. On top of that, the technologies of the system, as well as the complete functionality and restrictions, need to be documented.
- Refactoring: The multiple extensions of the system in the past rendered it increasingly complex requiring refactoring to bring it back to a maintainable state.

- Extending: The project owners requested that the resources of general information and lecturer be associated with a school and not be global like they used to be. Extending on that, they also requested the ability to post global general information that will be visible to all schools but the coordinators would not be able to modify. Furthermore, the system did not provide the users with adequate feedback when something went wrong e.g an announcement did not contain a title something that was frustrating to the users. That needs to be addressed.

4
Reverse Engineering

4.1 Reverse engineering

According to the paper Reverse Engineering and Design Recovery: A Taxonomy written by Elliot J. Chikofsky and James H. Cross in 1990. Reverse engineering is the process of analyzing an already implemented system in order to extract higher-level information along with the representation of the system's components and their relations. The paper continues to explain that the goal of reverse engineering is to increase the comprehensibility of the system in order to increase maintainability and the ability to extend the functionality. This goal is closely aligned with the objective of this project, thus many of the techniques suggested in the paper were also used in our process of documenting the system. Namely, the authors suggest the generation of alternative views and the recovery of lost information about the system[3].

4.2 Technology Stack

Before documenting the existing system on a high level, it is important to talk about the technology stack used to implement it. The technology stack used for the development of the legacy system was the MEAN stack. MEAN is an acronym that stands for MongoDB, ExpressJS, AngularJS, and NodeJS.

4.2.1 MongoDB

MongoDB uses collections of documents that are JavaScript like objects. The JavaScript nature of the database allows the use of the JSON format without conversion, this is beneficial in the case of an API that deals with JSON objects. MongoDB gained a lot of popularity in the database community in recent years, that is mostly due to its flexibility in production. Besides that MongoDB has built-in support for horizontal scaling and it is magnitudes faster than traditional relational databases when dealing with a sufficiently large amount of database operations. However when a well-structured database with a lot of relations between the tables and collections is desired it is best to choose one of the well established and supported database solutions like Oracle.[9].

4.2.2 Node.JS

Node JS is a JavaScript runtime environment built on Chromes V8 JavaScript engine. NodeJS uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. These qualities allow for rapid development of highly scalable and fast applications[10]. Furthermore, the package ecosystem that Node JS uses, NPM, is the largest ecosystem of open source libraries in the world[11].

4.2.3 ExpressJS

Express is a framework built on the already existing capabilities of Node, it provides a higher level of abstraction allowing for faster and safer development. Express includes straight-forward and easy to read/write routing that makes projects easier to manage [12]. In addition to that, the Express community offers a plethora of middleware. Middleware, commonly referred to as middleware stack, are actions that can be performed before or after a request is handled, common tasks for middleware can be database connection, logging, security and so on. This extra layer is what makes express so powerful, as different developers can load or create the appropriate middleware for their unique needs.

4.2.4 AngularJS

AngularJS or Angular is an open-source framework that is used for running client-side code. Developed and maintained by Google, Angular has risen in popularity over alternatives like JQuery. The many features provided by the framework provide an easy way to create applications based on the well documented and supported Model View Controller(MVC)[13]. Angular provides the ability of two-way binding and automatic view render, allowing the programmers to focus on the logic of the application. In addition to that, built-in directives allow the developers to extend traditional HTML tags with the possibility to create custom directives that fit the application's needs.

4.2.5 Mongoose

Although not a part of the MEAN architecture Mongoose plays an integral role in the system. Mongoose acts as the connecting layer between the NodeJS and MongoDB. Mongoose is schema-based, meaning that every collection in MongoDB (similar to traditional tables) has a structure. Mongoose exposes an API for creating new instances of schemas (called documents) and inserting them into the appropriate collection. Besides that, the schemas offer built in support for validation.

4.2.6 Markup and styling

For markup and styling, the choice was HTML and CSS. The styling was enhanced by Bootstrap a toolkit that includes multiple pre-built components among other tools helping to create responsive, modern looking and cross-platform websites.

4.2.7 Benefits of using MEAN

The MEAN software bundle has many benefits over its alternatives, the fact that the programming language used throughout the whole stack is JavaScripts can greatly increase maintainability and the speed of development for an application. JavaScript Object Notation (JSON) is a popular choice for the encoding of data in web applications. Using JSON in combination with MEAN and ExpressJS comes seamlessly due to the fact that there is no need for parsing since they are of the same format[12]. The described behaviour can be illustrated in the Figure 4.1 which is a code snippet from the Summer and Winter schools application. In the figure, we can see how the post request for an announcement is handled, notice how the body of the request is saved exactly as it is received in the database without any conversion. Besides that, in the code we can also see the power of the ExpressJS middleware before anything is saved the request is checked for sufficient privileges and if something is wrong the error can be passed to the error middleware. If the saving is processed without any error ExpressJS responds with a redirect to the page that made the request effectively refreshing it. Mongoose validates all the resources that are inserted into the database, the way to specify the requirements of the validation can be seen in Figure 4.2. The above examples showcase how seamless the transitions between the layers of the application are.

```
router.post("/API/announcement", auth.isAuthorised("ALTER_ANNOUNCEMENTS"), function (req, res, next) {
    new Announcement(req.body).save(function (err) {
        if (err) {
            err.shouldReload = true;
            err.status = 400;
            next(err);
        } else {
            res.redirect(req.get("referer"));
        }
    });
});
```

Figure 4.1: Posting an announcement using MEAN

```
const mongoose = require("mongoose"),
    Schema = mongoose.Schema;

const AnnouncementSchema = new Schema({
    title: {
        type: String,
        required: "Announcements require a title",
        trim: true
    },
    created: {
        type: Date,
        default: Date.now
    },
    description: {
        type: String,
        required: "Announcements require a description",
        trim: true
    },
    poster: {
        type: String,
        trim: true
    },
    school: {
        type: Schema.ObjectId,
        required: "An announcement needs to be associated with some school."
    }
});

mongoose.model("announcement", AnnouncementSchema);
```

Figure 4.2: Mongoose schema declaration

4.3 High Level Architecture

From the analysis of the technology stack of the system the relations between the stack and the connecting pieces are observed and documented in Figure 4.3. In the figure, we can see an instance of the system and how the components are interconnected. The server side that runs on Node has the main components that support the functionality of the system. Express runs within the environment of Node and handles the routing of the system. It is also responsible for the communication with the components that handle the authentication and the exceptions. In addition to that, the route handling component also serves the Angular code and the views to a clients browser where they are rendered. The Mongoose component also runs on the Node instance and communicates with the route handling and the instance of the database running independently on the physical server machine. This facilitates the creation and exchange of data between the database and the instance of the system. Using the figure we can observe that there is a straight line of communication starting from the content management system (client side) towards the database with the components within Node facilitating the exchange of data based on some restrictions.

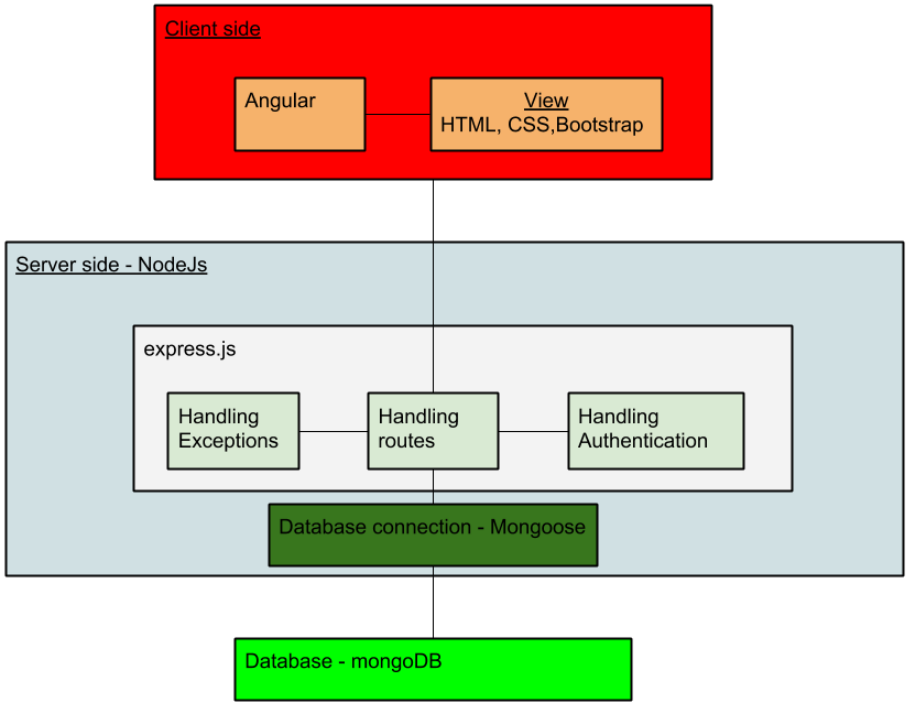

Figure 4.3: High level architecture arrows indicate communication between components

4.4 Restful API

To document the behaviour of the API we start off by analyzing the system code responsible for the endpoints. By observing the code we were able to deduct the table at Figure 4.4. In the figure, we can see a list of the resources the path where they can be accessed and all the fields they include. Closer inspection of the system revealed that all of the endpoints support the standard creation, retrieve, update and delete requests. Different HTTP requests can be used for each operation:

- POST - create resource
- PUT - update resource
- GET - retrieve resource
- DELETE - delete resource

The system adheres to the principles of REST[14].

Figure 4.4: Restful API

Resource	Path	Mandatory fields	Optional fields
Announcements	/API/announcement	title, description, school	created, poster
General information	/API/generalinfo	title, description, category, school	created
Lecturers	/API/lecturer	name, description, website, school	created, imagepath
Forum-thread	/API/forum/thread	title, description, author, posterID, school	created, imgURL, comments
Forum-comment	/API/forum/comment	text, author, posterID, parentThread	created, imgURL
School	/API/school	name, startDate, endDate	created
User	/API/user	username, password, rank	school
Login Code	/API/logincode	code, school	created

4.5 User rights and security

During the analysis of the endpoint function definitions, like the one in Figure 4.1, we were able to identify and document the restraints on operations of the system. The above code, before proceeding with the fulfillment of the request and the return of the resources, checks whether the request is authorized. Some of the requests could only be performed through the content management web interface and after the user has logged in with an administrator account and others were performed by the coordinator type account. Analyzing the files responsible for the authentication aided in the extraction of Figures 4.5 and 4.6. In the figures, we can see which endpoints can only be accessed through requests from the web interface and which endpoints can be accessed by everyone.

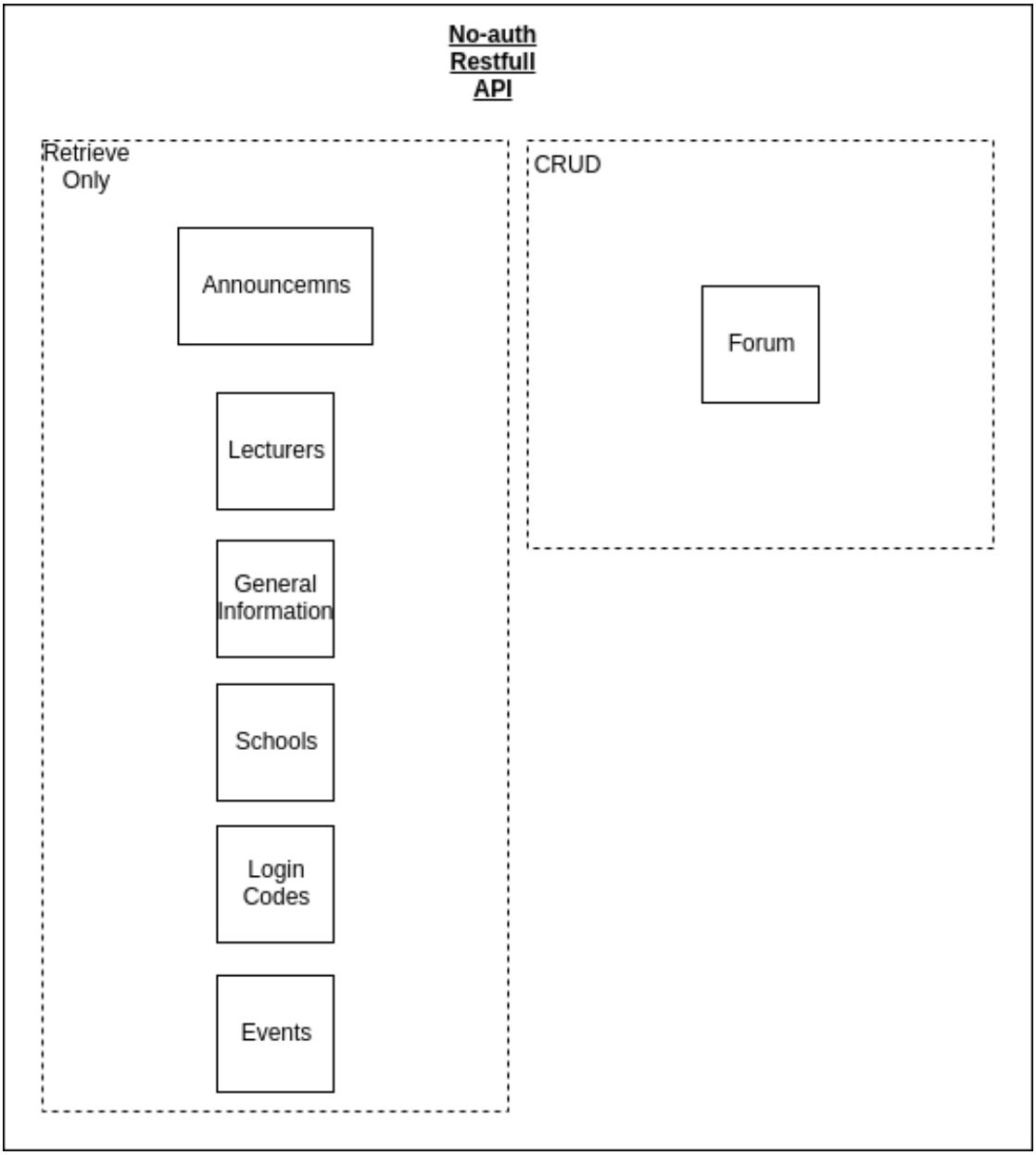

Figure 4.5: Calls to the endpoints requiring authorization

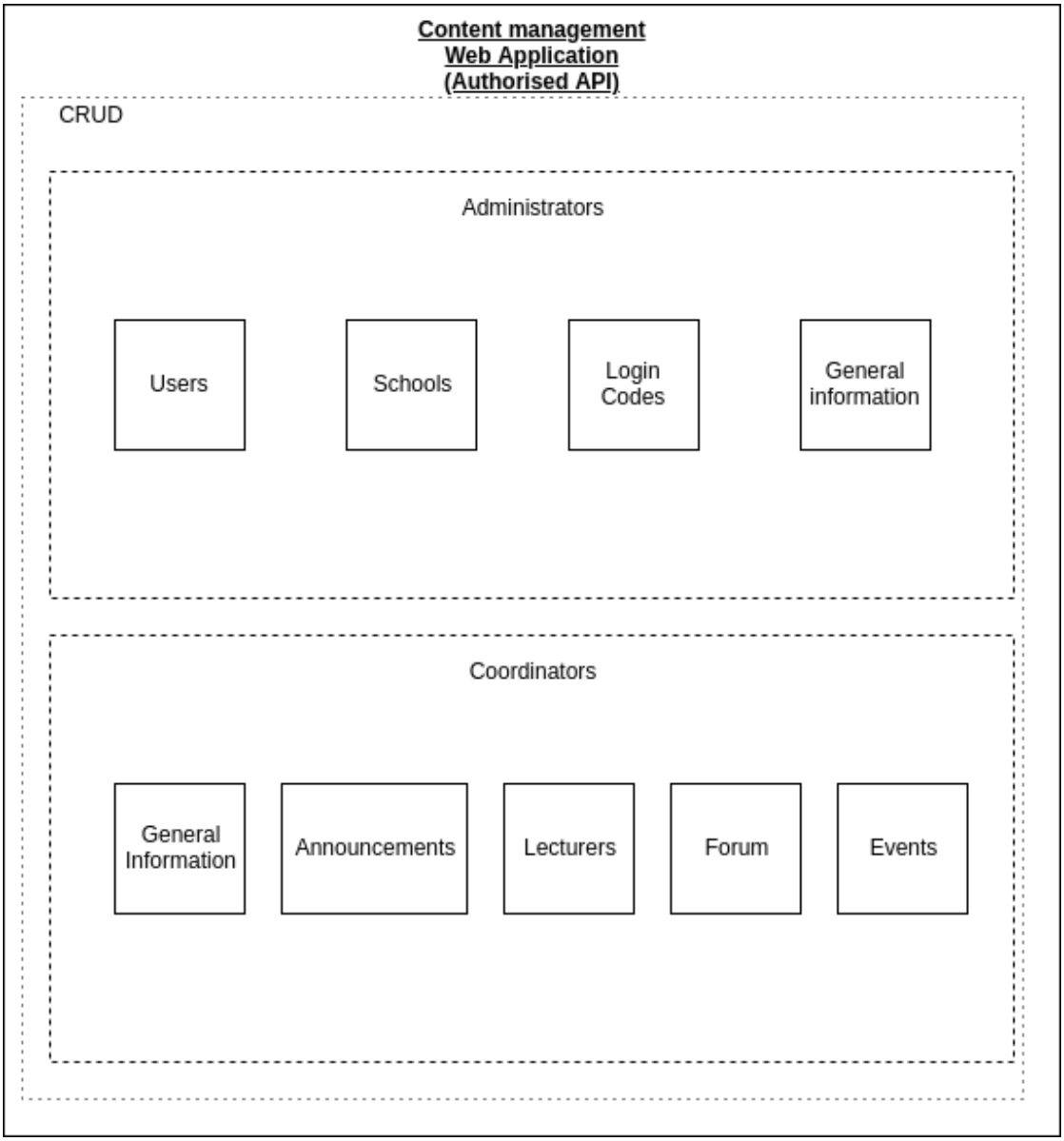

Figure 4.6: Calls to the endpoints not requiring authorization

5 Refactoring and Extending

5.1 Refactoring

Refactoring is the process of altering the code of a system in order to improve its design without changing any functionality. The prime goal of refactoring is to eliminate code smells[15]. In our overview of the system, 2 code smells were identified that needed immediate addressing.

5.1.1 Unified error handling

The system did not have a unified error handling. Whenever an error was thrown during the endpoint processing there was a separate console statement to log the error and a hand-written error message was returned to the user. This creates many problems when there is a need to change the code, for example, if there was a need to use a different logging method than simple console logs the developer would have to manually change all the concerning code. Using the middleware capabilities of ExpressJS a custom middleware was created. Now, whenever an error occurred the endpoint simply passed control to the error handler and the error handler was responsible for logging and responding to the user with the appropriate error code and message. This opened up the possibilities for a more uniform error reporting with more accurate error messages that were provided by the component that had thrown the error.

5.1.2 Migrating from JQuery to Angular

At some point, the system transitioned from using JQuery to AngularJS. JQuery was not completely eliminated from the system though, several of the views still depended on it and a complicated mixture of both AngularJS and JQuery still existed. The more simple cases of this were solved by fully refactoring the functionality to Angular. However, some more complicated issues would require more time than what is available for this project. In Figure 5.1 we can see how introducing Angular directives can declutter the code and reduce the lines of code considerably. The code on the right shows that a simple built-in directive for pagination replaces a whole file of JQuery code that handled the same pagination functionality used in several of the system's pages.

Figure 5.1: Replacing JQuery code with Angular to achieve pagination

5.2 Extending the functionality of the System

5.2.1 Schools specific resources

The resources of lecturers and general information were changed to be school specific. To achieve this the schema files of mongoose were altered to include an id that will associate the resource with a school. The full schema files can be found in the GitHub page of the project here. The retrieval operations of the system already supported filtering of results so nothing had to change in that regard. The maintainers of other applications that depend on the system had to be contacted so they can change their HTTP requests to include a school filtering for the altered resources.

5.2.2 Global general information

For this extension we introduced a virtual global school that was represented with a special school identification number. General information posted by administrator accounts received that special ID and coordinators were restricted from modifying them. Figure 5.2 shows an example of a global general information in its JSON format, the school ID *000000000000000000000001* is the ID representing the virtual global school. A crucial part of the process was to ensure that the API would be backward compatible so any applications based on it would not break. To achieve that the GET request for the general information was altered to serve the JOIN between the general information associated with the requested school in addition to the global general information.


```
{
  "created": "2018-07-09T09:49:19.919Z",
  "_id": "5b432f9fd8e5132b35774187",
  "title": "Instagram Challenge",
  "description": "Here is the chance to win a University of Groningen Hoodie: Post a boomerang of your summer school",
  "category": "Info",
  "school": "000000000000000000000001",
  "__v": 0
}
```


Figure 5.2: Global general information

5.2.3 Providing errors to users

Due to our refactoring of the error handling and migrating to Angular, we were able to provide errors to the user using the error callback function of the Angular HTTP request to push an error that was displayed through a Bootstrap alert at the top of the page. This process can be seen in Figure 5.3. This additional functionality was only implemented for the options page dealing with administrative management. The rest of the pages were too heavily dependant on legacy JQuery code for their HTTP requests. Introducing this

to all of the pages must be delayed for when the complete overhaul of the front end is done.

```
$http.post(path, toAdd).then( function (){
    window.location.reload();
}, function (error) {
    $scope.addAlert("danger", error.data);
});
```

Figure 5.3: Displaying an error to the user

5.2.4 Dealing with Downstream

One of the biggest challenges of software evolution is keeping up with the downstream [16]. This was particularly evident in the case of this system. At the time that the above changes were implemented the summer school season started and making drastic changes to the system was not an option as it would jeopardize the stability of the system. For that reason, we chose to leave non-essential complicated changes like the revamp of the front end for the offseason where the system is not as heavily used.

6 Evaluation

6.1 User Evaluation

To validate our work, we offered to the users of the system a 30-minute training session followed by a few questions. The whole process was:

- A brief explanation of the system functionalities followed by a tutorial on how to use them.
- The users then interacted on their own with the system trying to perform all the actions necessary.
- Observe if they had any problems with the usability of the system.
- Ask them if they would change something in regards to the design.
- Ask them for any functionality that they think would be essential for the system that is not currently present.

This way we could observe how normal users interact with the system and extract information that they would otherwise be unable to provide through a standard questionnaire. We trained 10 of the summer schools' coordinators and through discussion, we were able to identify some problems and future work for the project. All the users were able to navigate through the system after their initial brief training with ease, they were also satisfied with the aesthetics of the system. One problem that was identified was that the users had difficulty navigating back to the overview page of the system. To get

back to the overview the user had to click on the logo of the system on the upper left corner. When we asked users regarding that, they mentioned that they would prefer an overview button close to the navigation bar or an indication next to the logo. With the exception of that issue, all the coordinators were able to use all the functionalities of the system without asking us for further instructions. 10 out of 10 of the users told us that besides the overview page button, they wouldn't change anything on how the system looks and feels. However, some extra functionalities were requested. When asked 3 out of 10 users requested integration with Dropbox or Google drive. They remarked how they use those services in their schools extensively and since they would be using the application anyway, it would be nice to have all of their work into one service. In addition to that, 2 out of 10 users asked for the ability to attach files like PDF or word documents to announcements. They mentioned that it is an essential functionality of other communication applications that they use and it is particularly useful to them. One of the users suggested that we also create a web application where attendees can use. Currently, the application is only available through IOS and Android for attendees. The user mentioned that some of his older participants are having a hard time with mobile applications and they prefer to use their laptops because of larger letters and the ability to zoom into pages.

6.2 Refactoring and Reverse Engineering

Due to time constraints, we were unable to empirically evaluate the maintainability of the new code. It is important to note that the already existing applications that depended on the system continued to work properly.

7 Discussion and Future Work

Summarizing, we were given an already implemented legacy system that was no longer useful to the customers, the system was unmaintainable with missing functionality. Throughout the duration of this project, we were able to bring the system back to a maintainable state using the process of reverse engineering. Following that, we used the extracted documents that we produced and we were able to refactor and extend the system with the critical functionalities that it was lacking. As a direct result of our work, the system was used by 10 schools and we received encouraging feedback regarding its usefulness and functionality.

7.1 Discussion

During the work described above, we had some setbacks that we would like to draw attention to. First and most importantly reverse engineering is not a trivial process. Analysis of the system is required on top of experience with the programming language and the frameworks used in the system. That is especially hard when the team trying to reverse engineer is not the one that developed it. In our case, we were fortunate enough to have a direct line of communication with the initial development team that allowed us to resolve a lot of questions we had regarding the code in an efficient way. Although we were able in this case to extract some meaningful architectural documents of the system, we would not recommend the use of reverse engineering in the place of careful initial planning. When possible initial planning and documentation should take place before the system is developed so problems like our collision of technologies can be avoided.

Besides that, all the problems we faced with keeping downstream changes to a minimum were also a hindrance to our work.

7.2 Future Work

Despite the system being in an acceptable state with schools using it, we believe that there is still much to improve on. At first opportunity, the front end of the system needs to be rewritten in full Angular. That will ensure a system that is more maintainable and can be more easily extended. Besides that, there are requests from the users for additional functionalities that need to be considered. The documentation we extracted is sufficient for an elementary understanding of the system but the system still lacks thorough documentation at the network and database levels. Finally, we would like to explore the possibilities of using the system in other conventions besides the summer schools that the system currently supports.

[1]The source code of the project can be found at:
https://github.com/RUGSoftEng/2017-Winter-Summer-School-App/

Bibliography

[1] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: Its extracted software architecture," in *Proceedings of the 21st international conference on Software engineering*. ACM, 1999, pp. 555–563.

[2] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[3] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns*. Elsevier, 2002.

[5] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992. [Online]. Available: http://doi.acm.org/10.1145/141874.141884

[6] D. Garlan, "Software architecture: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 91–101.

[7] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri, "Revealer: A lexical pattern matcher for architecture recovery," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 170–178.

[8] M. Lungu, M. Lanza, and T. Girba, "Package patterns for visual architecture recovery," in *Conference on Software Maintenance and Reengineering (CSMR'06)*, March 2006, pp. 10 pp.–196.

[9] A. Boicea, F. Radulescu, and L. I. Agapin, "Mongodb vs oracle–database comparison," in *Emerging Intelligent Data and Web Technologies (EIDWT), 2012 Third International Conference on*. IEEE, 2012, pp. 330–335.

[10] I. K. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Is node. js a viable option for building modern web applications? a performance evaluation study," *Computing*, vol. 97, no. 10, pp. 1023–1044, 2015.

[11] Node package manager. [Online]. Available: https://npmjs.com

[12] A. J. Poulter, S. J. Johnston, and S. J. Cox, "Using the mean stack to implement a restful service for an internet of things application," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 2015, pp. 280–285.

[13] N. Jain, A. Bhansali, and D. Mehta, "Angularjs: A modern mvc framework in javascript," *International Journal of Global Research in Computer Science (UGC Approved Journal)*, vol. 5, no. 12, pp. 17–23, 2015.

[14] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks*, p. 33, 2008.

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[16] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A quantitative analysis of developer information needs in software ecosystems," in *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, 2014, pp. 1–6. [Online]. Available: http://scg.unibe.ch/archive/papers/Haen14a-QuantitativeEcosystemInformationNeeds.pdf