ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Μικροϋπηρεσίες εξαγωνικής αρχιτεκτονικής με gRPC για διαδικτυακό σύστημα διαχείρισης προσωπικού»

Του φοιτητή
Άγγελου Κουρατζίνου
Αρ. Μητρώου: 174949

Επιβλέπων
Αντώνης Σιδηρόπουλος
Αναπληρωτής Καθηγητής

Θεσσαλονίκη, 1/2/2025

Τίτλος Δ.Ε. Μικροϋπηρεσίες εξαγωνικής αρχιτεκτονικής με gRPC για διαδικτυακό σύστημα διαχείρισης προσωπικού
Κωδικός Δ.Ε. 22270
Ονοματεπώνυμο φοιτητή Άγγελος Κουρατζίνος
Ονοματεπώνυμο εισηγητή Αντώνης Σιδηρόπουλος
Ημερομηνία ανάληψης Δ.Ε. 19/10/2022
Ημερομηνία περάτωσης Δ.Ε. 25/01/2025

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Άγγελου Κουρατζίνου που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

«Στο μέλλον της τεχνολογίας»

Prologue

Before enrolling in higher education, I was mostly interested in topics related to system administration and service operation. During my time in university, I learned more about programming and software development and became increasingly interested in that area. Motivated primarily by realizing that software I had thought of as difficult to get running was just software like any other and could be improved to more closely align with new deployment and operations practices, I wanted to find out more. To that extent I read up more on how large systems functioned and how companies with millions of users approached both the operational and development aspects of the technology they employed. With that in mind I chose this topic for my thesis due to the freedom allowed in the implementation and the opportunity to explore the concepts I had read about and applied in smaller demo-like projects in an application with a real-world use case. After the conclusion of the software development part, I believe it has benefitted me by changing my perspective on how the concept can and should be applied as well as the complexity involved in applying them to smaller scale systems.

Summary

The topic of the paper is building an application using modern technologies and development practices. The application's purpose is to help the user, in this case a university administrator, schedule the upcoming semester by assigning courses to professors. The application should be modular, have the ability to scale and follow today's best practices to the extent possible. The exploration of these technologies and practices is also a goal of this paper. In the end, the goals were achieved and there were a lot of useful takeaways from following this implementation strategy. If the project was to be re-implemented from zero, a few things would have been done differently thanks to the knowledge and experience gained. However, the process was a positive development experience that resulted in an application that fulfills the requirements.

«Hexagonal architecture microservice with grpc for a web-based personnel management system»

«Angelos Kouratzinos»

Abstract

This paper explores the application of modern software development practices to build a full stack web application that helps a university administrator manage the assignment of courses to teaching staff. The technology stack and the internal application structure and architecture are of interest while making the application easy to deploy and manage outside of a cloud environment. During the last 20 years the way applications are written, tested, deployed and managed have changed drastically and practices previously reserved for large scale environments are entering the mainstream. A secondary aim of this paper is to explain the benefits of the practices used and provide the reader with an understanding of where they should and should not be applied.

Acknowledgements

Σε αυτήν την ενότητα ο φοιτητής/ φοιτήτρια προαιρετικά μπορεί να ευχαριστήσει όσους αισθάνεται ότι συνέβαλαν (επιστημονικά, ηθικά, οικονομικά κτλ) στην ολοκλήρωση της διπλωματικής εργασίας.

Table of contents

Prologue ... iv
Summary ... v
Abstract ... vi
Acknowledgements ... vii
Table of contents ... viii
Figures ... x
Listings ... x
Abbreviations ... xi
Chapter 1: Problem space ... 12
1.1 Introduction ... 12
1.2 Functional requirements ... 12
1.3 Non-functional requirements ... 13
1.4 Design process ... 13
1.5 Epilogue ... 15
Chapter 2: Technologies used ... 16
2.1 Go ... 16
2.2 Protocol buffers ... 19
2.3 gRPC ... 19
2.4 gRPC-gateway ... 20
2.5 Buf ... 21
2.6 Docker ... 21
2.7 GitHub ... 22
2.8 MySQL ... 22
2.9 OAuth2 ... 22
2.10 Prometheus metrics ... 24
2.11 Template project ... 25
2.12 Epilogue ... 26
Chapter 3: Application architecture ... 27
3.1 Introduction ... 27
3.1.1 Domain-driven design ... 27
3.1.2 Hexagonal architecture ... 27
3.1.3 Microservices architecture ... 27
3.1.4 Backends for frontends ... 28
3.2 gRPC communication ... 29

3.2.1 Protocol buffers definition ... 29
3.2.2 The grpc package ... 30
3.2.3 The gateway package ... 31
3.2.4 gRPC API description ... 31
3.2.5 REST API description ... 36
3.3 Data storage ... 40
3.3.1 SemesterSchedulingRepo interface ... 40
3.3.2 MySQL implementation ... 40
3.3.3 Database schema ... 41
3.4 Domain logic ... 44
3.5 Frontend ... 44
3.5.1 Per-page analysis ... 45
3.6 Utils package ... 53
3.7 Configuration ... 55
3.8 Documentation ... 55
3.9 Request-response examples ... 56
3.10 Continuous Integration ... 57
3.11 Epilogue ... 58
Chapter 4: Conclusions and future work ... 60
BIBLIOGRAPHY ... 63

Figures

Figure 1: Oauth Architecture. 23
Figure 2: Database schema main tables 43
Figure 3: Database schema secondary tables 43
Figure 4: Logged out homepage 46
Figure 5: Logged in homepage 46
Figure 6: Profile page 46
Figure 7: Semester overview page 47
Figure 8: Individual semester page 47
Figure 9: Individual semester page text overflow 47
Figure 10: Semester creation page 48
Figure 11: Semester creation copy assignments dropdown 49
Figure 12: Professor overview page 50
Figure 13: Individual professor page 50
Figure 14: Professor creation page 51
Figure 15: Assignments overview page 52
Figure 16: Assignment edit page 53
Figure 17: About page 53

Listings

Listing 1: UniversityAuthenticationService protobuf definition 32
Listing 2: Semester RPCs 34
Listing 3: Professor RPCs 35
Listing 4: Irregular RPCs 35
Listing 5: Professor JSON response 37
Listing 6: Course JSON response 37
Listing 7: Semester JSON response 38
Listing 8: Assignments JSON response 39

Abbreviations

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία
DDD	Domain-Driven Design
protobuf	protocol buffers
RPC	Remote Procedure Call
gRPC	gRPC Remote Procedure Calls
BFFE	Backends For FrontEnds
CLI	Command Line Interface
CNCF	Cloud-Native Computing Foundation

Chapter 1: Problem space

1.1 Introduction

The main goal of this project was to write an application that can aid with the management of course assignment to university personnel for each semester. At the moment this job is handled by entering data into a database by hand, so the aim of the project is to improve the process. Along with fulfilling the functional requirements, the application developed for this paper was to be implemented using a modern and extensible internal application architecture, explore its usability and provide a concrete real-world example of applying hexagonal and microservice software design patterns. Additionally, to provide more value, the choices made should be evaluated with regards to their applicability in similar use cases. The target audience for the application is primarily university administration staff although provisions have been made to allow it to be extended and provide a subset of the functionality to non-administrators without the requirement to majorly refactoring the code. The intended user base impacted several design decisions both in the realm of security requirements as well as the design of the user interface and the API that the application provides. When starting the project, it is important to note down the requirements, functional and not, that it needs to be fulfilled.

1.2 Functional requirements

The functional requirements were described mostly at the start of the project and during development a couple more were discussed, only a subset of which ended up being implemented. The full list is the following:

- The application allows interaction through a web interface accessed through a web browser
- The application allows interaction through an HTTP API
- The application only allows university administrators to view and modify the data
- The application allows viewing assignments grouped by the professor
- The application allows viewing assignments grouped by the course
- The application allows copying the assignments made in the same semester of the previous year

First, starting from a user's point of view, is the interface. This translates to a web-based frontend for general users as well as an HTTP API for programmatic access which is mainly aimed for use by developers. These are common requirements for most web-based applications nowadays and they provide a solid starting point. Next on the list is the requirement for strict security. The application data about assignments should be available only internally and should not be seen by anyone except authorized administrators. Following that, are some more specific requirements regarding how the assignments are presented. They should be shown grouped by the professor and also grouped by the course. This allows more easily spotting professors who have too little or too much work assigned to them, as well as courses that are not assigned to someone while they should be. Last but not least, when creating the schedule for a new semester, last year's schedule for the semester at the same time of year should be able to be used as a template. This means copying over the assignments and allowing the administrator to make alterations instead of having to build everything from scratch. These were the main functional requirements that the application had, amounting to providing a simpler way to create the teaching schedule for a semester.

1.3 Non-functional requirements

In line with the goal of writing the application in a way that explores certain technologies and design patterns, there were quite a few non-functional requirements that guided and informed implementation details. These were the following:

- The application allows storing data in a MySQL database provided by the university
- The application allows user authentication using the university login API
- The application allows interaction using a gRPC API
- The application web interface does not use a JavaScript framework
- The application internal architecture follows hexagonal design principles to the extent possible
- The application architecture follows microservice design principles to the extent possible
- The application code is as flexible as possible without making sacrifices to the code quality to achieve this goal

Part of the manual workflow currently in place is an existing database where the schedule is stored. While exploring an extensible architecture is part of the development process of this application, it is also important to integrate with the existing systems that are already in place. For this reason, two non-functional requirements were set. The first one being that MySQL should be used as the database and second, the university login API should be used for authorization. If they are not the only choice, they should at least be in the pool of available options. These are external constraints and.

Other non-functional requirements were set due to my preference or curiosity. The first one I set, due to personal interest, was the requirement for the application to offer a gRPC API. Both gRPC and protocol buffers appear to be interesting technologies, and I wanted to explore them in the context of building an application with real-world constraints. In the same realm, I wanted to implement the frontend without using one of the big single-page application frameworks and instead take a more minimal approach to incorporating JavaScript. Any client-side scripting in the frontend should be included only when and where it is required.

Next in the realm of non-functional requirements, there are a few architectural patterns that are of interest to me and I wanted to explore as part of this paper. These are the hexagonal design patterns for internal application architecture as well as the microservices architecture which focus on splitting the application into multiple smaller ones. In these requirements I added the caveat of specifying to the extent possible due to inexperience with actually applying these patterns. It is an exploratory approach where I make my best effort to satisfy the non-optional functional requirements while also looking into the non-functional ones for what amounts to personal interest and curiosity. Last, and logically in the same category as the above, I added the goal of making the codebase generally flexible and pluggable without making the code worse because of it.

1.4 Design process

Before writing any code, a general understanding of the intended workflow as well as gathering the requirements was necessary. Starting with the first meeting and a discussion about the project, more details were discussed and a paper about a previous version of the application was provided. Following that, the existing database was dumped so it could be restored and there was a known working state, and it could be examined. It is generally good practice when creating an application based on existing data to take that into account and structureure the backend API in accordance with the database schema. It ended up easing the development process a lot since there was a working model of the domain and there

were very few issues with it and most of the ones that did exist were related to column naming which was easily fixable.

Domain understanding is important and so is knowing what the existing data means. A few examples are the difference between what is called teaching hours and assigned hours in assignments as well as default full hours with regards to professors. The former is the difference between hours spent in lectures, teaching hours, whereas the assigned hours are the total which can include lab time and is distinct. For professors, the default full hours amount determines how many hours of teaching a professor needs to be assigned if their contract is going to be properly fulfilled. Going under that amount or over it is an issue, and the application should help with making that known to the user, creating the semester schedule and assigning courses. Some other finer details were the kind and type columns in the courses table which were “enums” (enumeration fields). A few of the values were not obvious so after noting down what those meant, it was part of the plan to more clearly label them in the code. A similar situation was encountered with regards to the semester type and the exam periods but none of it ended up being any sort of significant hinderance to the overall design. Quite the opposite, they provided a good opportunity to demonstrate how the technologies chosen would model the same thing in a way that was idiomatic to them. With those first details in hand and a bit of disambiguation of what some of the values meant in some of the columns, the design process for the backend could begin.

The first step of the process was to clone a template project that was mostly working. This will be expanded upon in a later section, but as far as it relates to the scope of the project the important detail is that it is a proof of concept that uses the same set of technologies as well architectural concepts that I wanted to explore. Already having the database schema as a basis, the next step was to model the entities from the perspective of the application code. Owing to the template structureure, the process of adapting it for different entities was a matter of simple find and replace operations. Starting by creating protocol buffers messages following the database tables' form and then moving to Go structureures which more closely mimic the database tables. With the intent to reaffirm that the template was functional and would fulfill its purpose, a single piece of functionality was built end to end. This included a database function, a logic function, a gRPC method, a corresponding REST API mapping and finally a web page in the frontend. Getting a list of professors was a success and proved that the approach was going to be usable for the project.

Next up was getting to grips with ensuring the application could also meet the security and confidentiality requirements. The previous version of this application as well as other applications developed for a thesis use the university’s OAuth2 login API once to ensure that the user is authorized and then force them to login again after a specific amount of time. While in some situations this could be enough, making full use of the API and not reducing the security guarantees it provides means that every request for privileged information should be properly authorized. A user should lose access when their token becomes invalid and cannot be refreshed. For this reason, a lot of time and effort was spent on ensuring this worked properly. The login feature has functionality that started being written as a command-line tool even before the project properly started but was finished a long while into the project after the final known bug was fixed. Integrating with the external authentication API was an important part of the project and took a long time to come into fruition in the final form it exists in but security was an important aspect of the application, so it made sense to spend the time to get it right and properly integrate with the university’s OAuth2 API.

1.5 Epilogue

Concluding this chapter, the functional and non-functional requirements of the application have been noted down along with the design process that took place prior to beginning the coding part of the project. At first, it was an analysis of the domain entities and the types that represent them in the database and after that the equivalent types in the application code. Following the data types, a general idea of the application needed to be sketched out. This started with a template project that was reconfigured and then continued with a proof-of-concept implementation of one small slice of the application to broadly gauge the viability of the approach which was then used as a jumping off point to implement the rest of the project. These steps did not include a lot of code, instead leaning more towards getting an understanding of the context the application should exist in. This approach ended up being quite beneficial in the long term by providing a solid starting point and preventing foundational errors from being made.

Chapter 2: Technologies used

2.1 Go

The Go programming language [1] (sometimes also referenced as Golang) is the language used for most of the project. It is a statically typed, compiled, high-level language that is open-source and has seen a lot of adoption for web-related services. One of the most noteworthy features of it is the support for concurrent programming. The runtime offers programmers the ability to work using go-routines, lightweight user-space execution threads, and the standard library provides many useful packages to schedule work among go-routines and communicate between them. In addition to the features of the language itself, there is a great ecosystem of libraries and support for various software systems such as databases and communication protocols. The development team adds syntax features cautiously so there is little need to relearn ways of writing idiomatic Go code. The official tutorials encourage learning in an interactive way with online code compilation and execution to assist in understanding the concepts at hand. All the above factors in combination make the language have a smooth learning curve. The documentation for a package is also easy to write and view online for public code which helps reduce the friction in writing and reading useful documentation.

Go has a long history, starting life around 2007 inside Google before being announced to the public in 2009 and officially released in 2012. Its design goals were to achieve C-like performance with Python-like readability with a focus on networked services running on multi-core systems. The lead designers were Robert Griesemer, Robert Pike and Kenneth Lane Thompson. Robert Pike, to a greater degree than the rest, is more known for being vocal at present but everyone's contributions shaped the language into what it is today.

Parallel to that, since its adoption has increased over the years, there are multiple design patterns that have been implemented using the language and idioms have been developed that fit its unique characteristics. Most noteworthy and perhaps useful for the application in question is the use of interfaces to abstract away implementation details and provide a path for interchangeable implementations. Many concepts described in abstract as part of architectural documentation were implemented in code using Go interfaces.

The way Go handles interfaces can be referred to as "duck-typing". The common example is that an interface called Duck defines that any structure with a Quack method defined on it, which takes no arguments and returns no values (meant to print the string "Quack" to a terminal) satisfies the constraints of the interface. This does not mean that the structure satisfying the Duck interface cannot also satisfy any others at the same time. There could be another interface called Pet that says any structure with an IsPet method returning a boolean value and a Name method returning a string value, satisfies that interface. One structure could satisfy both interfaces and be used as an argument to functions requiring a Duck type as well as functions requiring a Pet type. The end result is expressing useful properties of types by defining methods on them and designing interfaces that specify their requirements based on those methods.

A very useful language construct after that was the ability to embed files in the resulting binary which aided in the simplicity of deployment. The interfaces supported by the standard library, specifically the ones in the io package, are crucial to providing flexibility for library and application developers. In this case, the embedded filesystem could be used instead of a directory or file on the disk. Any files that would need to be copied over manually can instead be included inside the compiled binary and reduce

deployment overhead. Go enabled this using a directive, essentially a comment in a specific format that instructs the compiler to perform some operation. In this case it is //go:embed and it initialized the variable above which it is placed with the value of the file or directory specified. This variable must be a string, an array of bytes or io.FS (a type representing a filesystem provided by the io package of the standard library). The compiler then reads it and performs the necessary actions to include these files in the generated binary. The use cases for this feature are plenty and utilizing it can greatly reduce the overhead of managing the deployment of a web application that includes HTML templates for rendering the frontend such as the application developed here.

In terms of the binary, another key part of Go is the ability to generate fully statically linked binary executables. By default, some parts of the standard library will link to the C library of the host system. As an example, the application developed in this paper will link to several system libraries when compiling on Linux using the go build command. In a lot of circumstances this does not have any negative side-effects such as the case where the application is compiled and executed on the same system. In other instances, however, it can be a great hinderance, for example when the development workstation does not have the operating system or the same version as the deployment target. Another case is for building Docker images that utilize a multi-stage build where the first stage is based on a Debian Linux image while the final image where the compiled binary is copied to is based on Alpine Linux. In those cases, the binary will fail to execute correctly if at all. To counteract that and to enable other use-cases too, Go provides developers the ability to compile a binary that does not depend on system libraries and instead uses equivalents written only in Go. The os/user and net libraries that are part of the standard library are the two main examples. This is easily achieved by setting just one environment variable, CGO_ENABLED to zero. When performing the test from earlier, compiling the application developed for this paper, the ldd tool on Linux reports that it is not a dynamic executable. Alternatively, this can be done by passing the argument -tags netgo,osusergo to the go build command. Fully static compilation is a very beneficial language and compiler feature that Go offers when it comes to creating deployable artifacts that can run independent of system libraries and enables a greater experience for service operators as well as developers.

In addition to static compilation, Go also provides easily accessible compilation across operating systems and architectures. In a similar fashion to the example detailed above, all that is required is to set the GOOS and GOARCH environment variables to the desired values and run the go build command. By default, these variables will be set to values appropriate for the host executing the compilation. If, however, on an x86_64 Linux system a developer wants to compile a binary suitable for a server that uses the arm64 architecture, all that's required is to set GOOS=linux GOARCH=arm64 and then run the go build command. Checking using the file command on Linux will inform the developer that the ELF executable is compiled for ARM aarch64 and is therefore suitable for the target server. Repeating the same process but changing GOOS to windows will produce a PE32+ executable while setting it to darwin will produce a Mach-O executable suitable for a MacOS system. This covers the main operating systems in use today and is a very important feature for cross-platform applications. In specific, developer tooling and more specifically command-line applications benefit the users greatly by allowing them to use the operating system of their choice without losing access to the tools they prefer and are used to. The cross-compilation features that Go provides are easily accessible and provide developers, service operators and end users the ability to run a Go binary on many systems even if the system that the binary was compiled on does not share the same operating system or CPU architecture.

The web development aspect of Go is one of the best aspects of using the language. Both the standard library as well as the wider ecosystem assist developers with writing fast and efficient web applications

easily. The HTTP server included in the Go standard library is fast, was one of the first to receive HTTP/2 support which all Go applications got just by updating and includes support for TLS 1.3 to secure connections. No matter what storage system the application utilizes, be it a traditional RDBMS like MySQL, a newer NoSQL database like MongoDB, a distributed database system such as Cassandra, a data cache like Redis and memcached, a search-oriented datastore like ElasticSearch or an online analytical processing database like Clickhouse, there are Go libraries for interfacing with them. As far as data formats are concerned, there is first-party support in the language itself. Structure fields can be annotated with structure tags to denote what the structure field should be called when data of that type is mapped to a specific data format. This works for YAML, TOML, JSON, BSON, XML and others but it can also be a row in a database table. The most popular ORM in Go called gorm uses this feature to create mappings between structure fields and database tables and so do a few other libraries. The mapping of Go structures to database tables and columns and vice versa is a very interesting feature and a major point of interest but more about it will be explained in a later section when discussing the internal application architecture.

As far as more generally used Go features, the testing functionality included by using the go test subcommand is another highlight of the language. Adding unit tests for the code in a file, for example code.go, is as simple as adding another file with the suffix _test, following the previous example this would be code_test.go. This test file should have functions named after the ones that need to be tested but prefixed by Test. Afterwards, the tests for a package can be run by executing go test path/to/package or for all code in a module by executing go test ./... which is a shorthand for looking through all subdirectories.

Analyzing tests results and gaining insight into which parts of the code are covered, and which are not, is an important feature of testing tools. The test sub-command of the go command-line tool can also generate code coverage profiles with the go test –cover profile command and then read them as HTML in a browser. These show exactly which statements in the codebase are covered by tests and which are not. This differs in approach to other languages where the testing framework is not as well integrated and might not even be included as part of the tools that the language developers provide. Go does not require developers to adopt a unit testing methodology though. In cases where an integration test is required or preferred, a binary can be built in such a way that the results will be recorded, and a coverage profile can be generated even in the case where unit tests are not used. Overall, the testing capabilities are great and provide an easy and well-integrated way of writing and executing tests using the default tools and language features.

Together with code correctness, developers are interested in performance monitoring and tuning their applications. On this front, Go has a handful of interesting features when it comes to performance profiling and analysis. First, the go test command has the -bench flag which runs any functions prefixed with Benchmark along with any prefixed with Test as was previously mentioned. That constitutes the baseline for benchmarking but can be augmented with several other flags such as benchmem to print memory allocation statistics which is a common starting point for optimization. Benchmarks can be configured to run a certain number of times or as many times as required to fill up a specific time period and with a specific usage limit on CPU cores alongside other options. Complementary to coverage profiles for tests, developers can also generate CPU, memory, mutex and blocking profiles. These facilitate diving deeper into CPU bottlenecks, non-optimal memory access patterns, mutex lock contention and goroutine utilization bottlenecks. To make using these statistics easier, the benchstat tool is officially provided to analyze and compare benchmark runs when conducting benchmarks to figure out which version of the code has better execution characteristics. In early 2023, this tool was also

rewritten and got a lot of new features and improvements. Another useful profiling feature specifically for web applications pprof statistics. This provides CPU profiling data as well as memory allocation statistics in the browser under the /debug endpoint. By importing the net/http/pprof package and adding a couple lines of code to the HTTP router, developers can view these performance profiles in the browser and observe them over time or while running a load test to gain insight into which code paths could potentially benefit from performance optimizations. Overall Go provides plenty of ways out of the box to analyze and improve software performance and enable developers to better understand their code's performance.

2.2 Protocol buffers

Protocol buffers, often shortened to protobuf, is a data format with the primary characteristic of efficient binary serialization and de-serialization. It avoids common issues with schema-less formats by requiring the message definitions to be accessible beforehand. By knowing what fields are supposed to be in which part of the message, de-serializing data in protocol buffers format from the binary data it is sent as when encoded can be done very efficiently. Additionally, the definition language of protocol buffers, both the messages and the RPC services, does not depend on the language for which the code will be generated. Even though it is not a requirement, some options specific to a language can be specified such as the package name in Java or Go so there is flexibility to adjust options based on the language.

The most important part of the protocol buffers tools is the compiler. The most common one is protoc and it can be used to generate code in a number of languages from a protocol buffers definition file. For this project, a newer tool called Buf was used since it improved the development experience. Target languages for code generation are supported through plugins which allow each programming language community to write a plugin that generates idiomatic and performant code for that specific language. Furthermore, it allows lesser known or used languages to add support for themselves without requiring upstream approval. Overall, utilizing protocol buffers with gRPC makes sense but each one is flexible enough to be used standalone and those use-cases are supported.

In the protocol buffers ecosystem, one example of a narrow use case being addressed by third parties is the nanopb project. This enables generating smaller code that can be stored and executed in a microcontroller. This fulfills a specific need that developers in the embedded space have but is otherwise not very applicable. Another example is the Julia programming language. It is not officially supported, but its community could write unofficial code generation support and use protocol buffers in Julia applications and services. The modular design of the compiler and the code generation offers a real-world tangible benefit as can be seen in the examples mentioned here.

The protocol buffers concepts have close equivalents in gRPC. For example, when used in conjunction, one gRPC server is created for each protocol buffers service and the protocol buffers messages are used as the inputs and outputs for each RPC method. With that, both protobuf language-specific types as well as gRPC communication code is generated. For Go in specific, they are even split in two different files by default since each one is generated using a different compiler plugin. This is a very specific and minor example that demonstrates how the two are related but not tightly coupled.

2.3 gRPC

Interaction with the backend primarily happens through remote procedure calls using the gRPC framework[2] as the communication method and gRPC-gateway which will be discussed in the following section. It is a technology used internally by Google and is built on top of HTTP/2 with

support for several communication modes. These modes range from fully unary RPCs resembling a typical function call or an HTTP/1.1 request and response cycle to entirely bi-directional meaning both client-side and server-side streaming. In essence, the client could be sending data at the same time as the server is responding, which creates new ways to interact between services. The HTTP/2 standard was instrumental in enabling technologies such as gRPC to exist due to the changes in the long-lived connection handling.

Other than what gRPC uses in terms of underlying implementation, other parts of its design make it an appealing option for writing APIs. It was built to support code generation in a lot of popular languages since Google uses mainly C++, Java and Python as well as Go. This feature allows a gRPC server and client to work together across language ecosystems. In large organizations that could have teams that only know one language and want to integrate with software that a team using a different language has, this is invaluable. In addition to multiple language support, gRPC supports multiple data formats although in most cases protocol buffers are chosen and to a lesser extent JSON. In theory nothing prevents it from working with XML but that is an uncommon enough use case that the ecosystem support for it is virtually non-existent. Since the protocol buffers data format was one of the technologies that this paper was meant to explore, using gRPC together made sense for this application. The code generation provisions for both protocol buffers and gRPC allow the creation of various tools for different purposes.

2.4 gRPC-gateway

Requiring all programmatic access to the application be through a gRPC client which requires some investment to learn, use and adjust to, would not fulfill the requirement of cleanly integrating with the rest of the university software. To address this issue, a plain HTTP API is offered alongside the gRPC one. One way to offer an HTTP API would be to manually write code that takes HTTP requests, maps the URL path parameters and body data to protobuf messages and then makes the appropriate RPC call to the gRPC server. That would be very time-consuming and prone to errors as manual work usually is. So, to achieve the desired end result, a protoc compiler plugin and library from the gRPC-gateway project was used. To integrate with the ecosystem gRPC-gateway is implemented as a protobuf compiler plugin that looks for google.api.http annotations in the RPCs defined in a protobuf definition file. By applying these annotations, programmers can create mappings from RPCs to a JSON-based HTTP API (usually a REST one but this due to convention, not a requirement or restriction that gRPC-gateway imposes) and provide both of them to API users without doing double the work.

In addition to the primary function of being a gRPC proxy, gRPC-gateway enables generation of health checking endpoints, which are used by load-balancers, Docker and Kubernetes, Additionally, it makes integrating with gRPC authentication easier since a developer does not need to write a lot of custom code to enable this functionality. At this time, with the minor addition of annotations to the protocol buffers definition file together with a protocol buffers compiler plugin is all that is required to automatically generate OpenAPI specification (formerly known as Swagger) for the API that gRPC-gateway generates. The application developed for this paper leverages the advantages of gRPC-gateway to offer a REST API along with an API specification suitable for generating user-facing documentation for it.

The HTTP API specification produced is a JSON file adhering to the OpenAPI spec which is not the most user-friendly way of presenting documentation. To fulfill this role, two approaches are shown, Swagger UI and Redoc. Swagger UI is distributed in several forms but in this case swagger-ui-dist was

chosen due to the application architecture requiring serving the assets from the server. It takes the location of a specification file and provides a single page with all the endpoints to allow someone to try it and inspect the response. Redoc takes a different approach and instead generates a single file, making distribution a bit easier. The goal is the same and their approaches are similar, but Redoc appears to be a bit more modern and well-maintained while development progress for Swagger UI seems slower. Overall, both are good choices with different tradeoffs, they are provided as options in the application and will be explored in more depth in the section discussing the application documentation.

Initially, the option of gRPC-web was considered but it has some obvious downsides. The main one was that due to being browser-based it has limitations that have been known for years such as lacking support for many of the communication modes that gRPC offers. Additionally, it would require the adaptation to be moved client-side meaning it would force client developers to do the integration work. For example, a mobile application developer would need to integrate using a different method, possibly by making gRPC requests directly, since the application runtime on a mobile device is not the same as a JavaScript engine in a browser and any gRPC-web code could not be reused. Overall, gRPC-gateway fulfilled the requirements and had several advantages over gRPC-web, so it was the solution chosen.

2.5 Buf

There are a few protocol buffers dependencies are not handled by the compiler, so an extra tool was needed. Buf is a project made up of a CLI tool and a plugin registry that aids in developing protobuf-based applications and simplifies dependency management. From the things mentioned so far, there are four compiler plugins needed to generate code for the application:

- A plugin for generating encoding and decoding code
- A plugin for generating gRPC communication code
- A plugin for generating gRPC-gateway code
- A plugin for generating documentation for the gRPC-gateway API

Managing all plugins mentioned above manually is a difficult enough process and that list does not even include the imports of annotations and protobuf types used as dependencies which are very commonly included and almost necessary to have in the majority of normal size projects. Even for a small project it becomes unwieldly to manage them by hand and take care to update them one by one. This is the problem that buf solves. The buf schema registry hosts the compiler plugins and a lot of the common protobuf dependencies. This allows developers to specify what dependencies they need in a similar way to the Go module system. Compiler plugins go in one file, buf.gen.yaml while dependencies go in another, buf.yaml, and the hashes of the dependencies are stored in the buf.lock lock file. The added simplicity and better workflow compared to protoc is why buf was chosen for this project.

2.6 Docker

With regards to packaging and shipping web-based applications, Docker has emerged as the category leader. After its launch in 2013 it garnered attention for simplifying application deployment and providing a unified type of deployable artifact, the Docker image. In later years, several parts of what Docker was originally, were turned into standards through the work of the Open Container Initiative. Adhering to the goal of using modern development and operations practices, the application has been tested and developed with Docker in mind. A Dockerfile is included in the repository and the Docker image produced by it is the main way that the application is meant to be deployed closely followed by running the binary directly if that is what the hosting environment is suited for.

The Dockerfile performs a clean build of the application including downloading the dependencies. It is not the only way to write a Dockerfile and depending on the use case, it could even be undesirable to perform the build this way. There are alternative approaches for handing software packaging inside Docker, such as providing an already built binary and then only using Docker as a last packaging step, which will be explored later. Support for the container ecosystem could be enhanced by adding a Helm chart. However, there were no plans to deploy this application to a Kubernetes cluster, so it was not included in the final version.

2.7 GitHub

GitHub was used to host the source code, manage code versioning, handle parallel independent development of features and continuous integration. For similar reasons that Docker support was added, GitHub was chosen as the code hosting platform. According to the StackOverflow Developer Survey from 2022, it is the most popular version control platform for both professional and personal use. Even in the free tier offering, it provides git repository hosting along with other helpful features such as pull requests to facilitate the process of change management and GitHub Actions for Continuous Integration with enough computation time for a small project. Both of those features were used, and the automatic testing of each commit helped when it came to debugging in which commit a problem had been introduced. Knowledge of both Git and GitHub was acquired during studying in university and are widely used tools in software development, so it made sense to incorporate them.

2.8 MySQL

As one of the functional requirements was integrating with existing infrastructure, MySQL was used as the application's database. MySQL is another open-source component used in this project and similar others, has a very big community. It is one of the most popular and widely used databases and is what our university uses in most cases. Rather than the Oracle version, the MariaDB fork is the version chosen. Its performance characteristics are excellent, and it has been very reliable throughout the development of this application. A lot of the companies using MySQL serve millions if not billions of users worldwide and this is one of the main reasons why database administrators and developers of old and new software choose it. Supporting the MySQL database where the existing data is stored was a clear requirement for the project and provided a great starting point for modelling the data structures used throughout the application code. The schema was already well-made and did not need to be created from scratch. The database was one of the areas that was least explored when it came to more modern alternatives. In the end, MySQL remains a solid choice up to this day and using it did not have any negative side-effects in the development experience.

2.9 OAuth2

Since its release in 2012, OAuth2 has become more and more common for authorization in web applications. The university offers an OAuth2-compatible API along with usage documentation for different scenarios. One of the application's functional requirements was that only users marked as administrators can perform most operations. The user performing an action should have the necessary access and permission to do so which is why the token is checked on all restricted calls. For this application specifically, the role of the user is determined by retrieving the information from their profile data, but this is an implementation detail specific to the university login API and not a requirement by the OAuth2 specification.

It is important to have a general understanding of how OAuth2 works and how it applies to the application. OAuth2 describes four roles, the Client, the Resource Owner, the Resource Server and the Authorization Server. The Client role is filled by the application which is asking on behalf of a user to gain access to a restricted or protected resource, in this case the user profile. For this purpose, the user in their role as Resource Owner, is redirected to the web page of the login API and is asked to authorize this application to access some of their data. If approved, the application provides its client identifier and secret to prove its own identity along with the authorization grant that the user allowed it to get. Assuming no errors were encountered, and the application identity and authorization grant are valid, the Authorization Server provides a valid access token to the application that it can use for requests accessing restricted resources. As a last step, the provided access token is used to get some data, like the user profile in this case, and if the provided access token is valid the Resource Server will return the requested data. The above process is using the Authorization Code grant type and is the most common.

The above assumes that some setup has taken place, the Authorization Server needs to have a way to identify the application which is not part of the described process. The application needs to be registered before it can start being authorized to access restricted resources. Since this is aimed towards application developers, it is found is the relevant page of the website of the service. In there, some details about the application need to be provided such as the name, description along with one or more redirect URL (sometimes also called callback URL). Following that registration, the developer will receive a client identifier and a client secret, these are what make up the identification that their application will need to provide together with the authorization grant before being provided an access token as described previously. In the case of the university login API, the callback URL is configurable by the application, but it must be one of the URLs provided during application registration.

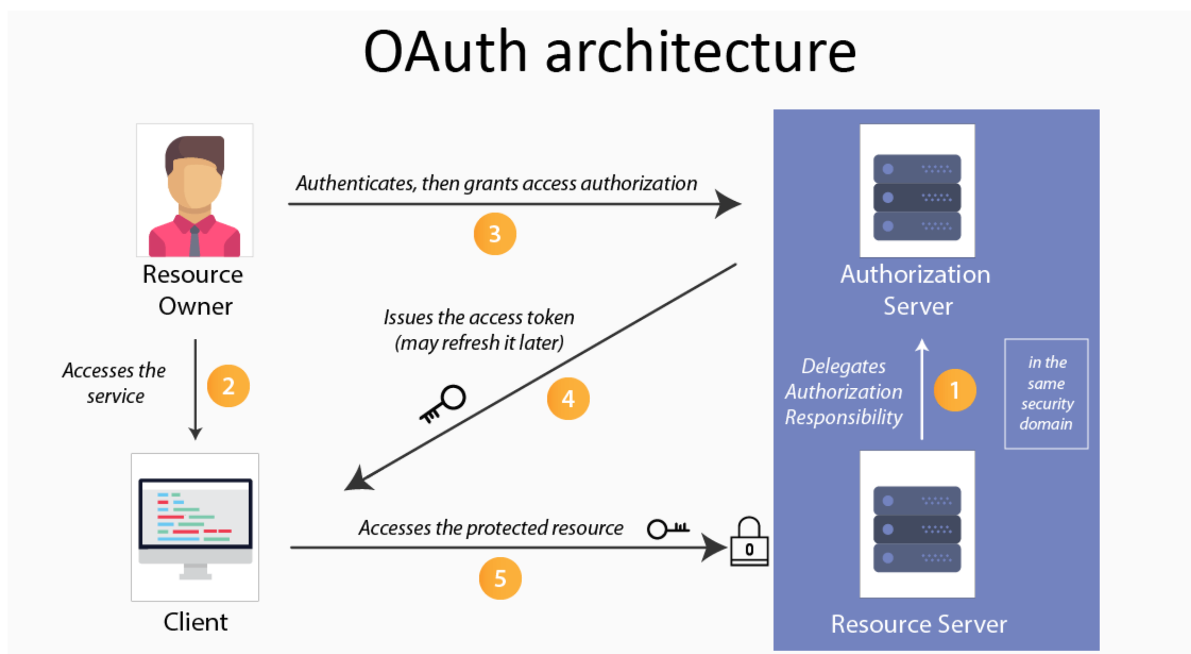

Figure 1: Oauth Architecture.

In diagram form, the process described above looks like the picture in Figure 1. The part about the refresh token is especially worth highlighting here. In our application, the refresh token is used to get a new access token without the user having to perform the login authorization flow again. There is also a retry mechanism for requests in the frontend. This works by checking the error type when an error is

returned and if it is a token invalid error, the client tries refreshing the access token and making the request again. Using this approach, the application always abides by the authorization level of the university's API by respecting the access token as well as the refresh token validity timeouts. The downside is that if a user is idle for too long, they have to log in again.

2.10 Prometheus metrics

In the last decade there has been a growing emphasis on monitoring applications and nowhere is it more evident than microservice ones. The growth in the quantity of services also requires adapting and improving the surrounding infrastructure such as the monitoring system. As part of this process is where the Prometheus monitoring system was created [3]. SoundCloud was moving to a microservices architecture and needed a better solution for monitoring. In a move that turned out to be critical to its adoption, Prometheus was open source from the beginning of 2012 which helped with initial adoption. Further down the line, it became one of the first few projects to reach Graduated status as part of the Cloud-Native Computing Foundation. Due to multiple factors, it has ended up as one of the most widely used monitoring systems with an incredibly rich community and wide ecosystem support. A lot of other widely used infrastructure projects have support for exposing metrics in the format Prometheus uses and understands so it made sense to add this option in the application.

Exposing metrics for software like databases and message queues is necessary for them to be monitored. For this purpose, there are programs called exporters that run alongside and connect to a database for example. The exporter can then serve an HTTP server so Prometheus can scrape metrics from it and that way monitor the database. There are exporters for PostgreSQL, MariaDB/MySQL, Redis, MongoDB and many others. Altering the monitoring setup of other infrastructure components was outside the scope of this paper but it's worth mentioning as a show of ecosystem support.

Looking at Go in specific, Prometheus is written in it. This means that there is idiomatic and first-class support for instrumenting Go applications and automatically exposing metrics from them. One of the most helpful things is the ability to expose runtime information with only a handful of non-intrusive changes to a Go program. In the case of an existing service that exposes HTTP endpoints, the change is just two lines. This then allows a Prometheus instance to start scraping that endpoint for metrics and not much else is required from the developer. For a more feature-rich experience or even some more specific use case, custom metrics can be created. Here too, the Go libraries are very helpful, especially promauto which has some convenience functions for creating new custom metrics and makes the general Go library a bit easier to use. Of course, it's not required, and other languages and frameworks are not left behind. One great example is the Spring framework in Java which is widely used for backend applications, through a simple checkbox selection in the Spring Initializr webpage, allows enabling support for a Java Spring application to expose Prometheus metrics. Other languages have their own equivalents and libraries for creating such metrics are widespread and well supported. Since in this application only the built-in metrics are exposed it's just one option to allow monitoring the basics for performance and uptime.

Though the monitoring system itself is not a part of the project, adding support for it was one important addition. Monitoring an application is a big part of keeping it in service and maintaining it. To fulfill this role, Prometheus metrics about the application internals and the Go runtime are exposed in a format that Prometheus can scrape and then store. This enables alerting to be built on top of it with something like Alertmanager, a piece of software commonly used in conjunction with Prometheus, in the future. Supporting monitoring via Prometheus is common for both application software as well as infrastructure

software at this point in time so it seemed appropriate for an application trying to follow modern development and operation practices.

2.11 Template project

Before starting the process of writing the application I had an interest in this type of application structure. A lot of the practices and patterns seen here were first tried out and iterated upon in a toy project that served as the starting point. Certain pain points such as creating OpenAPI documentation and how to work with it when there are multiple API versions were first tried out there. As the application developed for this paper evolved and required more features, it became the reference point, and the template project fell behind.

There are some great lessons that can be learned from the development experience of this application and those should be replicated in the template and explored further. The main ones are problems encountered when debugging the frontend and creating a more cohesive configuration mechanism. First, as far as the frontend goes it was by nature difficult to debug. Taking advantage of Go's new structured logging package from the standard library, namely log/slog, would enable adding debug logs that would not be printed when running normally. Right now, the application frontend can only show errors to the user or developer in a limited number of instances, and it depends on where the errors occur. A particularly difficult case is when some part of the authentication fails. Since all HTTP handlers use the same helper function to handle storing and retrieving the cookie that holds the authentication token, the message is not as helpful as it could be. Additionally, the template rendering function requires a lot of parameters, which makes it hard to use and any change to it means refactoring unrelated pieces of code. An improved version of the template would account for both of those and offer ergonomic and idiomatic ways to debug the frontend and work with its code without adding code duplication.

Regarding the configuration, storing everything in a configuration structure that depends on the API version makes some sense. However, this means that some tradeoffs have to be made for the sake of making the application operator's life easier. For example, there aren't any version-dependent environment variables for configuration even though the application can otherwise support using a different database for each API version. This is not even optionally exposed at the moment, which could be implemented by having a MYSQL_URL environment variable and populating all API versions with that value unless a version-specific one is set. Due to the number of possible combinations of databases and API versions, this was not at the top of the priority list. Furthermore, the way each part of the application gets information about the rest is not very ergonomic. In the main part of several packages, there is a lot of setup code where the abstraction of the other components is not as good as it should be, but the refactoring required to fix it was not worth the time investment. The template should be expanded in the future to expose all the configuration options supported in code and at the same time have better abstractions in the code for the configuration of each component.

Overall, the choice to create and use a template project paid off. There are many details that needed to be figured out that were mainly related to the project structure and the ability to expand on the code in the future that would have been a lot more difficult to figure out while also developing the project. This in conjunction with having worked on smaller scale projects and some existing familiarity with the tools, libraries and software ecosystems involved made the experience a lot better.

2.12 Epilogue

The main technologies chosen for implementing this application are the Go programming language, the gRPC communication protocol, and the protocol buffers data format. The frontend was built using HTML text templating in Go, very simple CSS as well as a bit of JavaScript and jQuery only where needed for enhancing the presentation. The technologies were chosen specifically to be explored in the context of developing a microservices architecture web application. In addition, several other technologies were used. Specifically, the MySQL relational database, Docker for packaging and optionally running the application, OAuth2 for authorization and GitHub as the code hosting platform.

Other than existing technologies, the internal code architecture had first been tested out at a smaller scale in a project. That project was then used as the starting point but over the course of development the application written for this paper outgrew what the template had provisions for and became the main place to test out ideas. Each of these technological choices was made due to interesting features that can be argued to offer certain advantages. Other than delivering a final product, the paper aimed to explore how these can affect the implementation of a full-stack web application.

Chapter 3: Application architecture

3.1 Introduction

In this chapter, the internal architecture of the application will be analyzed. The primary focus will on the way the components fit together as well as what each part of the code is supposed to know about. At first, the multiple design patterns will be explained followed by explaining the application architecture along with how the design patterns were applied to it. Additionally, the APIs that the application exposes will be described as well as the details of the data storage part and the web-based user interface.

3.1.1 Domain-driven design

While the timeline is not easy to specify, domain-driven design appears to have been the ideological predecessor of both the hexagonal as well as the microservices architecture. It was first detailed by Eric Evans in a book of the same title in 2003 [4]. It argues for dividing software systems according to fine-grained domains called bounded contexts. Each part should be modelled according to experts in that domain to achieve a technical implementation that closely follows the conventions and concepts that are used in that domain. At the time of release, it was intended to lend itself to implementations using Java classes but in the time since 2003, the concept has been applied to writing small and network-interconnected services. This specific subset of applying the architecture will be explored in more detail in the section about microservices.

3.1.2 Hexagonal architecture

The hexagonal architecture focuses on separating the business logic from the other infrastructure the application uses. It was invented by Alistair Cockburn in 2005 in a blog post [5] with the same name. It was later renamed to "ports and adapters". Both names are still in use and refer to the same concepts so they can be used interchangeably. An application is modelled as a hexagon with business logic in the middle and what are called drivers which could be human users, tests or other applications surrounding it. This design allows the core of the application to be written without depending on concrete implementations of a specific component. It also enables looser coupling between the components thus making them more easily interchangeable. It started as an alternative to a layered architecture where each part of an application could include business logic which in turn meant that the core functionality of the software was spread out and depended on code that it should not have. It is related to but not the same as domain-driven design or microservices architecture which are sometimes referenced together.

3.1.3 Microservices architecture

The microservices architecture defines the way an application should be structured is by having multiple services that are split up based on the subset of business logic they address. It is closely related to domain-driven design in the sense that the domain that a service handles usually maps to a business domain. An official definition does not appear to exist since there wasn't a single point of origin. However, over the years few commonalities emerged. The first of those is that service communicate through the network using communication methods that are independent of the implementation language. Additionally, the size of each service should be small in size to enable further common characteristics such as independent development and deployment cycles and procedures. In turn, these characteristics enable scaling each service independently and a bigger service to be decomposed to microservices over time. There is no official test of how to decide on the size of the service, software

architects need to decide on it in order to get the benefits of the architecture without making the overhead unmanageable.

Developing an application using microservices architecture can be done in several ways. One way to categorize the approaches is synchronous and asynchronous calls between services. A synchronous implementation would look like a user making a request to service A and then that service making a request to services B and C. If either B or C is down, the user's request would fail since service A cannot fetch some data it requires to respond. In contrast, an asynchronous implementation would rely on some external infrastructure, usually an event store, where all services send data to and receive data from. In that case, even if a downstream service is unavailable the exact moment a user makes a request, the data is available to be worked on later after the service has been restarted and is in a position to process the request. These approaches might appear to conflict but the use cases they target are different. If the application needs to fit into a real-time access pattern, it is not feasible to use the asynchronous implementation. On the other hand, if guaranteeing reliability is the most important design goal, using a synchronous implementation where one service in the call chain could be unavailable at a specific point in time and cause non-recoverable failure does not make sense. The synchronous versus asynchronous categorization is only one way to look at implementations of the microservices architecture and it offers insight into designing large-scale distributed systems where the use-case they target can make a big difference in what tradeoffs can be made to fulfill the requirements an application has.

Based on the above, in the case of our semester scheduling application, the synchronous option was implemented. There are a few reasons for this, but it mainly came down to the amount of data being handled by the application and the expected access pattern. A user of the application expects to add, edit or delete an assignment and then have that change take immediate effect. This experience can be offered since the maximum number of people that would use it is about a handful and even then, they will not be using it concurrently. On top of that, the amount of data is miniscule so a single instance of the application can handle all of it without issue. There is no user-visible reason for architecting the application with scalability in mind since the chances that more than one instance will ever be needed are basically zero.

Additionally, the access pattern does not lend itself to an asynchronous approach. This contrasts with something like the workflow of an online store. When following a typical online shopping workflow, a user adds one or more products to the cart, finalizes the shipping details, pays for the products and gets some sort of confirmation that the order has been placed or should wait to hear back about it. This lends itself very well to an asynchronous design. The availability of the products might change by the time someone goes to handle the order, or shipping might be affected by the time the products are at a distribution center, or the price could be different in the case of a price drop for a limited number of the products. The entire process is asynchronous and the validity of the initial request, the order, needs to be checked multiple times until the user receives them.

3.1.4 Backends for frontends

The backends for frontends architecture describes the development of a purpose-specific backend that is tightly coupled with a specific user interface. It was first detailed in 2015 by Sam Newman during the initial rise in popularity of microservices where an increase in mobile clients was also observed. This architecture positions itself against a general-use or general-purpose backend API by proposing the addition of an in-between backend that is closely coupled with a specific user interface. This allows the

general-purpose API backend to remain loosely coupled with the client or frontend while also solving the issue of client-specific needs.

In architectural discussions an example is often helpful to demonstrate the benefits and reasons for choosing said architecture. For backends for frontends, one such example is that of a mobile application. In that instance, the client has a greater constraint when it comes to the number of requests it can make compared to a desktop equivalent. The nature of a mobile device includes a network connection that could be limited by both bandwidth as well as latency and as such prefer to make as few requests as possible to the API and potentially get less information in that response. This conflicts with the fully featured responses that another service or a client without a limited network connection would prefer. In such a case, the backends for frontends architecture can be implemented to create a backend for the mobile application that makes requests to one or more microservices, aggregates the data from the responses and strips out parts of it that the mobile client doesn't need. In this instance the response that the mobile client receives varies a lot in structureure from what the general-purpose API offers but is exactly what that specific user interface needs. In total, this approach retains the benefits of loose coupling between the backend and the frontends that access it while also providing a tailored experience for API consumers with more specific requirements.

3.2 gRPC communication

The gRPC communication component of the application is the interface that the outside world has with the application backend. Despite that, adding a different way to interact with the application such as an alternative communication method does not require major refactoring due to how the application is split into parts. The main parts of this component are the protocol buffers definition, the code in the grpc package and the code in the rest package. The case of gRPC-gateway which provides the REST API is a special one because it is essentially another client with the major difference that it is one that is automatically generated, acts as a reverse proxy and runs and ships together with the application. In this part we will analyze all the parts of the project related to the gRPC communication protocol as well as the protocol buffers format.

3.2.1 Protocol buffers definition

The definition of the gRPC API is stored in the semester_scheduling.proto file located in the proto/semester_scheduling/v1/ directory. This definition can be shared with anyone who wishes to create a client so they can generate the code stubs required for interacting with the application backend. In order to assist with discoverability of the RPCs, gRPC reflection has been enabled to allow clients to get a description of the API using tools such as grpcurl.

The protobuf definition file can be thought of as the contract the application signs with the outside world. With the previous idea in mind certain rules need to be followed so any client written against any given API version can continue to be compatible. It includes everything required to generate server and client stubs in the languages supported by the protobuf compiler.

The file starts by declaring that the third version of the protobuf syntax is used. Under that it declares the protobuf package name, in this instance it is also namespaced with the version to enable implementation of versions in the future without any renaming which would impact any client that have been written. Next are imports of external types that are commonly used in the protobuf ecosystem, especially empty and timestamp are very common. For the purpose of generating gRPC-gateway code, the google/api/annotations.proto import is required and for automatically generating OpenAPI version

2 documentation for the aforementioned HTTP endpoints that gRPC-gateway will handle, the protoc-gen-openapiv2/options/annotations.proto import is also needed. These imports are resolved by the buf CLI tool which is used as the replacement protocol buffers compiler in this project and also helps with importing definition files. After the imports we find two options, one that specifies the Go package name for the automatically generated code and another that configures gRPC-gateway.

For the purpose of supporting a user to perform authentication-related actions without having to use the login API separately from the application in question, a few RPCs are allowed to be called without requiring a valid authorization token. These are attached to a separate protocol buffers service called UniversityAuthenticationService to more clearly indicate that it is functionally a proxy for the login API. Additionally, they enable a more ergonomic way of handling these actions inside the application. The remote procedure calls that require authentication make up the majority of the functionality provided by the application and are part of the separate SemesterSchedulingService. Having this separation allows the code that sets them up to define different characteristics for them such as the requirement for authentication. More details on this are contained in the explanation of the code in the grpc package.

Inside the configuration of the buf protocol buffers compiler, the generated Go code is put in the same directory as the definition itself. The buf documentation examples show the generate code being put into a different directory but a different choice was made for discoverability reasons. In Go, imports work by having a module which usually is a URL to a git repository such as github.com/insanitywholesale/semester-scheduling and then to import packages inside that module, the sub-directory is added. This means that importing the Go models would require adding an import for github.com/insanitywholesale/semester-scheduling/proto/semester_scheduling/v1 which is the same place that the definition for the API can be found. This helps discoverability and includes the API version in the path to make it clearer to the package users what they are working with.

3.2.2 The grpc package

The code in the grpc package inside the grpc/v1 directory has three main parts: the non-authenticated server structure, the authenticated server structure and the authentication function. Each server structure has methods on it that correspond to the RPCs defined in the protocol buffers definition file along with a field that handles the case of the RPCs that have been declared in the definition but not yet implemented in code. Since it is the distinguishing factor between the two server structures, the authentication functionality will be discussed first.

Inside the file grpc/v1/auth.go exists the authentication function, named AuthFunc. It relies on the library github.com/grpc-ecosystem/go-grpc-middleware/auth which handles parsing metadata for gRPC requests. In the main.go file, this function among a couple others is set to be an interceptor so it runs before the requests reach the gRPC server. If a request is made to an RPC that requires authentication, but the user is not authenticated or the token has expired, an error is returned. For the unauthenticated service, a functioned called AuthFuncOverride was created which changed the authentication behavior to enable just passing through the request without validating the token or checking the user profile. This enables removing the requirement that the API caller already has a valid token obtained by using a separate API. Instead, any caller can instead use this application to get a token, refresh it and fetch the user profile without having to implement interaction with the login API separately. These RPCs were created both for convenience during development as well as a demonstration of a gRPC API that has

RPCs that require authorization as well as ones that do not. The interceptor concept that gRPC uses for this, lends itself very well and enables a lot of customization.

3.2.3 The gateway package

The code in the gateway package located inside the gateway/v1 directory is relatively simple but essential to the functionality of the application. It includes a single function which is all that is required to connect to set up gRPC-gateway. It creates a connection to the gRPC server based on the endpoint parameter and returns an object of type http.Handler. It is important to note at this point that this type is standardized throughout the Go ecosystem. An object of type http.Handler generally means something that responds to HTTP requests. In this case, the http.Handler object returned, helps integrate gRPC-gateway with the go-chi HTTP router. Using that object, a connection can be made to an HTTP router, in this case go-chi, which will then handle the normal non-gRPC HTTP requests. The product of that, is offering the REST API as defined by the annotations in the protocol buffers definition. Inside the main.go file there is a call to the CreateGateway function of the gateway package along with the creation of HTTP handlers for serving the documentation files and the frontend. All three of the above are accessible on the same port and separated only by their URL path.

3.2.4 gRPC API description

The gRPC API and the REST mapping on top of it, both follow the API design guidelines set by Google to the extent possible and as much as it makes sense as well as the buf tool's linting rules. For gRPC, this informs everything in the protocol buffers definition from the naming of the services, the naming of the RPCs themselves, the names of the messages and the structureure of the requests and responses.

While RPC guidelines do not necessitate being constrained to the equivalence between them and API endpoints, it is a good pattern to follow which in turn eases developlement as it did in this instance. For example, the RPCs are named the way Google suggests, adding a List verb which is meant to be for getting all entities while the Get verb is used for fetching just one. Some other guidelines have not been followed such as the one that says the Create RPC should have a response type of the object created because it made a bit more sense to follow the general protobuf suggestion of suffixing the RPC name with Request or Response to derive the message name which then enables extending the message in the future without modifying the type.

The API is made up of two protobuf services, one without authentication that allows easily getting and refreshing the access token and the user profile and one with authentication that concerns itself with the actual functionality of the application. These services are called UniversityAuthenticationService and SemesterSchedulingService respectively. The RPCs and the extra options used for defining the mappings that should be generated are the core of the API so they should be explained. The unauthenticated service will be used as an example to explain common concepts since it is a lot shorter.

There are 3 parts to the definition of a fully functioning RPC. First a service needs to be defined, the RPC will be part of this service. Second, messages that define what data the RPC will send and receive and last, the RPC itself. For the sake of brevity, the imports as well as the profile-related parts are omitted from the definition below. With those exceptions in mind, the UniversityAuthenticationService definition inside the proto/semester_scheduling/v1/semester_scheduling.proto file is visible in Listing 1.

```
service UniversityAuthenticationService {
  rpc GetAPIToken(GetAPITokenRequest) returns (GetAPITokenResponse) {
```

```
    option (google.api.http) = {
      post: "/api/v1/token"
      body: "*"
    };
  }
  rpc RefreshAPIToken(RefreshAPITokenRequest) returns (RefreshAPITokenResponse) {
    option (google.api.http) = {
      post: "/api/v1/token/refresh"
      body: "*"
    };
  }
  rpc GetProfile(GetProfileRequest) returns (GetProfileResponse) {
    option (google.api.http) = {
      post: "/api/v1/profile"
      body: "*"
    };
  }
}

message RefreshAPITokenResponse {
  APIToken api_token = 1;
}

message RefreshAPITokenRequest {
  string client_id = 1;
  string client_secret = 2;
  string refresh_token = 3;
  string url = 4;
}

message GetAPITokenResponse {
  APIToken api_token = 1;
}

message GetAPITokenRequest {
  string client_id = 1;
  string client_secret = 2;
  string code = 3;
  string url = 4;
}

message APIToken {
  string access_token = 1;
  string user_id = 2;
  string refresh_token = 3;
}
message GetProfileResponse {
  Profile profile = 1;
}
message GetProfileRequest {
  APIToken api_token = 1;
  string url = 2;
}
```

Listing 1: UniversityAuthenticationService protobuf definition

As detailed in Listing 1, there is the service, the RPCs themselves and the messages. There are a few things that are not intuitive. First, the option inside each RPC. Importing google/api/annotations.proto allows adding the google.api.http option to an RPC and setting some values to define how the gRPC-gateway will generate the HTTP API mappings. The asterisk as the value for body means that the whole request body maps to the protobuf request message, for example the GetAPITokenRequest message. It is possible to specify that it is a specific top-level JSON key whose value maps to the protobuf message. This is the case in some of the RPCs for SemesterSchedulingService.

The next unintuitive thing is the types having an equal sign and a number. This is the field number (a number between 1 and 536.870.911 with some exceptions) and is crucial for generating the language-specific code. It uniquely identifies a field inside a message and should not be changed after it has been set in order to allow clients to continue working in the future without having to regenerate code. The first 16 have are accessible quite a bit faster so it is good practice to either keep the messages shorter than that or give the most used data a field number from 1 to 16. As can be seen in the image above, a field can be declared to have the type of another message, similar to how Go structure fields can be of the type of another structure.

With that explanation in mind, the following are the two services and their RPCs, they are mostly CRUD actions, a couple convenience ones for listing assignments and a very application-specific RPC to copy the assignments:

- UniversityAuthenticationService
- GetAPIToken
- RefreshAPIToken
- GetProfile
- SemesterSchedulingService
- ListSemesters
- GetSemester
- GetCurrentSemester
- GetCurrentPrepareSemester
- CreateSemester
- ListProfessors
- GetProfessor
- CreateProfessor
- UpdateProfessor
- DeleteProfessor
- ListCourses
- GetCourse
- CreateCourse
- DeleteCourse
- ListAssignments
- CreateAssignment
- DeleteAssignment
- ListProfessorAssignments
- UpdateProfessorAssignments

- ListCourseAssignments
- CopySemesterAssignments

As can be gleaned from the above list, the RPCs both physically in the file as well in the list here are generally grouped according to the resource they are related to and to a lesser extent the REST API path they get (there a couple exceptions to that second rule that will be mentioned and expanded at the end of this section). Their messages are named after the RPC, suffixed by either Request or Response as noted earlier. A couple notable exceptions are ListSemesters, GetCurrentSemester and GetCurrentPrepareSemester which take the type google.protobuf.Empty to indicate that they take no parameters. Additionally, the RPCs DeleteCourse and DeleteAssignment return google.protobuf.Empty to indicate that no data is to be sent to the client as a response.

It is worth taking a closer look at RPCs in the main gRPC service, grouped in the file by entity, and see if there is anything special to note. First, the semester RPCs in Listing 2.

```
  rpc ListSemesters(google.protobuf.Empty) returns (ListSemestersResponse) {
    option (google.api.http) = {get: "/api/v1/semesters"};
  }
  rpc GetSemester(GetSemesterRequest) returns (GetSemesterResponse) {
    option (google.api.http) = {get: "/api/v1/semester/{semester_id}"};
  }
  rpc GetCurrentSemester(google.protobuf.Empty) returns (GetCurrentSemesterResponse) {
    option (google.api.http) = {get: "/api/v1/semester/current"};
  }
  rpc GetCurrentPrepareSemester(google.protobuf.Empty) returns
 (GetCurrentPrepareSemesterResponse) {
    option (google.api.http) = {get: "/api/v1/semester/current_prepare"};
  }
  rpc CreateSemester(CreateSemesterRequest) returns (CreateSemesterResponse) {
    option (google.api.http) = {
      post: "/api/v1/semester"
      body: "semester"
    };
  }
```

Listing 2: Semester RPCs

The path parameters in curly braces map to the field of the same name in the protobuf message which is how that value ends up being passed to the code. Additionally, the CreateSemester RPC is an example of the value of a top-level JSON key being taken from the request body instead of it being the entire body. Below the semester, there are the professor RPCs as seen in Listing 3.

```
  rpc ListProfessors(ListProfessorsRequest) returns (ListProfessorsResponse) {
    option (google.api.http) = {get: "/api/v1/professors"};
  }
  rpc GetProfessor(GetProfessorRequest) returns (GetProfessorResponse) {
    option (google.api.http) = {get: "/api/v1/professor/{professor_id}"};
  }
  rpc CreateProfessor(CreateProfessorRequest) returns (CreateProfessorResponse) {
    option (google.api.http) = {
      post: "/api/v1/professor"
      body: "professor"
```

```
    };
  }
  rpc UpdateProfessor(UpdateProfessorRequest) returns (UpdateProfessorResponse) {
    option (google.api.http) = {
      put: "/api/v1/professor/{professor_id}"
      body: "professor"
    };
  }
  rpc DeleteProfessor(DeleteProfessorRequest) returns (DeleteProfessorResponse) {
    option (google.api.http) = {delete: "/api/v1/professor/{professor_id}"};
  }
```

Listing 3: Professor RPCs

Here all the different HTTP verbs are used as well as the same technical detail of the entity creation and update requiring the professor key in the JSON request body. Courses are more of the what has already been mentioned so it will be skipped, same with the assignment CRUD ones.

At the end, the non-standard assignment RPCs are listed. Contrary to the other ones, these do not perfectly follow the CRUD convention, making them quite a bit more interesting, see Listing 4.

```
  rpc ListProfessorAssignments(ListProfessorAssignmentsRequest) returns
(ListProfessorAssignmentsResponse) {
    option (google.api.http) = {get: "/api/v1/professor/{professor_id}/assignments"};
  }
  rpc UpdateProfessorAssignments(UpdateProfessorAssignmentsRequest) returns
(UpdateProfessorAssignmentsResponse) {
    option (google.api.http) = {
      put: "/api/v1/professor/{professor_id}/assignments"
      body: "professor_assignment"
    };
  }

  rpc ListCourseAssignments(ListCourseAssignmentsRequest) returns
(ListCourseAssignmentsResponse) {
    option (google.api.http) = {get: "/api/v1/course/{course_id}/assignments"};
  }

  rpc CopySemesterAssignments(CopySemesterAssignmentsRequest) returns
(CopySemesterAssignmentsResponse) {
    option (google.api.http) = {
      post: "/api/v1/semester/{semester_id}/assignments:copy"
      body: "*"
    };
  }
}
```

Listing 4: Irregular RPCs

The first thing that sticks out is the path mapped to CopySemesterAssignments. According to Google's API Improvement Proposal 136 which references resource-oriented design, custom methods should be used for non-standard actions. The example showcased in that proposal is about archiving a book where the google.api.http option defines a POST with path value ending in :archive. This was emulated here by adding :copy in the end. Looking at it from a semantic point of view, the assignments from a specific

semester are copied which explains well what the endpoint action is except it is not as clear as it could be where the assignments are copied to. This is a mark against the self-documenting nature of the API but there didn't seem to be any ergonomic solutions. The endpoints providing a different view of the assignments have a path that is set to be under the professor or course resource itself, respectively. This can be argued to be incorrect since the assignments are not part of the resource's fields, however it appeared intuitive. The REST API mapping will be discussed in more detail in the following section.

3.2.5 REST API description

The REST API follows a versioned CRUD (create, read, update, delete) convention with a couple notable exceptions. The main entities are the semesters, the courses, the professors, and the assignments. Out of those, only the assignments do not have a unique identifier and are just a combination of the unique identifiers of the other three entities. The endpoints follow the CRUD convention, a GET request on the entity endpoint with no path parameter returns all information available for all existing entities while a path parameter of the unique identifier returns all information for that specific entity. A POST request is used to create a new entity and since the unique identifier is generated on the server side, the entity is returned to the caller with that information filled in. In case the client tries to fill it in, it is simply ignored and discarded. For updating information, PUT requests are used where the information supplied by the client completely replaces the information on the server for a specific entity. Last, a DELETE request can be used to delete a specific entity by supplying its unique identifier as a path parameter. Additionally, any requests made to secured endpoints need to have the “Authorization: Bearer 1234” header since they rely on that bearer token to which is then used as the access token for the university's login API.

Starting off with the endpoints related to the professor entity, the endpoints corresponding to the action mentioned previously are the following:

- Get all information about all professors: /api/v1/professors
- Get all information about a specific professor (replace 123 with the ID of the specific professor): /api/v1/professor/123
- Create a new professor based on the JSON on the POST request body: /api/v1/professor
- Replace all information about an existing professor (use a PUT request and replace 123 with the ID of the specific professor): /api/v1/professor/123
- Delete a professor (use a DELETE request and replace 123 with the ID of the specific professor): /api/v1/professor/123

An example professor JSON object is structured in the way seen in Listing 5.

```
{
  "professor": {
    "id": "1",
    "surname": "Αδαμίδης",
    "name": "Αδαμίδης Π.",
    "full_name": "Αδαμίδης Παναγιώτης",
    "uname": "adamidis",
    "is_admin": false,
    "is_permanent": true,
    "is_virtual": false,
    "description": "adamidis@it.teithe.gr",
```

```
    "default_full_hours": "6",
    "sex": "SEX_MALE"
  }
}
```

Listing 5: Professor JSON response

This is the response returned when a properly authorized GET request is made to /api/v1/professor/1 by a client. The sex field has an enum-like value since the value of that field in the protocol buffers message is defined that way. This will be visible in several other cases since it is a common pattern used for the project.

The other entities are structured the same way but have some notable differences in their endpoints. Courses are also uncomplicated entities with just one attribute to be filtered on, so they are remarkably similar to the professor endpoints with the obvious exception of their API endpoints having the /api/v1/course(s) prefix. The following are the course-related endpoints:

- Get all information about all courses: /api/v1/courses
- Optional query parameter semester_number which limits the results returned according to which semester the course is registered to be in (1st, 2nd, 3rd, 4th semester for example)
- Get all information about a specific course: /api/v1/course/123
- Create a new course from the JSON off the POST request's body: /api/v1/course
- Delete a course (DELETE request): /api/v1/course/123

An example semester JSON object is structured in the way seen in Listing 6.

```
{
  "course": {
    "id": "1503",
    "kind": "COURSE_KIND_THEORY",
    "type": "COURSE_TYPE_MANDATORY",
    "semester_type": "SEMESTER_TYPE_WINTER",
    "semester_number": "5",
    "is_active": true,
    "description": "Σχεδίαση Λειτουργικών Συστημάτων"
  }
}
```

Listing 6: Course JSON response

This is the response returned when a properly authorized GET request is made to /api/v1/course/1503 by a client. Here again we see that the course kind and type as well as the semester type are enumeration values, as is the case in the professor example response.

For semesters, a couple of endpoints have been provided to serve as useful shortcuts. These endpoints allow the client to easily retrieve the semester that has the current or current_prepare flag set to true. The endpoints are located at /api/v1/semester/current and /api/v1/semester/current_prepare so sending a GET request to each saves the client from retrieving all the semesters and then checking one by one to find the semester if those flags are set to the appropriate value. This is a minor optimization that could

save a bit of network bandwidth as well as slightly reduce the load on the database. With the above information in mind, this is the list of endpoints related to semesters:

- Get all information about all semesters: /api/v1/semesters
- Get all information about a specific semester: /api/v1/semester/123
- Get the semester marked as current: /api/v1/semester/current
- Get the semester marked as current_prepare: /api/v1/semester/current_prepare
- Create a new semester from the JSON off the POST request's body: /api/v1/semester

An example semester JSON object is structured in the way seen in Listing 7.


```
{
  "semester": {
    "id": "32",
    "name": "2022-23EAP",
    "name_full": "Ακαδημαικό έτος 2022-23, Εαρινό Εξάμηνο",
    "semester_start_year": "2022",
    "exam_period": "EXAM_PERIOD_SPRING",
    "lessons_start_date": null,
    "lessons_end_date": 2023-06-02T00:00:00Z",
    "exams_start_date": 2023-06-05T00:00:00Z",
    "exams_end_date": 2023-06-30T00:00:00Z",
    "is_hidden": false,
    "is_current": true,
    "is_current_prepare": true
  }
}
```


Listing 7: Semester JSON response

This is the response returned when a properly authorized GET request is made to /api/v1/semester/current_prepare by a client at the time of writing. Here we can see that the current_prepare flag is set to true, therefore this is the semester object that was returned. In this specific instance, with the data used for testing, this would have been returned if the request had been sent to /api/v1/semester/current_prepare instead since both the current and current_prepare flags are set for this semester.

Finally, the piece that brings together all the above, the assignment entity. As previously mentioned, an assignment is not unique on its own but is instead a combination of 3 unique identifiers. This made its API endpoints a bit more complicated to design, model and build. As will be discussed in the frontend section, there are two ways that assignments will be used. First, seeing all the courses a professor has been assigned as well as the same thing from the course's point of view. To handle this, there exist a few logical-only entities. This means that they're part of the API but are not stored in the database in the same way. Inside the gRPC API they are referenced as ProfessorAssignments and CourseAssignments respectively. In turn, this means that the endpoints related to assignments are different to a significant degree from the other entities mentioned so far. Creating just the basic CRUD operations and figuring out a way to make them useable was a design challenge, especially considering the lack of a unique identifier. This will be mentioned in a later section but it's worth noting here as well, the experience of working with such objects leads towards each entity having a unique identifier

being a generally good practice even if it's just a UUIDv4 string. There needs to be some way to tell them apart and reference them using a single attribute rather than a combination. Additionally, the API was built one endpoint at a time depending on the needs of the project, it was not all laid out beforehand to avoid committing to functionality that would not be used or would not make sense for the use case. A very prescient example of such an endpoint is the one that copies the assignments from one semester to the one marked as current_prepare. With the details mentioned above in mind, the API endpoints related to the assignments are the following:

- Get all information about all assignments: /api/v1/assignments
- Optional query parameter semester_id which limits the results returned according to which semester the assignment is for
- Create a new assignment from the JSON off the POST request's body: /api/v1/assignment
- Get all assignments for a specific professor: /api/v1/professor/123/assignments
- Optional query parameter semester_id which limits the results returned according to which semester the assignment is for
- Update assignments for a specific professor for a semester: /api/v1/professor/123/assignments
- Required query parameter semester_id which specifies for which semester the change should take effect
- Get all assignments of a specific course: /api/v1/course/123/assignments
- Optional query parameter semester_id which limits the results returned according to which semester the assignment is for
- Copy all assignments from a specific semester to the one currently being prepared: /api/v1/semester/123/assignments:copy

An example JSON object of a semester's assignments is structured in the way seen in Listing 8.

```
{
  "assignments": [
    {
      "professor_id": "1",
      "course_id": "1205",
      "semester_id": "30",
      "assigned_hours": "4",
      "teaching_hours": "4",
      "emvolimi_only": true
    },
    ...
  ]
}
```

Listing 8: Assignments JSON response

There isn't a lot to note here, it's a top-level JSON key with a list of assignment object as the value. In this case the request had the optional query parameter semester_id set to 30 so only assignments for that semester were returned. An interesting case is that of copying. After creating a new semester and changing that to be the current_prepare one, a request can be made to copy over assignments from a previous semester. Through a database query, the assignments will be copied and just have their semester identifier changed. The response sent to the client is a list of those newly created assignments

and uses the same format as the one displayed above. This endpoint has been highlighted in the description of both APIs since it is an obvious edge case, that of using a custom method and even outside of that, it was a major functional requirement making it an important and noteworthy feature of the application.

After having covered the endpoints corresponding to SemesterSchedulingService which is authenticated, it is important to also look at the 3 endpoints that are meant for getting and refreshing the access token. These are the following:

- Get an access token from the login API by sending a POST with the application ID, secret, code and URL: /api/v1/token
- Refresh the access token by sending a POST with all the same information as the above except the code is swapped with a refresh token: /api/v1/token/refresh
- Fetch the profile of the user based on the provided token: /api/v1/profile

This concludes the description of the REST API. The two APIs are obviously very closely connected and it provides some value to go through them in the order they were developed which is also how they are in the stack, the gRPC API exists first and based on annotations the REST API is then generated so it makes sense when presenting them to go in that order.

3.3 Data storage

The data storage component, sometimes called data repository, exists is the next component to explore. The main purpose it has is to encapsulate anything related to persistent data storage. It consists of an interface that the rest of the codebase uses instead of the actual types of the database object to enable flexibility during runtime as well as make adding support for different storage systems as straightforward as possible.

3.3.1 SemesterSchedulingRepo interface

The interface definition which determines the array of functions a database implementation needs to have, is located inside the models/v1/repo.go file, and currently only has one concrete implementation for MySQL. The main structure representing the database should be called Repo and all methods are attached to it. The function to create a new object, namely NewMysqlRepo, inside the file repo/mysql/v1/repo.go has a signature that signifies its return type is SemesterSchedulingRepo meaning it can be used wherever the interface is used. Strictly speaking, that return type is not required. If there is a mismatch between a variable of the type of the interface and the program tries to assign an object that does not match it, compilation will fail but it is a small precaution that allows the problem to be caught even if the code doing the assignment described previously has not been written yet. The files other than repo.go inside the directory mentioned are one file per model so the relevant code is easy to find, for example the database methods related to the Course type are inside the course.go file.

3.3.2 MySQL implementation

There are a few database-specific details inside the repo/mysql/v1 directory along with the code that implements the previously mentioned interface. The first one of these is the migrations directory where the export of the current database along with a few other migrations is located. Inside repo/mysql/v1/repo.go the private function newMysqlClient takes the URL to connect to as a parameter and returns a pointer to an sql.DB object which is a type from the standard library and an error. The return value provides access to the database and can be used to run SQL commands. The public function

NewMysqlRepo calls the private one and stores the connection URL as well as the database client object in an instance of the Repo object on which all database methods are implemented. Lastly it returns that to the caller in order to provide an object that implements the SemesterSchedulingRepo interface using MySQL. There is a function inside utils/v1/repo.go which allows the database to be dynamically chosen. Only MySQL exists, but if there was an additional PostgreSQL implementation, that file would be the place to add a check if that URL is populated and choose that instead.

The database object is passed to the createGRPCServer function inside main.go and it eventually ends up being stored in the gRPC server structureures. From there it is passed as an argument to the domain logic functions. This way the gRPC and logic components have no knowledge of what the database is or anything else about it other than that it satisfies the SemesterSchedulingRepo interface. Additionally, the logic component does not store the object since there is no concept of a logic unit or a logic server or something to that extent while gRPC server structures do exist.

In the signature of the methods on the Repo structure, only the custom models are used. The database part of the code does not care that the application is using gRPC to expose its API. The gRPC and protocol buffers part of the code handles the responsibility of converting between the format it understands and the models that the rest of the application knows about. This way we could replace either gRPC and protocol buffers as the communication method or MySQL as the database and the rest of the code would not need to be changed. In hexagonal architecture terms, the logic part has a port for a storage system to connect and in this case, MySQL is what connects to it.

Another important element in the part of the code that sends SQL queries to the database and retrieves the results, is the sqlscan library [6]which was used extensively. This library relies on another one by the same author, called dbscan [7]. On its own, dbscan is responsible for mapping a dbscan. Rows object to the correct fields of a Go structure. This is achieved by looking at the structure tags for that field, specifically the “db” tag. Even if a field does not have an explicit tag, it translates the structure field name to snake case and does a lookup this way. This functionality is very useful on its own but sqlscan makes it more convenient to use. It achieves this by making enhanced functions similar to the ones from the db/sql package of the standard library that also have dbscan functionality. The developer can then very easily execute a query and map the results to structures without having several loops and variables in between, resulting in less redundant and easier to read code. Throughout the project, sqlscan was used for all SQL queries, was immensely useful and improved the development experience to a great degree.

3.3.3 Database schema

The database schema is made up of tables that roughly correspond to the Go models. For the sake of brevity, the most important tables will be mentioned in detail. First, there is the semester table. It has a unique identifying number, its type (winter, spring or September), its short and full name, the year it starts, the dates it begins and ends, the dates its exam period begins and ends and some Booleans detailing if it should be hidden, if it is the current one and if it is the one the staff is preparing for meaning it is the upcoming one.

Next up is the course table. It has a unique identifier, the name of the course, if it mandatory or optional, information about the semester it is taught in and the nature of the course such as being a lab, theory or virtual which is a special case and last it has a Boolean denoting if it is active or not. These tables describe just two of the main entities that the application deals with.

Following the semesters and the courses, the application also handles data related to professors and assignments. The prof table specifically holds information about professors while the prof_type table,

renamed to prof_sem in a migration, describes the employment type of a professor for a semester. The prof table has general information such as a unique identifier, the professor’s name, email and username as well as information about their administrator’s permission status, if their employment is permanent and if they are a virtual user. Additionally, it stores their default full teaching hours for a week which can be set to something else in the prof_sem table. That table stores data about the employment type of a professor for a semester such as the professor and semester, the actual full hours and if the employment is full-time.

For combining all of the previously mentioned entities, the assignment table is used. This table describes an assignment of a course to a professor for a given semester with a few more details such as how many hours the assignment is for and how many of those hours are for lectures. Using the data in the semester, course and professor tables as well as the information in the prof_sem table, the application can calculate if a professor has too many courses assigned to them or not enough of the hours specified in their employment agreement have been filled in. Either of those conditions would mean that the professor’s contract is not properly fulfilled so it is important to note such cases. Putting all of those together makes up the core of what the application needs to do to fulfill its purpose.

The current scope is covered by the existing tables and provisions exist for expanding the capabilities to handle other duties. One such case could be if contracts were able to be encoded in a table, the existing functionality could be easily adjusted to show only the ones that have active contracts. Similar to that, if a professor is on sabbatical leave, a new table could be added together with some minor code changes to automatically change their default full hours to zero for example.

When describing database schemas and more specifically the connections between the entities and the tables that represent them, it is useful to showcase the full diagram with all entity relations. This hopefully helps anyone working with it in the future to get a bird’s eye view of how the tables connect which in turn sheds light on how they are used and why. The database schema can be visualized in the way seen in Figure 2.

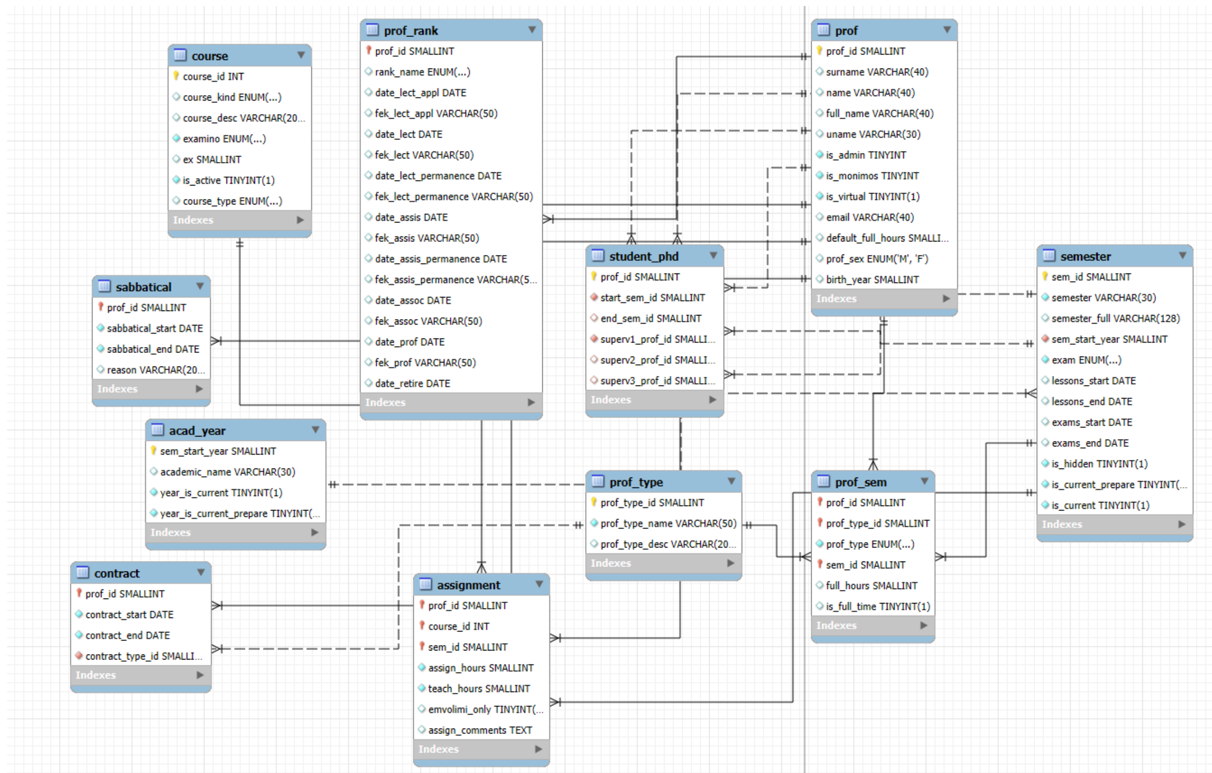

Figure 2: Database schema main tables

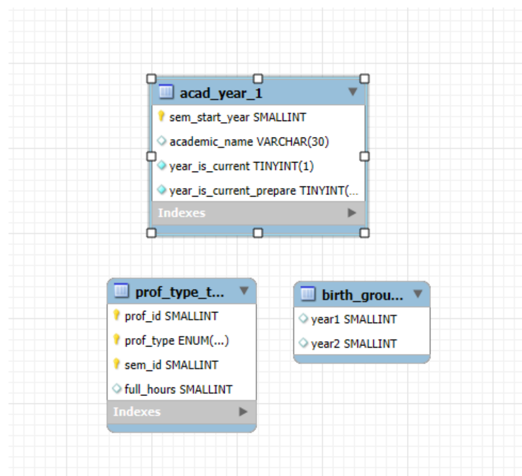

Figure 3: Database schema secondary tables

In Figure 2 and Figure 3 the relationships between the tables are clearer and the visual representation better demonstrates how closely interconnected the main entities that the application deals with are. Some provisions have been made for the future expansion examples mentioned above to demonstrate how those would relate and because there was a possibility of them entering the scope of the project but ended up not being included in it.

With an understanding of the gRPC communication layer, including the protocol buffers definition of the API as well as the REST mapping to it, getting a into the database schema demonstrates how the API was derived from it. A few design choices were made for the sake of keeping the API idiomatic, this mainly consisted of not carrying over the foreign key relationships between entities, while also

making it easy to write code that maps one to the other. With the two ends, outward API and internal data storage, explained it makes sense to move on to the layer that makes them work seamlessly together.

3.4 Domain logic

The next part of the application is the domain logic component, alternatively referred to as the business logic or the core depending on the design pattern. In a general context, its purpose is to encapsulate the logic of what the application should do and how it should function without containing implementation details and specifically without depending on them. For example, the core or business logic should not contain the notion of inserting a row into a database but instead use semantics such as updating or deleting a user. Since for this specific application there did not exist a lot of complicated business rules, in the vast majority of cases the logic component calls the corresponding data repository method on the provided database object. The logic component is implemented in the logic package which exists under the logic/v1 directory.

To facilitate the separation of the communication protocol and the data repository from the logic component a few general types were created. To better understand the purpose of the custom models it is useful to know that when writing the application, the protocol buffers definition file was written first. Because of that, the automatically generated types did not perfectly suit the pre-existing database schema and in cases where a different data storage technology was used the types that are idiomatic for a protocol buffers definition might be entirely misaligned with idioms that the data store uses. To accommodate such a case, the custom models represented by Go structures have a closer relationship to the relational database schema and their fields are annotated with the corresponding structure field tags. This enables the Go sqlscan library to use the structure field tags mentioned previously to map table columns to an instance of the structure. The custom models in the models/v1 directory helped bridge gaps where those existed between types generated automatically from the protocol buffers definition using the buf tool and the existing database schema in MySQL throughout the codebase.

3.5 Frontend

The web interface of the application satisfies the functional requirements set out and also contains a few interesting architectural choices. It has been implemented using plain HTML, CSS and JavaScript along with a couple JQuery libraries. Despite the technology stack being more traditional, writing the frontend to use gRPC to talk directly to the backend instead of going through gRPC-gateway as well as the use of Go templates to provide server-side rendering makes it an equally noteworthy part of the application.

At a high level, HTML forms are used to submit data that HTTP handlers in the backend for the frontend translate to RPC calls to the actual backend. The frontend is organized by version the same way as the API with its HTTP router being inside the frontend/frontend.go file. For routing requests, go-chi is used with the same middleware as in main.go. The CreateFrontendHandler function takes in a pointer to AppConfig instance which stores the address of the backend and calls the equivalent for each version of the frontend. Currently only a single version exists which is the default when visiting the frontend at the root URL path but is also available under the /ui and /ui/v1 paths. If a second API version and frontend version are created, this function should be updated so the root and ui paths point to that new version and a /ui/v2 handler should be added. The version 1 of the frontend includes another HTTP router for handling requests for each page and for submitting forms. This design allows a single HTTP router to be exposed to the rest of the application and to the outside in general while internally handing the routing to different frontend versions.

Looking inside the version 1 frontend directory, the implementation has been split across a few files. The frontend/v1/frontend.go holds the functions related to handling setup and configuration of the connection to the backend, the creation of the HTTP router mentioned previously as well as the HTML templates that are rendered and sent to the client. The errors.go file inside the same sub-directory defines the error types that the frontend will return, they're in a separate file for organizational purposes. Next, the helpers.go has a few convenience functions for handling operations related to storing, retrieving and validating the token as well as rendering templates. The last file, handlers.go, is where the main part of the frontend functionality exists. Each function is mapped to an API route and is responsible for either rendering a page or handling an HTTP POST request from an HTML form.

The templates directory and the templates contained inside is worth exploring a bit as well. There is a base template named base.html that includes the general structureure of an HTML document that the frontend should return. Inside the body tags, some room is left for 3 sections to be filled in, navigation, body and footer. In the majority of instances, the handlers just pass the specific template related to their functionality to the convenience function which in turn assembles the files in the correct order and renders the template. The navigation and footer could have been included in the base template but they've been left separate since at one point during development a different navigation was shown depending on if the user was logged in or not which was later removed. The case of different pages being rendered depending on the user being logged in or not still exists for the main page which is why there is a plain index.html file as well as a logged-in-index.html equivalent. The rest of the templates are generally self-explanatory and are named based on the page they are for. Last, an additional detail worth mentioning as part of this description is that the templates, just like the database migration files, are embedded into the binary at compile time using the Go embedded filesystem feature.

The design detailed above is an implementation of the backends for frontends architecture. For a web client that is not a single-page application and doesn't use JavaScript to interact with the backend API, it made sense to use HTML forms for sending data from the client to the server. However, since the data needed to be parsed in some cases and transformed into others, from what was convenient in a form to the shape of the data defined by the protocol buffers definition, an in-between backend was written for the specific purpose of handing frontend data. It is not intended to be a public API and adheres to the requirements that the gRPC API has. As the needs of the frontend change, the interface provided by the backend for the frontend can change together to support it. This approach allowed the creation of a tailored experience for the development of the frontend while keeping the public API unchanged.

3.5.1 Per-page analysis

Each page is interesting, starting with the homepage. The default state is that of a user that has not yet logged in. In that instance, all the user sees is the navigation, a short bit of text prompting them to log in and the footer. Attempting to navigate to any of the pages in the navigation except the "About" one will redirect the user to a login. The pages are supposed to be accessible only by staff with the administrator role therefore any other user should not have access to them. In Figure 4, a user's first experience when visiting can be seen.

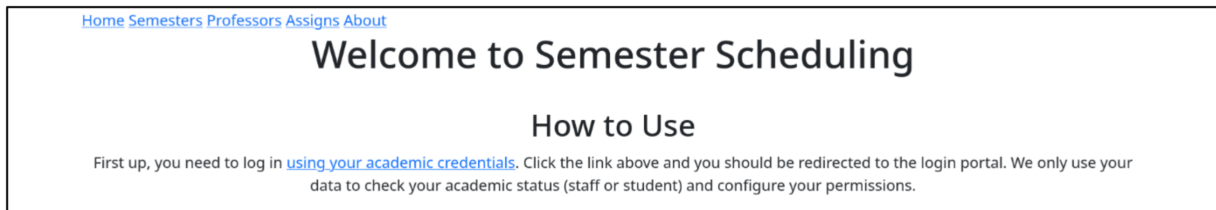

Figure 4: Logged out homepage

After following the link that the instructions point the user towards and then logging in, the page the user returns to is what can be viewed in Figure 5.

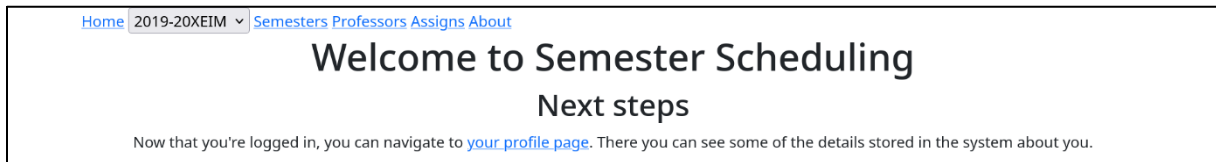

Figure 5: Logged in homepage

A few notable things have changed. First, a semester dropdown has been added to the navigation. The frontend retrieves the semesters from the backend so a properly authorized and signed in user is required (as previously mentioned, the RPCs that have to do with anything except login functionality are secured behind the authorized gRPC service). Next, the "How to Use" section prompts the user to look at their profile. This shows some basic user information taken from the login API and can be used to verify that the application is correctly communicating with it. For example, the profile page can be seen in Figure 6.

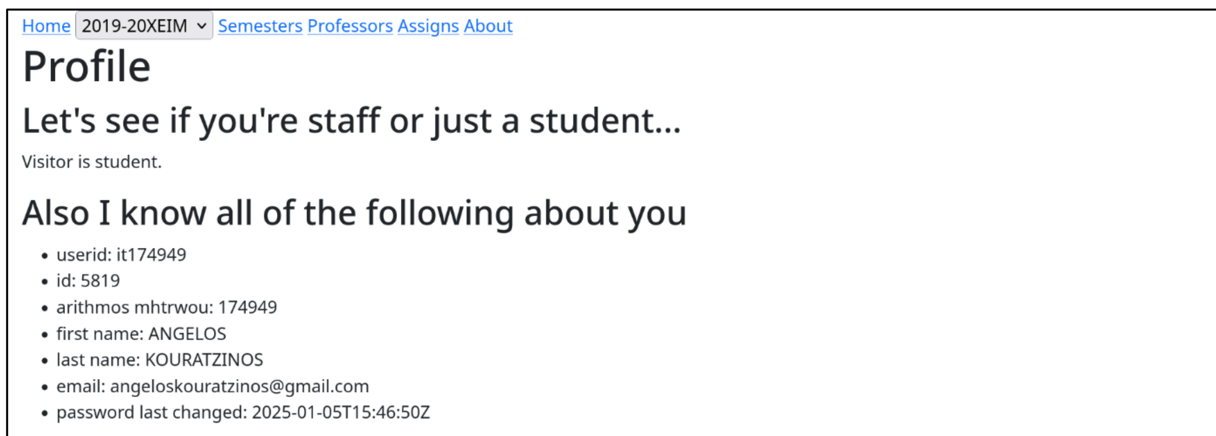

Figure 6: Profile page

It does not contain the entire JSON response, only enough information to allow verification of the login process and communication of the two applications. It is mainly a feature meant for debugging during development but can also be useful for troubleshooting issues in production, so it has been left in.

The next page in the order that they're listed in the navigation bar, is the semesters page. When clicking the link, the user is taken to an overview page of all the semesters. It doesn't have specific dates or every detail that is stored but serves as an easy to look at all of the semesters. Any hidden ones are not omitted

in this page. An administrator would see something like what is visible in Figure 7 when visiting this page:

Home 2019-20XEIM Semesters Professors Assigns About

All semesters

Create Semester

ID	Name	Full Name	View Details
15	2019-20XEIM	Ακαδημαϊκό έτος 2019-20, Χειμερινό Εξάμηνο	View
16	2019-20EAP	Ακαδημαϊκό έτος 2019-20, Εαρινό Εξάμηνο	View
19	2020-21XEIM	Ακαδημαϊκό έτος 2020-21, Χειμερινό Εξάμηνο	View
20	2020-21EAP	Ακαδημαϊκό έτος 2020-21, Εαρινό Εξάμηνο	View
24	2021-22XEIM	Ακαδημαϊκό έτος 2021-22, Χειμερινό Εξάμηνο	View
26	2021-22EAP	Ακαδημαϊκό έτος 2021-22, Εαρινό Εξάμηνο	View
30	2022-23XEIM	Ακαδημαϊκό έτος 2022-23, Χειμερινό Εξάμηνο	View
32	2022-23EAP	Ακαδημαϊκό έτος 2022-23, Εαρινό Εξάμηνο	View
36	2023-24XEIM	Ακαδημαϊκό έτος 2023-24, Χειμερινό Εξάμηνο	View

Figure 7: Semester overview page

The page starts with a header, after which there is a button to navigate to a page for creating a new semester and then finally there is the table with the information. More details about a specific semester can be accessed by clicking on the View link in the corresponding row. The detailed view is visible in Figure 8.

:

Home 2019-20XEIM Semesters Professors Assigns About

Semester 2022-23EAP

ID	Name	Full Name	Start Year	Exam Period	Lessons Start	Lessons End	Exams Start	Exams End	Ac
32	2022-23EAP	Ακαδημαϊκό έτος 2022-23, Εαρινό Εξάμηνο	2022	E	1970-01-01T00:00:00Z	2023-06-02T00:00:00Z	2023-06-05T00:00:00Z	2023-06-30T00:00:00Z	tru

Figure 8: Individual semester page

Due to the length of the dates, there is a bit of overflow that is not visible by default, but the table is scrollable and the rest of the columns can be views as evidenced in Figure 9.

Home 2019-20XEIM Semesters Professors Assigns About

Semester 2022-23EAP

	Full Name	Start Year	Exam Period	Lessons Start	Lessons End	Exams Start	Exams End	Active	Prepare
P	Ακαδημαϊκό έτος 2022-23, Εαρινό Εξάμηνο	2022	E	1970-01-01T00:00:00Z	2023-06-02T00:00:00Z	2023-06-05T00:00:00Z	2023-06-30T00:00:00Z	true	true

Figure 9: Individual semester page text overflow

In the semesters overview page, the create semester button was mentioned. That takes the user to a new page where they get the option to create a new semester using an HTML form. If the user decided to

navigate to the new semester creation page instead of the overview, they would see something resembling Figure 10.

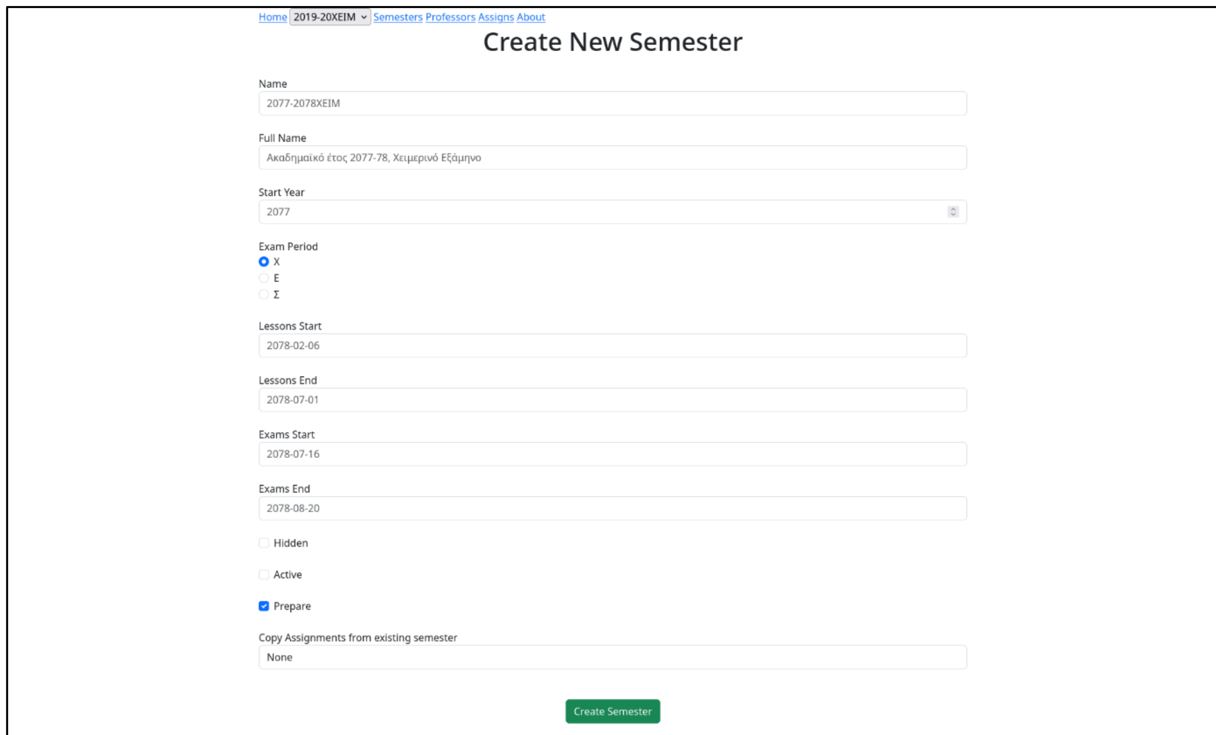

Figure 10: Semester creation page

All the attributes can be set, just as if this was an insert statement in the database. However, one additional option is presented here for convenience. The user, in this case a university administrator, can click on the dropdown in the bottom and select a specific semester to copy assignments from and initialize this new semester based on assignments that already exist, see Figure 11.

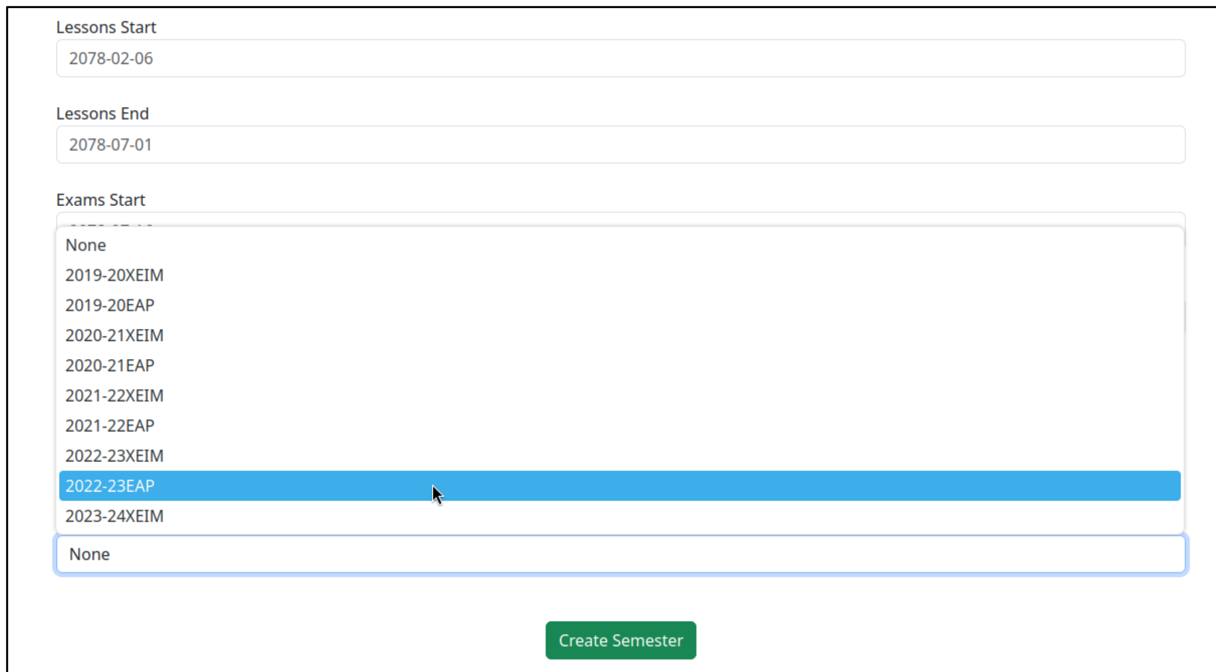

Figure 11: Semester creation copy assignments dropdown

This is quite helpful and saves a significant amount of time since the user only has to change anything that might be different this time or nothing at all if no changes are required. For example, if all the same staff has the same availability and the workload is equivalent to last year's, the task is over with right after clicking the green "Create Semester" button to finish creating the semester. It's worth noting that appropriate HTML tags have been used for each field, not just free-form text entry as to minimize the chance for accidental mistakes and ensure the process is completed successfully without data errors. Semesters are a pretty simple entity in the application, so their respective pages are also appropriately simple.

Next, again following the navigation bar order, are professors. The pages largely resemble the semester ones, except for the number of entries. Here again we have an overview page with the same structure, the most important details such as unique identifiers and names are visible for each professor along with a link to the detailed view page, see Figure 12.

Home 2019-20XEIM Semesters Professors Assigns About

All professors

Create Professor

ID	Name	Surname	Username	Action
0	-	-		View
1	Αδαμίδης Π.	Αδαμίδης	adamidis	View
2	Αμανατιάδης Δ.	Αμανατιάδης	dima	View
3	Γιακουστίδης Κ.	Γιακουστίδης	kgiak	View
4	Γουλιάνας Κ.	Γουλιάνας	gouliana	View
5	Δεληγιάννης Ι.	Δεληγιάννης	ignatios	View
6	Κεραμόπουλος Ε.	Κεραμόπουλος	euclid	View
7	Κώστογλου Β.	Κώστογλου	vkostogl	View
8	Ράπτης Π.	Ράπτης	praptis	View
9	Σαλαμπάσης Μ.	Σαλαμπάσης	msa	View
10	Σιάκα Κ.	Σιάκα	siaka	View
11	Σφέτσος Π.	Σφέτσος	sfetsos	View
12	Σιδηρόπουλος Α.	Σιδηρόπουλος	asidirop	View
13	Αντωνίου Σ.	Αντωνίου	antoniou	View
14	Βίτσας Β.	Βίτσας	vitsas	View
15	Δέρβος Δ.	Δέρβος	dad	View
16	Διαμαντάρας Κ.	Διαμαντάρας	kdiamant	View
17	Κλεφτούρης Δ.	Κλεφτούρης	klefturi	View

Figure 12: Professor overview page

That detailed view page does not include a ton more information but it does follow the style of the equivalent one about semesters like in Figure 13.

Home 2019-20XEIM Semesters Professors Assigns About

Professor

ID	Name	Surname	Username	Email	Hours	Permanent	Virtual	Admin
2	Αμανατιάδης Δ.	Αμανατιάδης	dima	dima@it.teithe.gr	12	true	false	false

Figure 13: Individual professor page

Due to the number of columns as well as the length of the data in each, the table tends to not have any overflow and as such does not need a scrollbar. The ”Create Professor” page in Figure 14 is also very similar. When a new staff member needs to be added, the administrator can fill in the form visible in that figure.

Figure 14: Professor creation page

The fields have sensible placeholder values, these do not need to be erased they just disappear after something has been set, again to nudge the user towards correct data entry, and at the end there are three checkboxes for extra attributes that a professor might need to have. These are all the same as the database. There didn’t seem to be a need for manually setting the number uniquely identifying a professor or for automatically generating their username, so these have been left as-is. Again, a pretty simple entity with a few pages for completing the necessary tasks regarding their data.

Where the complexity ramps up, is the assignments page. As mentioned in previous parts, the assignments need to be viewed both from the perspective of the professor as well as the perspective of the course. This is because two different questions need to be answered, first does the professor have all their hours correctly filled up according to what their contract says, and second do all the courses that need to be part of the semester have a professor assigned to teach them. For this purpose, the overview page for the assignments utilizes two tables. Referencing the API, these tables are the professor assignments and the course assignments. As a reminder, these entities are entirely logical within the application code and do not exist in the database structure. They are meant to serve as convenient ways to translate between the tables and this way of viewing the assignment entries. Again, for convenience the tables here have the search functionality enabled to allow finding results without needing to flip through several pages of results to find the one that needs to be edited. Additionally, most columns are

sortable by clicking on the column name. For the administrator that has access to it and wants to get a view of all of them, the page looks like Figure 15.

Home 2022-23XEIM Semesters Professors Assigns About

Assignments

Professor Assignments

Create Assignment

Show 10 entries Search:

Professor ID	Professor Name	Course IDs	Assigned Hours	Teaching Hours	Full Hours	Assigned Percent	Edit
1	Αδαμίδης Παναγιώτης	1205,1945,1950,59001	11	11	6	183	Edit
2	Αμανατιάδης Δημήτριος	17011	12	12	12	100	Edit
4	Γουλιάνας Κων/νος	1102,1641,1802	14	14	6	233	Edit
6	Κεραμόπουλος Ευκλείδης	1505,1743,1950	8	8	6	133	Edit
7	Κώστογλου Βασίλειος	1942,59001	5	5	6	83	Edit
9	Σαλαμπάσης Μιχαήλ	1405,59001	5	5	6	83	Edit
12	Σιδηρόπουλος Αντώνιος	1403,1941,59001	9	9	6	150	Edit
13	Αντωνίου Στάθης	1101,1201,1302	12	12	6	200	Edit
14	Βίτσας Βασίλειος	1701	4	4	6	66	Edit
15	Δέρβος Δημήτριος	1401,1741,1743	9	9	6	150	Edit

Showing 1 to 10 of 46 entries Previous 1 2 3 4 5 Next

Course Assignments

Show 10 entries Search:

Course ID	Course Name	Professor IDs	Course Kind	Course Type	Semester Type	Semester Number
1201	Μαθηματικά II	13,91	Θ	YP	E	2
1202	Μετρήσεις και Κυκλώματα Εναλλασσόμενου Ρεύματος	84,92	Θ	YP	E	2
1203	Τεχνική Συγγραφή, Παρουσίαση και Ορολογία Ξένης Γλώσσας	20	Θ	YP	E	2
1204	Σχεδίαση Ψηφιακών Συστημάτων	82,86	Θ	YP	E	2
1205	Αντικειμενοστρεφής Προγραμματισμός	1,79	Θ	YP	E	2
1304	Οργάνωση και Αρχιτεκτονική Υπολογιστικών Συστημάτων	16	Θ	YP	E	4
1401	Συστήματα Διαχείρισης Βάσεων Δεδομένων	15	Θ	YP	E	4
1402	Τηλεπικοινωνιακά Συστήματα	121	Θ	YP	E	4
1403	Εισαγωγή στα Λειτουργικά Συστήματα	12	Θ	YP	E	4
1404	Ηλεκτρονικά Κυκλώματα	87,92	Θ	YP	E	4

Showing 1 to 10 of 20 entries Previous 1 2 Next

Figure 15: Assignments overview page

In order to fit all the data in while making the page easy to view and navigate, the course and professor numerical identifiers have been used instead of full names or titles. A notable difference here is that instead of just a ”View” link, in the professor assignments table, there is a link for each assignment

which takes the user to a page where the assignment can be edited. Clicking the ”Edit” link then brings the user to a page that can be seen in Figure 16.

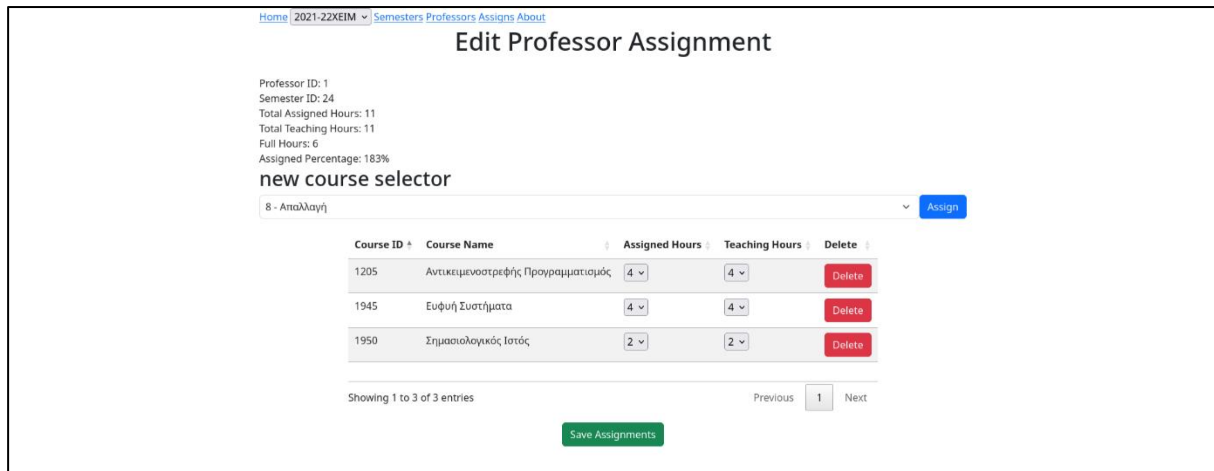

Figure 16: Assignment edit page

The edit page starts by listing the assignment data at the top, most important of those for the administrator is going to be the assigned percentage. Further down the page, there is a dropdown selector for assigning a course to the professor for this semester, thus editing the assignment. Below both of those is a table showing the currently assigned courses along with how many assigned and teaching hours each one is assigned for. The dropdown can be used as expected, after all the desired changes have been made, clicking the save assignment button makes those changes permanent. While in the other two pages the dropdown in the navbar was not of any use, in the assignment pages the dropdown changes the selected semester and adds the appropriate query parameter to show the corresponding results. This works when looking at the overview as well as in the edit page for more easily checking what courses the professor had been assigned in the past.

Last, there is the about page that just has a short description of what the application does, see Figure 17.

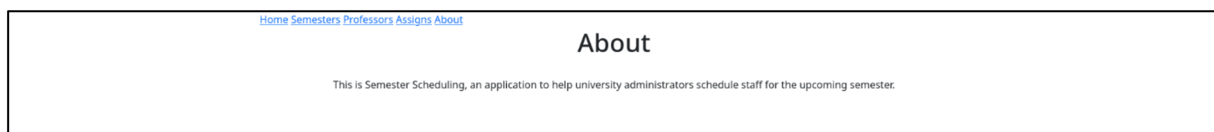

Figure 17: About page

In this section, the frontend was analyzed in more detail. The several pages created, their structureure and functionality was detailed along with what makes each page unique. This highlighted the usefulness of the datatables library [8] for displaying, sorting, and searching through results as well as making them more readable. Additionally, the approach of using HTML forms and POST requests to the backends for frontends was viewed from a different angle. With this understanding, the most user-facing part of the application has been detailed.

3.6 Utils package

For certain recurring operations such as conversions between types or timestamps, a utils package was created. There are two categories of utility functions employed in this application, ones that are general-

use and ones that are specific to one API version. The functions inside the non-versioned utils directory are not related to the API but rather the standard library and aim to make a few operations easier. In contrast, functions converting between the custom Go types and the types generated based on the protocol buffers definition are very specific to that API version and as such are stores separately. They are located inside the corresponding versioned API directory such as utils/v1 for clarity. The decision to put type conversion functions in a separate package instead of inside the grpc one was made because there is potential for them to be applied in more ways than just abstracting away the gRPC API code since they are related to the data format and not necessarily the communication protocol. For example, someone could decide to refactor the application to use a raw TCP socket for data transport while retaining the protocol buffers data format. In that case the conversion functions would still be applicable while the gRPC code would not be. This is not a likely scenario but on a code organization level it seemed like the most technically sound option.

Starting with the general utility functions, there are a few of them that deal with creating instances of net.Listener based on some parameters or extracting information from a net.Listener that has already been created. On one hand, ListenerFromAnything creates a listener from either a port or a Unix socket path by calling the respective ListenerFromPort or ListenerFromSocket. On the other hand, AnythingFromListener extracts the port or Unix socket path from the listener passed as a parameter by calling PortFromListener or PathFromListener by checking the address of that listener. In neither of those categories there is another function, CreateRandomListener, that creates a listener that listens on a random available port and is mostly useful for tests.

During earlier stages of development, before moving to the go-migrate migration tool, a couple of functions for working with SQL files were created. For the purpose of reading the large SQL files of the database dump and returning a string, StringsFromFileSplit and StringFromFileChunked were written. StringsFromFileSplit reads a file at the specified path, splits it based on the supplied delimiter and returns an array of strings. The file is read in chunks of the supplied size with the help of StringFromFileChunked. The functions are as of now unused but not deleted since they can become useful again in the future.

The last function in this category is CreateNullTime. This is a conversion function that takes an instance of time.Time and if it is the timestamp equivalent of zero, it returns an sql.NullTime indicating that the timestamp is null in SQL. This function’s usage is in the API version specific conversion functions because dates in the custom models are stored as sql.NullTime to account for dates potentially being left empty in the database and there was a need to fill those date fields with the equivalent value extracted from the protocol buffers value.

The utility functions specific to the API version are split into three categories, type conversion, creating a SemesterSchedulingRepo instance as well as application configuration from environment variables. Each of those is significant enough on its own and worth going over. First, the type conversion. Facilitating the translation between generated code structures and hand-written model structures was important since each endpoint must do it at least once. Initially this was written to be part of the logic layer but conceptually it was not the right place in the chain for type conversion. Later on, the calls to the conversion functions of the utility package were moved to the gRPC layer, specifically the grpc package. The functions themselves are also layered in the sense that single-object conversion was written first and then when a slice or array needed to be handled, functions calling the single conversion in a loop were added.

The function that creates the database object is also important and is placed in the versioned directory because it is conceivable although not likely that at some point in the future it could be desirable to have a different database for different API versions and the SemesterSchedulingRepo interface is specific to an API version. A similar reason is why the configuration is also in the versioned directory. The configuration structure could change with a new API version and the v1 configuration should remain unchanged and not be affected by that addition. In general, if something uses types that are specific to an API version, it needs to be in a directory that ends with the API version, so the package makes it obvious to any consumers that what is contained inside it is not universally applicable.

3.7 Configuration

The application configuration is done exclusively using environment variables. According to the twelve-factor app methodology, one aspect of well-built applications is separating configuration from code. The recommended way to achieve this is to set any required configuration in the environment that the application runs in instead of inside a file that would be checked into a revision control system such as git. In this application, the login details such as the client identifier and secret key, the URL for the login API, the URL for token-related actions, the URL for getting the user profile as well as the URI for the database are all taken from environment variables. In the event where one of them needs to change, they are already independent as far as the code is concerned and each one can be changed individually.

The approach of using environment variables means that they should be set when deploying the application which can be achieved in a variety of different ways by using any number of different tools depending on the execution environment. For example, when deploying using only a compiled binary release of the application, they can be set inside of a systemd unit file while a deployment using Docker could set them as flags in the docker run command or inside a docker compose file. Additionally, an example of setting the variables is included in the README file at the root of the application source code repository for future reference. In the end, this approach allows the application to be independent of its execution environment while at the same time adapting to it by employing the use of environment variables for configuration.

3.8 Documentation

Documentation was an important part of this project, both to provide relevant information to the API consumers as well as demonstrate a couple different methods of approaching the task. Three formats of documentation are included, two markdown files in the repository with the source code of the project, online automatically generated OpenAPI version 2 interactive documentation as well as documentation generated by the ReDoc tool and an equivalent PDF. The two markdown files in the repository are README.md and ARCHITECTURE.md, one containing more general information and a quick example for getting started with local development while the architecture one briefly explains a few key concepts about the internal application architecture. The automatically generated documentation that is produced by a protocol buffers compiler plugin is available as noted in the section about gRPC-gateway. Finally, online ReDoc documentation was added to showcase another often-used tool except the Swagger UI and the output was converted to a PDF and made available too. A Java program in the form of a Java archive file was required to convert the OpenAPI specification to a PDF so it is included along with the other project files inside the bin directory.

The online documentation provides a way for an API user to learn what endpoints are available as well as try out a request and examine the response. In the case of the usual Swagger UI, the files for it are included inside the openapiv2/semester_scheduling/v1 directory. The automatically created file is

semester_scheduling.swagger.json and will be regenerated when generating code from the protocol buffers definition. Contrary to that, ReDoc only needs one file, redoc-static.html, which is inside the same directory. The main difference between the two approaches is that ReDoc is packaged as a command-line tool while the Swagger UI is released in a few different forms. The NPM module and React component would not be fitting for the project so swagger-ui-dist was used. This comes in the form of plain Javascript files that were added manually to the openapiv2/semester_scheduling/v1 directory and are embedded in the compiled binary's embedded filesystem. This is useful for cases where server-side rendering without external dependencies is required or NodeJS modules from NPM cannot be used which is the case here. This documentation is mainly developer-focused and has helped with showing an overview of the endpoints. Since it is not very time-consuming to add, the benefit nearly always outweighs the one or two hours it takes to properly add it.

Overall, the project includes several different types of documentation. This was mostly done as a technology showcase, but it ended up being used for the actual intended purpose more than a handful of times. The result of this exercise is that it seems to almost always be worth it to add the automatically generated API documentation, but it doesn't replace developer-written instructureions that cover other details of the project. The deployment instructureions, the names of the environment variables, the allowed values for them as well as their purpose should be written manually and the same goes for architecture notes. These can be difficult to keep in sync with the code so developers need pay extra attention to them when adding new features as well as perform regular audits and update of the existing documentation, both of those processes should be put in place to ensure the accuracy of the documentation.

3.9 Request-response examples

Due to the separation and decoupling of components it can sometimes be difficult to conceptualize how a request is handled. For this purpose and to more clearly demonstrate the separate parts of the application working together two examples will be analyzed in this section, one request to the HTTP API created by gRPC-gateway and one request to the frontend.

First, a request adding an assignment sent to the HTTP API. Assuming the application defaults have not been changed, the main HTTP listener of the application is listening on port 9000 when running in all-in-one mode. Inside main.go, the request can be considered to arrive at the listener called restListener and is handled by an HTTP handler that is a go-chi request router defined inside the createRESTServer function. Assuming the request was made to the appropriate endpoint, /api/v1/assignment, the router skips the /api/v1/docs endpoint and routes it to the /api route which is handled by restHandler which is an HTTP handler returned by the CreateGateway function inside the gateway package. Since that request is an authenticated one it will go to the SemesterSchedulingService protocol buffers service and call the CreateAssignment RPC. The bearer token from the HTTP request will be set in the metadata context of the gRPC request by gRPC-gateway and then make the gRPC request. When the gRPC server gets that request, the authentication interceptor gets it first and runs AuthFunc inside grpc/v1/auth.go which retrieves the user profile to check if the user corresponding to the token has the necessary authorization to do so. If that conditional check succeeds, a new context with metadata including the token is created and the requests moves on to the actual gRPC server. More specifically, inside grpc/v1/assignment.go, the CreateAssignment method is called which first transforms the assignment data from the data structureure generated based on the protocol buffers definition to the format of the hand-written Go model. Afterwards, it calls the AddAssignment logic function with the database handler as well as the request data. Following that, inside /logic/v1/assignment.go the AddAssignment function calls the

StoreAssignment method on the supplied database object with the assignment data as the argument. For its part, the StoreAssignment method inside mysql/v1/assignment.go executes an SQL INSERT statement with the provided data and returns the assignment as it was stored. The data is then passed backwards through the same points and returned to the client.

An example frontend request contains a lot of similarities with the above example. As done in the above example, a correct request will be assumed where the user has logged in successfully and has the required authorization to perform the desired action. This time, the request will be about adding a new professor. In the frontend, from the navigation at the top or directly using the /professors path a user can navigate to the page to view all professors. From there, clicking the “Create Professor” button will navigate the user to a page with an HTML form where a new professor can be added. First the form needs to be filled in and at the end submitted using the “Create Professor“ button at the end. This form submit executes a POST request to /createprofessor. Since there is a cookie with the login token stored, it will be sent along with the form data to the form handler mentioned previously. This request will be sent again at the same port and the same listener inside the code except this time it will be routed to frontendHandler in the router of the createRESTServer inside the main.go file. That handler is created by the CreateFrontendHandler function inside frontend/frontend.go which then forwards it to the router inside frontend/v1/frontend.go. The router includes a router for a POST request at the /createprofessor path which is to be handled by the CreateProfessor function. This handler can be considered the backend for the frontend as its responsibility is to handle the transformation of HTML form data and handling the request to the general-purpose backend. This is exactly what the CreateProfessor function does, it checks the token like most of the HTTP handlers in the same file and then if the user is authorized it performs an RPC call to the backend. After that, the same chain of function calls takes place, the token is authenticated again by the backend since someone could have made the POST request without using the frontend, then the CreateProfessor method of the gRPC server is called which calls the logic function that then calls the database method. If the data was correct and nothing malfunctioned, the user will then be redirected to the page listing all professors with the new one listed at the end.

The examples above leaned more into adding too much detail rather than too little with the intent of demonstrating a request being handled by the application. With the explanation included above, the reader should have gained a more thorough insight regarding the connection between the components mentioned as separate parts in this chapter and how they work together to service a user request. One important aspect is how much of the process is the same between the two examples with the main differentiating factor being the user interface, on one hand the HTTP API and on the other hand the frontend based on Go templates.

3.10 Continuous Integration

The application has quite a few provisions for being tested and quality controlled as part of a continuous integration process. In general, continuous integration is the practice of making sure that the entirety of the application code is in a working state. This usually consists of at least a few of the following: making sure it compiles, has a consistent code style, has a minimum percentage of statements tested, the unit and integration tests pass, a deployable artifact can be created. This usually runs for every commit in a git repository or at the very least every time a feature branch is merged. For the application written for this paper, support for both GitHub Actions and GitLab CI has been added.

The application code passes through dozens of linters to ensure that the Go code adheres to best practices as much as possible. The ability to compile as well as formatting issues are checked as the first step.

Following that, Go's built-in vet tool does a more comprehensive check followed up by golangci-lint, a meta-linter, which ensures an even more thorough check of the codebase. After those checks, the unit tests are executed with a real MySQL instance to ensure functionality has not been impacted by a change. These are done on every commit which helped in spotting issues with features earlier in the development process. One example of this was an issue with the update operation in a repository layer function. The test showed that the logic was incorrect in one edge case and that problem was resolved before the REST API endpoint was written. Due to its advantages, continuous integration is one best practice that was applied for this project and had a positive impact on the development process.

3.11 Epilogue

This chapter went into great depth and presented different angles of the application architecture, its internals, as well as the different ways that the users it is meant to serve will interact with it. The premise initially was to apply some interesting architectural patterns to a significant extent and then evaluate the experience. A few architectures were mentioned and kept in mind when developing the application. First, is domain-driven design which informed the very fundamental modelling of the entities and their relationships. Next, the hexagonal architecture affected how the programming language was used and how the internal architecture looked after the abstractions were put in place. Outside the scope of a single service, the microservices architecture was applied in an optional way here. This minimizes the tradeoffs usually associated with it, allowing an exploration of the concept without complicating deployment. Finally, the backends for frontends architecture was chosen when writing the frontend code to provide an API specific to that user experience. Architectures are by nature quite abstract when described on their own, but the component-specific references expand on where and how those were applied as well as what code snippets the architecture influenced.

Moving beyond the architecture, each major part of the application was analyzed. Starting from the gRPC API and the protocol buffers definition file. The section about gRPC focused on describing the way the communication layer is implemented and what component is responsible for what functionality. This includes gRPC-gateway which generates the code that makes an HTTP API available according to the value of some settings specified in the protobuf definition file. Following that, both the gRPC and HTTP REST API were described in detail.

The abstraction of the repository layer is also important since it is where the data crucial to the functioning of the application ends up before being passed to a specific storage technology. In the data storage section, the way interfaces allow the implementation of some architectural patterns was described along with the database schema. In addition to the above, some very useful tools such as the sqlscan library were explained to demonstrate certain idioms in the Go ecosystem and how they were applied in this case. The repository layer and more generally data storage, can be thought of as the opposite end of the API. The API is how data gets into the application and the data storage is where it ends up, leaving the in-between to be discussed next.

The logic layer which sits between the API and the data storage layer was discussed in the section following the above. This is the place where business logic fits in, where the information known about the domain is written down and is not tied to any specific implementation of any other component according to the hexagonal architecture. Essentially this is the core of what the application is or rather what it does, the purpose it serves and the context in which it operates. However, in this instance there was not much to be done as the database took care of most of the heavy lifting or the rules were on purpose supposed to be enforced by a knowledgeable person. The way it is laid out in code makes it

quite ergonomic to call from different higher-level abstractions and thus enforce the separation of concerns in each layer.

The frontend was left last since it is the most easily changed component. It could theoretically not exist, and the application would have still been usable by a subset of users. However, having a web-based frontend was part of the functional requirements set forth at the beginning. Not to be outdone by other parts of the project, the frontend also has architectural significance. The backends for frontends architecture was mostly oriented towards mobile clients at the time it came out but its core concepts and reason for existing can be just as easily applied to other types of clients. In this case, HTML forms and POST requests from those that then call the gRPC API. As such, it remains separate and does utilize the same API that the backend exposes but there is an intermediary that makes the process easier. This the backends for frontend architecture and in this project, it allowed taking advantage of the Go standard library, standard HTML forms and the language's HTML templating capabilities to translate what the user does client-side to a gRPC request to the backend. No change was required on the backend such as creating specific endpoints there, it is all handled on what the backend sees as the client end from its perspective, but the frontend part of the codebase is the one that has that translation layer. It ended up being more complex than initially theorized and provided some great lessons.

The interesting parts of the code structure did not stop at just those. The purpose-specific utilities, the code for parsing the configuration and the integrated documentation for the API also included noteworthy features. When taken together, the application ended up structured in the way initially envisioned and worked as expected for the end users while having interesting architectural features and providing an incredibly valuable learning experience.

Chapter 4: Conclusions and future work

The application, just as most software projects, has areas that could be improved while still aligning with the original goals and all of them would be welcome additions if they were to be contributed to the original project. Some of the improvements were omitted due to inability to implement them in the allotted time while others were clearly outside of the scope.

The first improvement would be to minimize the overhead of component communication when running in the “all in one” mode. This could be achieved by changing the functions in the main.go file to accept a UNIX socket instead of an address and a port.

Next, more tests could be added so the test suite is more complete. The logic part of the application is not very complex, but a few places have tests already because it reduced the manual effort to verify the correctness of some functionality. The front end is a bit more difficult to test but still relatively straightforward due to the use of standard HTML forms. These forms could be populated with a range of test data, in the form of table-driven tests which are common in Go, to verify that even corner cases are handled appropriately. Currently, the tests are geared towards the right path while the error handling and its correctness are less well covered. Aside from the error handling in general, a few database functions have multiple conditional checks that depend on the input parameters and there aren’t enough tests to check all possible combinations. Some of them have been manually verified but that’s not repeatable or automated the way unit tests inside GitHub Actions or GitLab CI are.

In a related area, a lot of validation was purposefully not implemented in order to allow flexibility in the data entered. Basics such as verifying that the numbers represented years are valid, checking that a semester doesn’t span more than a year and similar cases were not handled. After the application has been in use for a while, it would be clear which parts of the application do not need to offer such flexibility and then verification and validation steps of user-supplied information could be added. The HTML forms already try to use the appropriate input to limit how invalid the data can be, for example the forms do not allow letters in number fields, but it is not the strictest of checks. Anyone could override it by editing the client-side HTML or simply by using the API. Since there is conversion from strings to numbers, the problem is partly alleviated, however proper data validation would be beneficial.

As mentioned earlier, Go 1.22 provides routing functionality that could perhaps replace what go-chi is used for. In the context of this application there aren’t any overly complex routing rules so what the standard library offers from 1.22 and onward is more than enough. This change can be seen as a security improvement as well as lessened maintenance overhead. Additionally, it would make the project a bit more future proof since the language itself is more likely to be around in 10-20 years than a specific HTTP routing library. Not a necessary fix by any means but it would improve the state of the project.

Tangentially related to request routing is the ability to have the REST API and gRPC server listen on the same port. A request demultiplexer could be written fairly easily since gRPC requests set the content-type header to application/grpc and can be routed according to that. An alternative to that would be the cmux[9] project which was built with the express purpose of implementing this functionality. It was not implemented since it could introduce problems for marginally non-compliant gRPC clients and further complicate the configuration functions. The addition of this feature was left as a potential nice to have but not necessarily required and it got pushed back in favor of other higher priority optional features like adding CRUD for as many entities as possible.

As far as other extensions to the application are concerned, more could be done with the contract and sabbatical tables. Those tables are not utilized in the current version, but their data could be used for automatic actions. One example would be to not apply the “copy previous year’s semester assignments” operation to them and perhaps show an extra column indicating their sabbatical status on the page listing all professors. This is outside of the scope of this project, so it wasn’t done but the idea was discussed and a migration adding those tables exists.

The assignments API was structureured according to the needs of the frontend. This means that it is not as general-purpose as it could or maybe should otherwise be. In a potential second version of the API, this could be addressed by having universally unique identifiers for the assignments and then implementing all the CRUD operations on the assignment entity. A few more query parameters could also be added to aid in filtering the assignments in more useful ways. The current design fulfills only the necessary use case of the specific needs the application has but due to the lack of general utility, it is less suitable as a building block for other applications.

On the operational side, as mentioned in the overview of Docker, the project includes a Dockerfile but not a Helm chart. For fully covering the container deployment use case it would be helpful if a Helm chart could be written that supports running in all the different deployment modes. As a more proof-of-concept version, it would be beneficial to have the option of running it in Kubernetes and testing the scalability of it. If this was added to the project, it would make it even more well-aligned with current deployment practices. Like the previous improvement it’s not something broken but it would improve the project to be more feature complete.

An additional Docker-related improvement would be to build a Docker image on every commit and push it to the GitHub Container Registry. Due to pre-existing familiarity with GitLab CI this feature was included in the GitLab CI configuration but not in the GitHub Actions equivalent. This would improve the project by better integrating with the platform it’s hosted on.

Related to using GitHub and GitHub Actions more, the process of releasing a version could be automated. An often-used tool in Go projects is goreleaser which can build multi-platform Go binaries and create GitHub releases. One tricky part is handling the API version, which is noted inside the protocol buffers definition file, correlating with the application version while also applying it to the Docker image as well. This would improve the application of DevOps best practices in the project and enable a more automated workflow.

The authentication-related code should be abstracted more. At the current version there is no separation of the authentication implementation details and the rest of the application code. In specific, the frontend and to a lesser extent the AuthFunc function in the grpc package are tailor-made for the university OAuth2 API without any other authentication methods offered as alternatives or even different OAuth2 providers tested. The time format parsing is also specific to that API which is not ideal. To more closely follow the hexagonal architecture, authentication should be modular as well.

The metrics for Prometheus are only the baseline ones created by default. In addition to these, a lot of metrics based on the use case could be generated. Examples for this can include exposing the assignments as a metric, counting and exporting the number of requests to endpoints on the authorized-only gRPC server without a valid token. Additionally, for adding better monitoring, the database could also expose metrics in Prometheus format but monitoring the surrounding infrastructure was outside the scope of the project, so it was not implemented.

One of the big improvements made in the gRPC code generation is the introduction of the Opaque API in Go protobuf. This contrasts with allowing direct field access that the earlier versions of the plugin did. Currently, it is possible to set a value to a structure generated from the protobuf definition. The opaque API provides a different paradigm which is much closer to how access to private class fields would be handled in an object-oriented codebase written in a language like Java. This allows the decoupling of the generated code API from the in-memory representation of protobuf messages. As a result, it is possible to end up with code that uses less memory. Additionally, the opaque API allows distinguishing between unset and zero values which is a common pain point with Go in some situations. These advantages do have a cost, significant refactoring is required to adapt code to use the new way of using generated code. To ease that workload, there is backwards compatibility provided so the transition between the current paradigm and the new one can be smoother. If the application started being written today, this would be something done by default so if there were any updates done to the code, this should be one of them.

The OpenAPI (formerly known as Swagger) online interactive documentation applies to the HTTP API but there seems to be an equivalent for gRPC APIs called grpcui. It is available in a couple different forms and its purpose is to enable developers to explore the protobuf schema of a gRPC service as well as make requests to the API. For easy integration, there is the github.com/fullstorydev/grpcui/standalone Go package which provides an HTTP handler that provides the web-based user interface including all HTML, CSS, jQuery and image resources. Therefore, this appears to be the best way to go about adding it to the application. Since gRPC APIs using protobuf for messages tend to be a bit opaque by nature, this would allow for easier experimentation without having to build one time use clients to explore the API. It can also be added in a version-agnostic way to enable exploring all API versions. This would be a nice addition which would carry over a similar experience to what is provided by the OpenAPI web-based user interface to the gRPC API.

BIBLIOGRAPHY

[1] G. Authors, “Go,” https://go.dev/. Accessed: Jan. 21, 2025. [Online]. Available: https://go.dev/

[2] gRPC Authors, “gRPC,” https://grpc.io/. Accessed: Sep. 17, 2024. [Online]. Available: https://grpc.io/

[3] J. Volz and B. Rabenstein, “Prometheus: Monitoring at SoundCloud,” https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud. Accessed: Jan. 21, 2025. [Online]. Available: https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud

[4] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1st ed. Addison-Wesley Professional, 2003. Accessed: Jan. 21, 2025. [Online]. Available: https://ia802901.us.archive.org/9/items/ebooks_202003/Eric%20Evans/Domain-driven%20Design%20%281503%29/Domain-Driven%20Design_%20Tackling%20Complexity%20in%20the%20Heart%20of%20Software.pdf

[5] A. Cockburn, “Hexagonal architecture,” https://alistair.cockburn.us/hexagonal-architecture/. Accessed: Sep. 17, 2024. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/

[6] G. Savva, “sqlscan library,” https://pkg.go.dev/github.com/georgysavva/scany/sqlscan. Accessed: Jan. 21, 2025. [Online]. Available: https://pkg.go.dev/github.com/georgysavva/scany/sqlscan

[7] G. Savva, “dbscan library,” https://pkg.go.dev/github.com/georgysavva/scany/dbscan. Accessed: Jan. 21, 2025. [Online]. Available: https://pkg.go.dev/github.com/georgysavva/scany/dbscan

[8] SpryMedia, “datatables library,” https://datatables.net/. Accessed: Jan. 21, 2025. [Online]. Available: https://datatables.net/

[9] S. Hassas Yeganeh, “cmux library,” https://pkg.go.dev/github.com/soheilhy/cmux. Accessed: Jan. 21, 2025. [Online]. Available: https://pkg.go.dev/github.com/soheilhy/cmux