jamk.fi

Developing a Web Service

Databases, Security and Access Control

Markus Moilanen

Bachelor's thesis
January 2019
Technology, communication and transport
Degree Programme in Media Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Moilanen, Markus	Bachelor's thesis	January 2019
		Language of publication: English
	Number of pages: 75	Permission for web publication: x

Title of publication
Developing a Web Service
Databases, Security and Access Control

Degree programme
Media Engineering

Supervisor(s)
Rantala, Ari
Manninen, Pasi

Assigned by
Protacon Solutions Oy

Abstract

The goal of the project, assigned by Protacon Solution Oy, was to develop work safety familiarization software called *TyPe*. The web application and service were built for companies to test their employees' knowledge of occupational topics. The various common, yet critical aspects of developing the web service included database design and management, security measures and an access control implementation.

The web service was built using the popular PHP framework Symfony. It uses a MariaDB database to store application data, such as courses, companies and users. On top of that, JSON Web Token authentication was implemented for managing logged in users and authentication alongside a set of firewalls and guards. Additionally, a customized heavyweight access control implementation was created due to the complex project requirements. The thesis covers the most essential aspects of developing said features, focusing on the critical issues and solutions to them.

The primary result of the thesis is the product itself, which includes the web application and web service. Along with the concrete results, the developers gained immensely useful experience as well as new programming techniques and knowledge.

In the end, the project successfully fulfilled the specified requirements. There are certainly more improvements that can be made to the software, even though certain aspects of it received positive feedback from the clients. As a result of the successful release, more features will be added when necessary.

Keywords/tags (subjects)
Web Service, Database, Security, Access Control, Software, PHP, Symfony, Doctrine, Backend, Back-end, REST, JWT

Miscellaneous

jamk.fi

Kuvaus

Tekijä(t)	Julkaisun laji	Päivämäärä
Moilanen, Markus	Opinnäytetyö, AMK	Helmikuu 2019
		Julkaisun kieli Englanti
	Sivumäärä 75	Verkkojulkaisulupa myönnetty: x

Työn nimi

Web-palvelun kehittäminen

Tietokannat, turvallisuus ja kulunvalvonta

Tutkinto-ohjelma

Mediatekniikka

Työn ohjaaja(t)

Ari Rantala

Pasi Manninen

Toimeksiantaja(t)

Protacon Solutions Oy

Tiivistelmä

Protacon Solutions Oy:lle tehdyn opinnäytetyön tavoitteena oli kehittää työturvallisuusperehdytysohjelmisto nimeltä *TyPe*. Rakennetun web-sovelluksen ja –palvelun avulla yritykset voivat testata työntekijöidensä tietämystä heidän työnkuvaansa liittyen. Web-palvelun kehittämiseen kuului useita kriittisiä ominaisuuksia, kuten tietokannan suunnittelua ja hallintaa, turvallisuustoimenpiteitä ja kulunvalvontatoteutuksen kehitystä.

Web-palvelu rakennettiin käyttäen suosittua PHP-sovelluskehystä Symfonyä. Sovellustietojen, kuten koulutusten, yritysten ja käyttäjien, tallentamiseen käytettiin lisäksi MariaDB-tietokantaa. Kirjautuneiden käyttäjien hallintaan ja todentamiseen käytettiin JSON Web Token –autentikaatiota ja konfiguroitiin palomuurit tarvittavilla asetuksilla. Projektin monimutkaisen vaatimusmäärityksen takia siihen kehitettiin tavallista raskaampi mukautettu kulunvalvontatoteutus. Opinnäytetyössä käydään läpi tärkeimmät ongelmat ja ratkaisut näiden aihealueiden toteutuksesta.

Opinnäytetyön ensisijainen tulos oli itse web-sovelluksen ja –palvelun sisältävä ohjelmistotuote. Konkreettisten tulosten lisäksi projektin kehittäjät saivat suunnattoman tärkeää kokemusta ja oppivat uusia ohjelmointitekniikoita.

Loppujen lopuksi projektin tulos täytti tarpeelliset vaatimukset onnistuneesti. Vaikka ohjelmistoon voidaan toki tehdä parannuksia, on se saanut positiivista palautetta asiakkailta muun muassa käytettävyyden kannalta. Menestyksekkään julkaisun ansiosta uusia toiminnallisuuksia kehitetään tarvittaessa.

Avainsanat (avainsanat)

Web-palvelu, tietokanta, turvallisuus, oikeuksien hallinta, ohjelmisto, PHP, Symfony, Backend, Back-end, REST

Muut tiedot

Contents

Figures

Terminology

API

Application Programming Interface allows different programs or machines to communicate with each other by sending data, for instance an endpoint on a server.

Backend/Frontend

Backend refers to server-side implementations such as databases and web services whereas frontend indicates client-side implementations, for instance web applications or websites.

Database

Database is a collection of data that is stored on a computer. Web services use databases to manage information including users' login information or products in an online store.

Framework

Frameworks are extensions to programming languages, which are used for developing most modern web and mobile software. They allow developers to save time by utilizing prebuilt features and reducing the need for boilerplate code. Occasionally frameworks offer better maintainability and security depending on the framework.

Git

A common version control system, or VCS. Git saves changes made to the code and keeps track of all the changes throughout the project. Web-based third-party hosting services, such as GitHub and GitLab, store the changes on a remote server instead of only on the local machine.

HTTP

Hypertext Transfer Protocol is a stateless protocol for sending hypertext documents, including HTML, scripts, JSON and XML among others.

JSON

JavaScript Object Notation is a light-weight format for describing data. Despite its name, it can be used by any number of programming languages outside of JavaScript.

The data in JSON files is formatted as key/value pairs similarly to properties in a programming object. JSON that is sent over the Internet can be parsed to objects for use in the application code. Due to its simplicity, it can also be easily read and written by any human.

Middleware

Middleware is a broad term for software between two or more other software which connects them together. When it comes to web software, middleware can be a communication framework that sends requests from an application to the server.

Regular Expression

A pattern for finding a specific character sequence within text.

Singleton

A class that can only be instantiated once in the application runtime. Commonly used pattern with services that are used widely inside an application.

SQL

Query languages, with the most common being SQL, are used to request information from or save it to a database. The results of a query can then be used in the web server code and sent back to the client.

State

In computer science, state means globally stored information in an application. The information can be of previous application events or backend response data. The properties of an object are not considered part of the application state.

URL

Uniform Resource Locator is a web address for a resource which can be accessed through a browser or by sending a request. It consists of the protocol such as HTTP, host name and domain name such as *www.google.com*, as well as any route specifications following them. URLs point to a specific IP address.

Web server

Web server can mean either the computer hardware, software or both of them running in unison. Essentially, web server hardware is used to host software, such as APIs or static assets like websites, which are accessed through the Internet.

XML

Extensible Markup Language is a format for describing data. XML documents can be used to specify all sorts of things e.g., information about a person or a machine. Structurally, XML is a collection of tags which can nest more tags and attributes inside of them, similarly to HTML file structure. XML can be sent over the Internet and converted into another format by using an XML parser, for instance by a web browser or server.

1 Introduction

1.1 Background

The working life of a software developer is one of ever-changing variables. Brand new technologies and techniques such as programming languages, architectural styles and design patterns are constantly coming out and being improved further. Having a good understanding of the technologies that are relevant to one's work is imperative to developing modern high-quality products in the software world. Although legacy skill and knowledge cannot be underestimated, learning new things is also necessary.

Every once in a while, when working on a software project, a developer may find the selected technologies or a particular solution to be insufficient. If the need to switch technologies is apparent early on, it can be viable to try something else, although it can be problematic. Frankly, a great deal of it boils down to lack of knowledge and experience in certain areas and can be solved by proper planning as well as information gathering. That is when the understanding of different technologies and techniques comes into play.

When it comes to developing a web service, there are several important matters to consider. First of all, there are multiple types of databases for different purposes as well as techniques for using them. Selecting the correct one for each use-case will most certainly help in the long run for both development costs and learning purposes. On top of that, there are various ways to handle security and user rights including authenticators, firewalls, user roles and access control. Although many web applications only have public data, in reality, a considerable amount requires setting up a complex web of roles and rights.

These issues became a reality during the development of a software project. It turned out that the requirements were more complicated than previously thought, resulting in some unforeseen consequences. What are the best methods to use when encountering such problems? Are there any counter-measures to actively mitigate the issues? The thesis goes in-depth into the development of the project, specifically on the server-side programming.

1.2 Project Premise

The goal of the project was to develop a software system for familiarization with work safety, abbreviated *TyPe*. It includes a frontend web application and a backend web service. In a nutshell, the application should allow the users to create *courses*, which are essentially pop quizzes that are used to test other users' knowledge of various topics. The exam results and other information can be viewed and managed by administrative users. In practice, the amount of potential use-cases for the application is huge due to the customization options and features.

In terms of scalability, the system has to support large organizations. Each organization is essentially a client that pays for a license to use the service. Furthermore, the organizations may have subsidiaries and subcontractors with considerable amounts of employees in each one. Some companies may even employ freelancers who are not directly linked to them. Not only that but each user can also work under multiple companies with different roles and access rights.

1.3 Protacon Solutions Ltd

The project was assigned by Protacon Solutions Ltd, a subsidiary of Protacon Group. Protacon Solutions is a software development company based in Jyväskylä, Finland with multiple offices around the country. The company focuses on digitalization and software development including creating web and mobile applications and services as well IoT solutions. (Digitalisaatio n.d.) The project in question, *TyPe*, was one of Protacon Solutions' many original products.

The project was developed by a team of software developers employed by Protacon Solutions at their office in Jyväskylä. The team consisted of developers with various talents in frontend and backend development as well as user experience enhancement. The development started in early 2018 with new features still in progress at the end of the year.

1.4 Thesis Objectives

Naturally, the main goal of the thesis was to create a functioning system that meets the requirements set by Protacon Solutions and its clients. It was difficult at times due to the complex nature of the requirement specifications and lack of resources in general. Although it took plenty of effort, a team of software developers undertook the challenge to meet the specified requirements.

In addition, the thesis aims to shed light on some of the design choices made by the developers and to spread knowledge about the associated techniques and technologies. Although the thesis focuses on a particular project, other development options are also discussed to highlight their features and shortcomings. To share information to other developers, the thesis covers some of the most critical issues and solutions when it comes to developing a web service. The TyPe project also serves as a case study of design choices, which includes discussion of its flaws and ways to improve the software system.

The thesis also served as a method of self-learning. Learning about different development options is highly useful since there is not enough time to study them thoroughly in normal day-to-day work. Not only that but there are also countless critical aspects of server-side programming that are often ignored or hastily presented in school curricula, resulting in low proficiency and experience when using them in a workplace environment. For those reasons, the thesis was used to improve the author's practical skills and knowledge in software development.

2 Web Services

2.1 What Are Web Services?

The blanket term *web service* can have different meanings for different people oftentimes making it confusing. Essentially, a web service is a means of communication for machines through a network, usually the World Wide Web. (What Are Web Services and Where Are They Used? 2013) Due to the vague nature of the definition, it covers

most server-side software for online stores, social media, management systems, online video games and many mobile applications.

From a software developer's standpoint, web services are associated with server-side programming, also called *backend* programming. It means developing software that ordinarily communicates with a *frontend* application such as a website. Both parties communicate back and forth by sending requests and responses to each other (What Are Web Services and Where Are They Used? 2013).The messages can contain data about practically anything, whether it is submitted form data, a user's login information or metadata. Web services often utilize databases to save the received data or load data to send back to the client.

Web services are accessed via transport protocols, such as HTTP and SMTP (Mueller 2013). One of the most common ways of accessing a web service for normal users is through an application in a web browser. When a user interacts with a web application, it sends requests to a dedicated web service in the form of XML messages using one of the transfer protocols. Web services decode the data and perform any necessary actions before returning a response in the same format. (What Are Web Services and Where Are They Used? 2013.) Certain modern web services, for instance RESTful web services, send JSON-based messages instead of XML. A simplified version of the client-server communication process is illustrated in Figure 1.

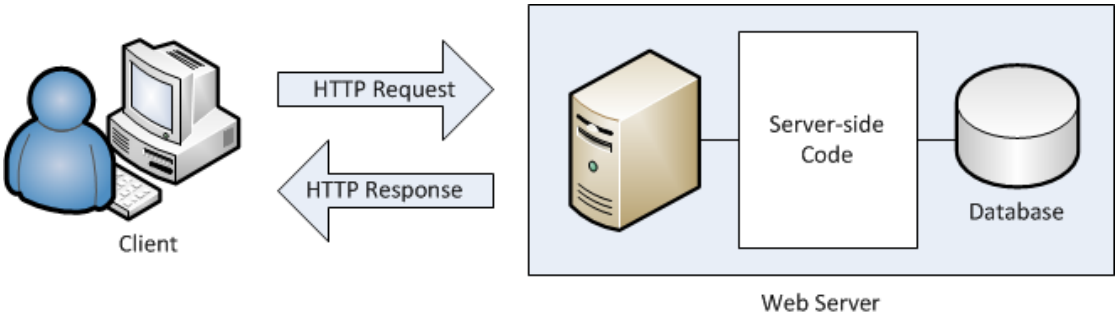


Figure 1. Communication between a client and a web server.

Web services are occasionally mixed up with web APIs. Certainly, they are both related to machine-to-machine communication, although in slightly different ways. De-

spite the fact that all web services are APIs, not all APIs are services. They differ fundamentally in terms of their approach to communication. In general, web services are somewhat tightly coupled with their web applications, whereas API requests require less knowledge of the server architecture to deliver a request. In the case of RESTful APIs, they use HTTP methods such as GET and POST to redirect the request to the appropriate destination. (Verma 2018.) In a sense, RESTful web services also have multiple API endpoints that are used to access certain resources in the service.

2.2 Design Models

2.2.1 General Info

Since web services have to communicate with the frontend application and occasionally other APIs, it is important to choose the right *design model*. Most web APIs and services use one of the two most dominant techniques, SOAP and REST, which account for around 90% of all APIs (REST in peace, SOAP 2010). The distribution of usage in 2010 can be seen in Figure 2 (Hoppe 2015, 9).

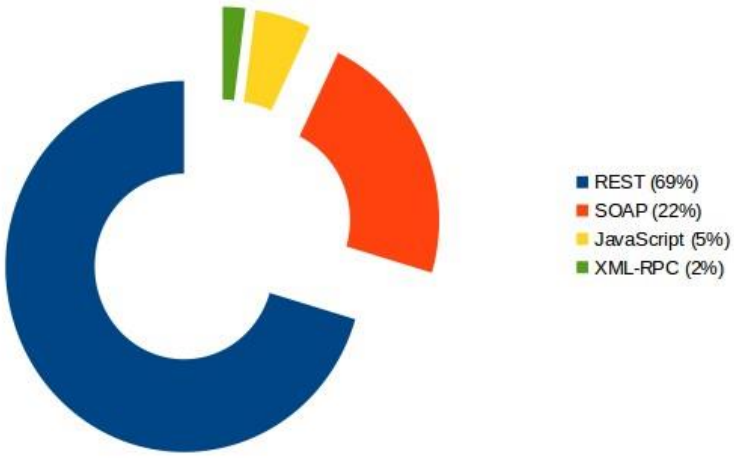


Figure 2. Usage statistics of web API models.

Technically, SOAP is a protocol whereas REST is an architectural style. (What is REST? n.d.) Despite them being fundamentally unalike, developers usually have to choose between the two or one of the less common technologies. The model will greatly in-

fluence the design of a web service, as everything revolves around the communication architecture set up by the selected technique. However, programming languages and frameworks often use a predetermined model, which means that developers cannot alter it, even if they wanted to. The differences between the technologies lie mostly in communication methods, bandwidth, statelessness, caching and ease of use, which are covered in the following chapters (SOAP Vs. REST: Difference between Web API Services n.d).

2.2.2 SOAP

SOAP, or Simple Object Access Protocol, was designed to allow machines to communicate with each other over the Internet, even if they use software with distinct programming languages. It also supports multiple communication protocols such as HTTP, SMTP and FTP (Kumar n.d). Developed by Microsoft in 1998, SOAP quickly became the *de facto* standard web service technology. (Box 2001.) Despite no longer being the predominant option, SOAP is still utilized in certain software systems.

The most apparent characteristic of SOAP is its heavy use of XML. The protocol utilizes *WSDL*, or Web Service Description Language, to define information about the service in XML format, including how it can be accessed, what sort of data to provide it with and a set of endpoints. (Shah n.d.) The WSDL document is a sort of contract between the client and the service. This results in SOAP being tightly coupled with the server, reducing its flexibility (Wodehouse n.d). An example of a WSDL file can be seen in Figure 3 (Tutorial – Example of a SOAP message 2012). Although SOAP uses WSDL, it is not unique to the protocol.

```
<?xml version="1.0"?>

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
  <m:Trans xmlns:m="http://www.w3schools.com/transaction/"
  soap:mustUnderstand="1" soap:actor="http://www.w3schools.com/appml/">234
  </m:Trans>
</soap:Header>

<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>

  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

Figure 3. An example of a SOAP WSDL document.

SOAP messages include an XML document with the following parts: Envelope, Header, Body and Fault. First of all, the Envelope wraps the entire message together with a start and end tag. Body is the main part of the message, which contains any data returned from the service or what kind of data is being requested, while the optional Header tag can include any additional parameters. Lastly, the Fault element includes information about handling errors that may occur during the communication process. (The structure of a SOAP message n.d.) In the case of errors, it allows retrying of certain operations or redirection to another part of the service. REST, on the other hand, has to receive a manually sent request in a similar case.

SOAP offers high security making it a good choice when dealing with delicate information. It utilizes WS-Security, an extension to SOAP, which improves integrity and confidentiality of the message. The extension allows various encryption types to be used for safe transportation of security tokens. WSS is not a perfect technology on its own; however, SOAP also supports Secure Sockets Layer, SSL, for establishing a secure connection between the client and server. On top of that, SOAP has a multitude of other extensions available for it. (Wodehouse n.d.)

Although SOAP offers great security and transactional reliability, it suffers in performance. By default, XML tends to be somewhat slower than JSON format, notably used by REST, due to the envelope's size. (SOAP Vs. REST: Difference between Web API Services n.d.) Not only that but SOAP-based calls can also not be cached on the client-side for later re-use and as a result cost more time (Wodehouse n.d). Since most features can be implemented using any technology, developers tend to prefer the quicker ones.

2.2.3 REST

Compared to SOAP, Representational State Transfer is a more recent technology, having been created in the early 2000s (The History of REST APIs 2016). It is an architectural style that aims to fix some of the issues with the more traditional SOAP. Instead of inventing new standards, REST implements existing HTTP functionalities, for instance HTTP verbs. (Kumar n.d.) RESTful service is a term generally used to describe a web service that implements the REST technology.

The goal of REST is to offer a more loosely coupled technology for web service communication. A tightly coupled web service can only be used for a certain application and even minor changes require drastic changes to the opposite side. The loose coupling of REST allows developers to save time by using the same or slightly modified API for multiple projects. (Gilmore 2018.) Whether the developers decide to use this to their advantage is up to them; nevertheless, it is one of the main advantages of REST.

RESTful service structure design is comprised of three parts: resources, verbs and representation (Kumar n.d). Resources are available through URIs. For instance, when requesting videos, the resource might be located in the following address: *http://www.my-web-service.net/video*. Furthermore, there could be specific sub-resources under the video resource, for example cat videos, in which case the noun cat would be added to the end of the URI. REST resources are accessed using the HTTP verbs, such as GET, POST, PUT, DELETE, which indicate the backend what operations to perform. Generally, GET is for requesting, POST for creating and other miscellaneous operations, PUT for updating and DELETE for removing data, although there can

be exceptions. Finally, REST supports the use of most formats including but not limited to JSON and XML. (Kumar n.d.) REST is stateless, meaning that each request includes enough information to handle it on the server-side without saving data about the session (The History of REST APIs 2016).

The primary reasons for using REST lie in its performance speed, maintainability, ease of use and flexibility. As previously mentioned, JSON is generally quicker than XML due to the nature of the format. On top of that, it is considered easier to learn and has great support for web browsers. HTTP methods and the resource architecture are also simple to use and understand. Due to REST's independency of representation format and programming languages, it can be used in most situations. (Kumar n.d.; Wodehouse n.d.)

The main disadvantages of REST are over- and under-fetching of data from a database and weak data typing. Under-fetching is due to the resource-based architecture, which results in extra requests that have to be sent back and forth in order to receive all the necessary data. Over-fetching, on the other hand, can become an issue when the application only needs a part of the data from a resource. (The ultimate guide to API architecture: REST, SOAP or GraphQL 2018.) Since REST does not inherently offer strong data typing, developers have to make sure the data matches the appropriate type.

2.2.4 Other Alternatives

GraphQL is Facebook's take on web service protocols. It is a query language that was first introduced to the masses in 2015 after having been used internally at Facebook since 2012. GraphQL enables the client application to precisely determine what kind of data it wants to receive or manipulate. This is achieved by a inserting a query inside the request, determining all the different properties or fields, as shown in Figure 4 (Anser 2017). Using this logic can reduce the amount of requests that the client has to perform to receive the required the data, while also optimizing the performance by avoiding over-fetching. That said, having so many useful features also comes with drawbacks. Due to the freedom of constructing requests on the client-side, the server-side code has to carefully handle security issues and set query limitations for performance reasons. (Wieruch 2018.)

```
{
  user(id: 4802170) {
    id
    name
    isViewerFriend
    profilePicture(size: 50)  {
      uri
      width
      height
    }
    friendConnection(first: 5) {
      totalCount
      friends {
        id
        name
      }
    }
  }
}
```

```
{
  "data": {
    "user": {
      "id": "4802170",
      "name": "Lee Byron",
      "isViewerFriend": true,
      "profilePicture": {
        "uri": "cdn://pic/4802170/50",
        "width": 50,
        "height": 50
      },
      "friendConnection": {
        "totalCount": 13,
        "friends": [
          {
            "id": "305249",
            "name": "Stephen Schwink"
          },
          {
            "id": "3108935",
            "name": "Nathaniel Roman"
```

Figure 4. GraphQL request and response.

Similarly to GraphQL, Falcor is a technology that describes data querying parameters. It is a server-side JavaScript library developed by Netflix to simplify data transportation. As its format, it uses JSON just like REST and GraphQL. However, instead of replacing a web service's backend, it acts as middleware between the client and the server. In most cases, Falcor is a more light-weight and easy to use alternative, although it does not offer as much utility as the other available technologies. (Helfer 2016.) It is also limited to applications using JavaScript language, such as Node.js services.

RPC, which stands for Remote Procedure Call, is another alternative to the aforementioned technologies. There are various implementations of RPC for different formats, such as XML-RPC and JSON-RPC. When the service consists of varying actions, RPC can be a good choice. This is because RPC is essentially a collection of methods that are called directly on the server by the client application via an HTTP request. (Sturgeon 2016.) Certain languages have their own language-specific technologies, such as the Java RMI, short for Java Remote Method Invocation (Pohjolainen 2016). It is an object-oriented implementation of RPC (Difference Between RPC and RMI 2017). RMI revolves around a stub class and skeleton class. Stub is basically the client's representation or interface of a remote object. It transmits or "marshalls" data to the server-side skeleton, which converts it into Java code and invokes the proper method on the

server. Despite their quite fast performance, Java RMI applications are tightly coupled as they require that both the client and server use Java. (Pohjolainen 2016.)

2.3 Development Frameworks

2.3.1 Symfony

Symfony2 is one of the most versatile backend development frameworks. The PHP-based framework is essentially a collection of smaller reusable libraries, known as bundles. Developers can select which bundles to use for each project making the framework extremely flexible. Although the libraries can also be used separately from the framework, Symfony allows for quicker installation and guaranteed compatibility. (Hansen 2017.) One of the most important bundles is Doctrine, which is used to access information in a database via an ORM. Doctrine makes using databases simple without having to learn SQL.

Symfony's long-standing position as one of the best frameworks is proof of its reliability. It is still regularly updated and has a large user base meaning that someone has likely encountered most issues when it comes to using the framework. Overall, Symfony is a mature framework with great documentation. (ibid.) Symfony is also equipped with useful debugging tools, including a profiler tool that shows detailed information about requests and responses.

Compared to its rivals, Symfony loses in performance and ease of use. (ibid.) Granted, some of the more recent PHP versions introduced performance improvements, but using an ORM can still result in many extra database queries, especially with certain versions of Symfony. Symfony's difficulty comes from having to learn Doctrine and other technologies that are often used in conjunction with it (ibid). Configurations can also take time to get used to.

2.3.2 Laravel

Similarly to Symfony, Laravel is a PHP framework of separate bundles. In fact, it was originally built from Symfony components (Sakhibgareev n.d). It has, however, become its own entity with many benefits over its competition, including simplicity, robust tools and its own templating engine. According to a survey on SitePoint (Skvorc

2015) Laravel is by far the most popular framework among developers in 2018, especially for personal projects. (Rytov n.d.)

Laravel uses the Eloquent ORM to access data, similarly to Symfony's Doctrine. Eloquent can programmatically structure SQL queries using a query builder. It is compatible with different SQL databases including PostgreSQL and MySQL. More often than not, the backend and frontend environments are separated. However, Laravel's templating engine, Blade, can be used to integrate data from the backend straight into the application template. Blade is a straightforward, yet efficient feature for developers who prefer programming in that fashion. In addition, Laravel's Queue function allows performing actions asynchronously from the standard code. (ibid.) It is especially useful for sending emails or executing other time-consuming operations.

When it comes to performance, Laravel is about on par with Symfony, depending on the version. In the end, a framework's speed is the sum of its parts' speed. Overall, when weighing the pros and cons of each technology as well as reading developers' opinions, it is hard to argue against Laravel. In fact, it seems to be the go-to framework in in 2018.

2.3.3 Node.js Frameworks

Node.js is a runtime environment that can execute JavaScript code independently from a browser. It uses Chrome's V8 JavaScript engine to convert the code into low-level machine code, which performs far faster. (Patel 2018.) It can therefore be used to create web services by developers who prefer using JavaScript for building web or mobile applications. Similarly to frontend web applications, Node.js uses npm, Node Package Manager, and node modules to easily manage project dependencies (ibid).

Since most web applications use JavaScript or one of its frameworks, it can be beneficial to use the same language on the backend as well. JavaScript uses JSON as its native object notation format, meaning that the client, server and database can all use the same format. In addition, Node.js has an event-driven and non-blocking I/O. Whereas most servers run on multiple threads by spawning another for each request, Node.js runs on a single thread while creating additional threads for running callback functions determined in the server code. This feature makes Node.js a good

choice for applications that must support a high number of requests while having light data processing on the backend. On the other hand, Node.js suffers when there is a great amount of data to process in a single request. Therefore, it shines when it comes to real-time online applications. (Why the Hell Would You Use Node.js 2017.) A typical Node.js application can be a chat room or even a real-time browser-based game. The technology is not something one would select for every scenario but more of a situational option.

There are many Node.js frameworks available for use, with Express.js being the most common. It is a minimalist routing framework that creates an HTTP server on top of Node.js. Express is considered easy to use, scalable, and flexible for various purposes. Additionally, there is a myriad of ready-made solutions available due to the large user base. There are also full MVC frameworks, such as Koa2 and Sails.js, which are slightly less popular. Although they are based on Node.js, each framework is completely distinct from each other. (10 best Node.js frameworks in 2018 2018.)

2.3.4 Other Alternatives

Selecting a framework often comes down to the developers' preference in to programming languages. Developers who are familiar with Ruby might find Ruby on Rails to be a great choice. Similarly, Django being the most versatile Python framework is an obvious choice for Python developers. Even PHP has more frameworks to offer including Yii and CodeIgniter, which are both considerable options.

2.4 Database Types

2.4.1 Relational Databases

Relational Database Management Systems, or RDBMSs, are the most commonly used databases. Some of the most popular relational databases are Oracle, MySQL, Microsoft SQL Server, PostgreSQL and DB2. Usually, relational databases make use of the Structured Query Language, SQL. (Hammink 2018.) According to DB-Engines, (DB-Engines Ranking – Trend Popularity n.d) approximately 25% of all databases are relational databases. RDBMSs have been around for a long time, and, therefore, most developers more or less understand how they function.

Relational databases consist of tables. Each table has named columns, which indicate the data types for each cell in the table. Rows can be inserted to, updated or removed from the table. Related data in another table can be connected by using a Foreign Key or an index. Rows in each table typically have an ID property, also known as a Primary Key, which can be inserted into another table as a Foreign Key. This allows logically linking relevant data together by using SQL Join statements. On top of that, optional cascade rules can be set to make it so that the unused data in a relational table is automatically removed when a Primary Key is deleted.

Relational database structure and data types are defined in a schema, which updates whenever new tables are added or existing ones are altered (see Figure 5) (What is a Database Schema? 2016). The data types are enforced by constraints. (Homan 2014.) Some of the most common constraints alongside primary and Foreign Keys are NOT NULL, meaning that the value has to be defined, and UNIQUE, meaning that another row cannot have the same value in the column.

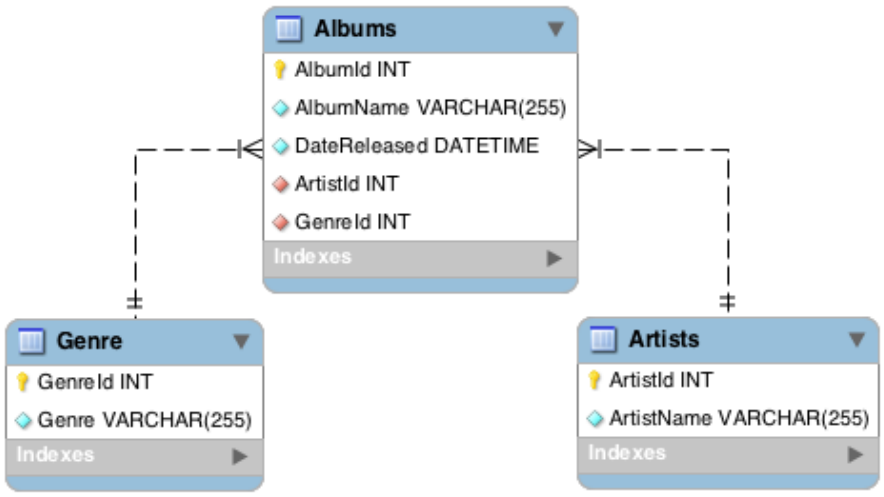


Figure 5. Simple relational database schema. In this case, artists can have multiple albums that belong to a single genre out of many.

One of the problems with complicated relational databases is that the data in a single table will often not match an object in an application (Hammink 2018). Oftentimes developers use an ORM to map database table data into objects, which is more of a workaround and results in slower performance. It can force developers to learn both

the object mapping and the database structure since it is not always viable to use the ORM. These workarounds are essentially why NoSQL databases were invented (Harris 2016). Generally, relational databases are best used with a clear database design in mind when first implementing it. Creating new relations later on into the development, for example, causes performance issues due to the amount of Joins required.

2.4.2 Non-Relational Databases

Non-relational databases, also known as NoSQL databases, greatly differ from relational ones. Generally, NoSQL databases specialize in a handful of use-cases, as they were designed to overcome specific technical difficulties with traditional RDBMSs. Although situational, NoSQL databases come in various sorts, including Key-Value Stores, Wide Column Stores, Document Stores and Graph Stores. (Hammink 2018.) Even though RDBMSs are more commonly used, many tech giants including Facebook, Google and Amazon have widely adopted the use of NoSQL databases. (Reeve 2012).

Key-Value Stores are self-explanatory. They store elementary data, such as strings, numbers and dates, as values, which are mapped by a generated key. Key-Value Stores offer lightning fast query speeds with extreme simplicity as a tradeoff. Document Stores are a sort of key-value store as well. The difference is that the value is stored as an object in JSON format, giving it much more complexity and flexibility. (Hammink 2018.) JSON makes querying more intuitive from an object-oriented standpoint by allowing direct access to an object's properties.

Wide Column Stores are similar to relational databases in that they consist of tables with rows and columns. The main difference, however, is that Wide Column Store data is not constrained by data types, and, therefore, different rows may include different types of data. It may even store empty values instead of a NULL value, saving some space on the machine. Additionally, Wide Column Stores allow partially updating a single column value instead of the whole row. Each cell value also has a timestamp attached to it that indicates the time of insertion. (What is a Column Store Database? 2016.)

Graph Databases are collections of interconnected nodes, where each node is an object with properties. The nodes are connected unidirectionally to other nodes in a relational way. Other than the start and end nodes, all of the relations must define a type, e.g., a company node might have a “HAS_EMPLOYEE” relation to user nodes. (Carpenter 2018.) Graph databases are easy to visualize and understand, as illustrated in Figure 6 (Sasaki 2018). That said, in practice, there would be too much data to visualize, especially in huge social media applications.

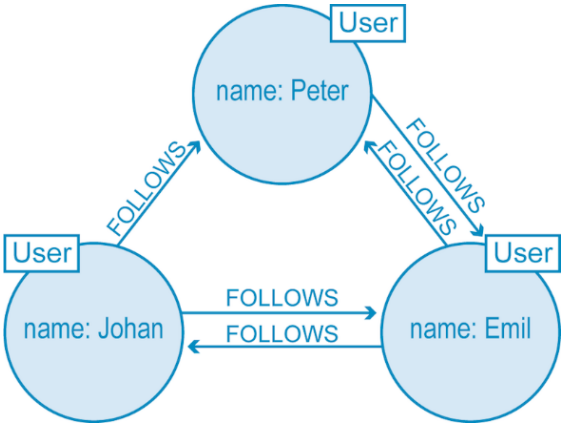


Figure 6. Graph database visualized.

The information in NoSQL databases is less strictly constrained than in relational databases. They do not natively support constraints on data types, although data integrity must still be manually enforced in most cases. Rather than using SQL, NoSQL databases each have their own query language, whether it is derived from SQL or completely original. (Yegulalp 2017.) Hence, it can be a hurdle to jump from using an RDBMS to a NoSQL database.

3 Development Choices and Project Setup

3.1 Selected Technologies

3.1.1 Symfony and MariaDB

The framework of choice for the project was Symfony. Since it is commonly used at Protacon Solutions, there is a good understanding of it among the developers. Someone at the company has likely encountered most issues already and help is never far. Not only that but there is plenty of reusable code made by the employees, which can greatly reduce development time. At any rate, Symfony is a well-rounded framework for any project in general.

Symfony makes use of relational SQL databases. MariaDB, a fork of MySQL was chosen for this project. It works mostly the same way with some tiny differences. MariaDB supports some cutting-edge features including thread pool, NoSQL queries with Cassandra and dynamic columns. It also claims to have quicker performance, especially with large datasets. (Nayyar 2018.) Although most of these features are insignificant for small-scale software, it is still a great modern database option.

3.1.2 Angular

The frontend web application was developed using the Angular framework, not to be confused with AngularJS. Similarly to Symfony, it is widely used at Protacon Solutions. Angular is a JavaScript framework that is based on TypeScript, a strict syntactical superset of JavaScript. The object-oriented TypeScript enables type annotations for functions and class properties as well as the use of object interfaces. It compiles into standard JavaScript code. (Lease 2018.) The Material Angular library was used for creating a clean and intuitive layout.

Angular notably utilizes Observables, which enhance event handling in the application. Observables allow a component to subscribe to an event, including to changes to a variable or application state property. Along with Observables, the *ngrx* library was used in the project in order to manage the application state. It is a Redux-inspired state management framework that acts as middleware for sending requests

from the client-side to the backend web service. Afterwards, the response data is stored in the state for later use.

3.2 Setting up the Project

3.2.1 Frontend Environment

Setting up the frontend application locally was quite simple. After cloning the codebase repository, the first step was to install the required packages using Yarn which is a dependency manager. It fetches the dependencies defined in the project's *package.json* file. Other required software include Node.js for hosting the application and Git for version control. Additionally, Webpack, or a similar module bundler, is required to deploy the application locally. The application uses information defined in an *.env* file. When starting the development, one has to make sure the *.env* file's configuration, such as the web service address is correct, to forward the requests appropriately.

3.2.2 Backend Environment

The backend code has to be hosted on a server that supports PHP. One way to easily deploy the application is to use Symfony's WebServerBundle. It uses PHP's built-in server and requires minimal configuration. Another way is to use a virtual machine, such as Vagrant, to separate the backend environment from the local machine, or to actually host it on another machine. In any case, the server must have PHP and MySQL available. Similarly to the frontend, environment files have to be properly setup to allow access from the local machine.

Symfony loads configuration files from the *config* folder. First of all, the *routes.yaml* file defines the location of the controllers, e.g., "src/Controller", and routes for other bundles. *Services.yaml* contains the configuration for service classes such as repositories for entity classes. In addition, the *config/packages* folder includes other configuration files for security, event subscribers, Doctrine ORM, migrations and any installed bundles. Lastly, there are *dev*, *prod* and *test* folders, which can include config-

urations specific to any of the three supported environments. (How to Organize Configuration Files.) For instance, having the Symfony profiler enabled for debugging is important in development environment.

Symfony bundles are managed by *Composer*. The first step to deploying the server is to run Composer's install command. Further bundles can be added to the project by using the require command and most of them have satisfactory documentation for setting them up. After installing the dependencies, the console script in the bin folder can be run with PHP. It includes useful tools for managing the database and other various features. For instance, the database can be set up with Doctrine's console commands, *doctrine:database:create* and *doctrine:migrations:migrate*. Doctrine itself is installed by requiring *symfony/orm-pack*. Migrate command builds the database schema from the project's migration files, which are generated using *doctrine:migrations:diff*. (Databases and the Doctrine ORM.) Moreover, MySQL can be used to directly access the database with the right credentials after configuring it in the *.env* file. If a graphical user interface is more desirable, HeidiSQL is a great alternative.

PhpStorm is a good text-editor option for the project. It can be utilized to automatically upload file changes to the remote server after configuring it properly. PhpStorm is naturally often used with PHP and Symfony, but other available options include Atom and Visual Studio Code. Postman is another optional program for testing purposes. It is used to more efficiently send requests to the server, instead of sending them via the application in the browser. Cmder is a useful console emulator tool for managing all the different environments, especially when using Git via the command line.

4 Database Management

4.1 Doctrine Primer

Entities are classes that are used by Symfony framework's Doctrine tool, specifically Doctrine 2. Similarly to classes in object-oriented programming, entities have properties and methods. The properties can include not only primitive data, such as strings

and numbers, but also other entities and collections of entities. Using private or protected properties with getters and setters to handle them in the code is highly recommended. Doctrine automatically maps the entity classes into database tables and objects into rows via an ORM (see Figure 7). (Databases and the Doctrine ORM.)

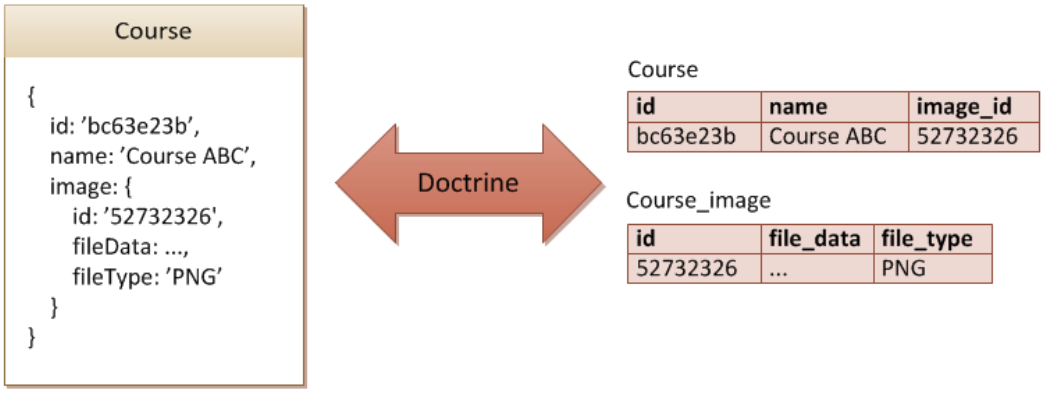


Figure 7. Visualization of a programming object and the corresponding database table schema.

The configuration file for Doctrine is located in *config/packages/doctrine.yaml* and *doctrine_migrations.yaml*. They must describe the URL and name of the database as well as the folder where entity classes reside in. Optionally, the naming strategy setting for entities and database types for objects such as DateTime can be included. The auto_mapping property should be set to true in order to automatically generate the Doctrine migration files. (How to Organize Configuration Files n.d.)

Doctrine's EntityManager class is responsible for handling the lifecycle events regarding the transformation of entities between the database and non-persistence and finding entities from the database. After modifying the entity properties in the code, calling EntityManager's *persist* method saves the changes to the object, although it does not trigger a database query yet. Calling the *flush* method afterwards runs all the necessary queries changing the database to match the created, modified and deleted entities that were persisted. EntityManager allows for native SQL queries as

well, if necessary. (Databases and the Doctrine ORM.) It is important to note that accessing a property results in a database query. Therefore, looping a property getter exponentially increases the amount of queries executed.

Everything regarding database can be configured with the ORM. Annotations can be used to set the table name, unique constraints, indexes, column names and types, nullable values and cascade rules (see Figure 8 and Figure 9). Some of the most common property types are string, integer and boolean as well as Collection for relational properties. Doctrine transforms these types into the database schema with the appropriate mapping, e.g., string becomes VARCHAR and Boolean becomes TINYINT. Furthermore, minimum and maximum lengths as well as nullable values determine validations for the property data. Finally, orphan removal can be disabled to keep the orphaned child entities in the database after removing the parent.

```
/**
 * Class Course
 *
 * @ORM\Table(
 *      name="course",
 *  )
 * @ORM\Entity()
 *
 * @Gedmo\SoftDeleteable(
 *      fieldName="deletedAt",
 *      timeAware=false,
 *      hardDelete=false
 *  )
```

Figure 8. Class annotation for the course entity.

```
/**
 * @var string|null
 *
 * @Groups({
 *      "ROLE_USER__Course",
 *      "ROLE_USER__Course.description",
 *      "ROLE_USER__CoursePackage.courses",
 *  })
 *
 * @ORM\Column(
 *      name="description",
 *      type="text",
 *      nullable=true,
 *  )
 */
private $description;
```

Figure 9. Description property annotation.

4.2 Schema Design

4.2.1 Entities

In the web application, the five main entities are separated into different views, where they can be accessed, created and modified (see Appendix 1 for various screenshots of the application). The entities are course, exam, course package, company and user. These entities were roughly designed at the beginning of the project as a baseline for what is to come. The main reason why the database schema was not fully realized at first was because the application required feedback and development ideas from the users. New features were added in whenever necessary. This, however, resulted in some problems later on.

Course

Course is the most essential entity in the application. It is a sort of quiz with a plethora of configuration options as well as questions. Both courses and their questions can include materials such as external links, files and images. Published courses can be assigned to users through course packages, after which they can practice it or take the exam. Each course can only be successfully finished once; however, failed ones do not count.

Exam

Exam entity is the result of a finished course, whether it is successfully passed or not.

It includes the answers that the user selected. On top of that, the exam includes all relevant data about the course at the time of completion in what is called an *exam snapshot*.

Course Package

As the name suggests, a course package is a collection of courses. They exist as a way to more efficiently share courses to relevant users, while categorizing them cleanly. The package itself only contains a name, description and image, although it features many relations such as included courses and owner companies.

Company

Companies are essentially counterparts to real-life companies in the application. Their properties include a name, address, image and Business ID. The Business ID is used in certain places to identify existing companies instead of using their Primary Keys. Hence, it is also used as a unique constraint in the database. Companies are the primary means of sharing and propagating access rights in the application. There are two types of companies, the ones with paying customers and their subsidiaries.

User

Users are associated with accounts that can access the application. They are tied to a single person by their full name and date of birth. Although there can be rare cases of multiple people having the exact same name and date of birth, this is robust enough for most cases. Users belong to at least one company when they are created, although they can be attached to others afterwards.

4.2.2 Organizations

One of the complications from a business perspective while designing the software was differentiating between the paying customers' companies and their subsidiaries. Although both are considered companies, the customers must have access to additional features and more rights in general. Some of those features include creating new subsidiary companies, courses, users and course packages as well as tracking users' exam results. In order to discern the two types of companies, the *organization* entity was designed.

When it comes to entity design, organization is fairly straightforward. It simply includes information about the real-life company so that Protacon Solutions can easily tell them apart and properly charge them for the service. Organizations can also have access to different paid features, which are distributed to users via the company entity related to that organization. These extra features, which include sharing course packages, more configuration options for courses and an SMS service, are retrieved when a user logs in to the application (see Figure 10). If the user belongs to an organization, the paid features are found directly through the organization entity relation. If they are a work manager, on the other hand, the PackageAdminRights entity itself, containing metadata about the work manager attachment, is retrieved. The paid features are then recovered by using the entity's ownedBy property, which indicates the organization that originally attached the work manager to the course package. In the application, users do not have access rights to organization entities, and, therefore the access control filter (see chapter 6.3.2) is temporarily disabled inside this function.

```
$filters->disable('accesscontrol');

$organization = $activeCompany->getOrganization();

if ($organization instanceof Organization) {
   $features = $organization->getPaidFeatures();
} else {
    $rights = $adminRightsResource->findOneBy([
        'admin' => $user->getId(),
        'company' => $activeCompany->getId()
    ]);

    if ($rights !== null) {
        $ownerOrg = $rights->getOwnedBy();

        $features = $ownerOrg ? $ownerOrg->getPaidFeatures() : [];
    }
}

$filters->enable('accesscontrol');
```

Figure 10. Paid feature retrieval from the database. The logic is handled differently depending on if the user belongs to an organization or a normal company.

Most importantly, the entity acts as a Boolean value for distinguishing organizations from other companies. By accessing the property in the server code, different use-cases can be handled neatly, as illustrated in Figure 11. This adds an extra layer of security and stability to the application, since it disallows users with high-level privileges from manipulating unintended properties on normal companies. It is also useful for accurately sharing access rights.

```
$activeCompany = $user->getActiveCompany();

if ($activeCompany === null || !$activeCompany->isOrganization()) {
    throw new HttpException(Response::HTTP_NOT_ACCEPTABLE);
}

...

/**
 * @return bool
 */
public function isOrganization(): bool
{
    return $this->organization !== null;
}
```

Figure 11. Organization property check for companies. The entity method returns a boolean depending on if the relationship exists.

4.2.3 Entity Relations

Relational database mappings are either one-to-one, one-to-many or many-to-many. One-to-one is simple in principle, as it means that only a single entity complements another one. In the application, one-to-one was only used for profile pictures and other images. Oftentimes, one-to-one relations can simply be added to the entity class itself as properties or implemented via a trait, although separation of concerns would suffer from that. On top of that, one-to-one relations are useful, when a single class, such as a file or a link is used by multiple entities. An important thing to note is that Doctrine will always serialize one-to-one relational entities, when retrieving an entity from the database, greatly slowing down the query in some cases. To counter-

act that, the relation can be defined on only one side, so that it does not get serialized from the other side of the relation. When it comes to the database schema, the entity data is separated into two tables with a Foreign Key in one of them. The target entity and Foreign Key Join column must be defined in the annotations of the property as shown in Figure 12. The definition is only required on one side of the relation. (Working with Relations n.d.)

```
* ...
* @ORM\OneToOne(
*     targetEntity="App\Entity\File",
*  )
*
* @ORM\JoinColumn(
*     name="profile_picture_id",
*     referencedColumnName="id",
*     nullable=true
* )
* ...
```

Figure 12. Annotation for a one-to-one relation.

One-to-many and many-to-one relations are counter-parts of each other. They were heavily utilized in the application; almost every entity was connected to each other in some way. The distinction from one-to-one is that both sides of the relation must define the property, although only one of them should have the Join column definition. (Working with Relations n.d.) Generally, the one-to-many side is considered the owning side of the relation whereas many-to-one is the child or *inverse* side. Therefore, the inverse side generally contains the Foreign Key column in the database table. The Doctrine ORM requires that both sides define the opposite property with *inversedBy* and *mappedBy* annotation definitions (ibid). Some one-to-many relations in the application include courses with exams and users with multiple sets of contact information.

Many-to-many is useful when ownership of an entity is shared between entities. For instance, companies have multiple users and users can be part of multiple companies, and the same principle applies to courses and course packages. In database terms, this results in what is called an associative table or *Join table*, which consists

of Foreign Keys of both sides of the relation. The owning and inverse sides can be chosen arbitrarily, although it makes sense to have some sort of hierarchy in mind. Case in point, the company entity should be the owner and user the inverse side. Although there are also many common naming conventions for Join tables, this project used the following style: “company_has_user”. Additionally, both many-to-many and many-to-one relation setters in the entity code must be called from the owner side, in order to properly persist the changes into the database (Working with Relations n.d).

4.3 Working with the Data

4.3.1 Serialization Groups

Serialization groups are used to determine which entity properties to include in the backend responses. This is done by Symfony’s *Serializer* component. (The Serializer Component n.d.) Configuring serialization groups is important for a few reasons. First of all, serializing only necessary properties improves the tidiness and maintainability of the code. Keeping things simple is almost always better. In addition, when sending objects in JSON format, they have to transformed, which takes time. There are also security issues with serialization. For instance, when requesting data for an exam, the correct answers should naturally not be included in the response. Even if they are not visible in the application itself, they can be viewed through the network tab of a modern web browser. In order to determine the serialization context for a request, one can use the Serializer component’s normalize method and pass it an array of serialization group names as a parameter (ibid).

As a rule of thumb, only non-relational properties should be serialized in the default group, i.e., “EntityName”. Relations, on the other hand, should have specific groups such as “Company.users” or “CoursePackage.courses” (see Figure 13). This way each request can contain different relational data. Another way to optimize the groups is by using custom names, such as “set.UserList”, which serializes necessary data for a specific view only. The only problem with that is that the group annotation must be added to each property that is included in it. Figure 14 illustrates how the serialized properties are utilized in the frontend application’s user list. The serialized group for

the request includes the users' generic properties, such as their full name, as well as their contact information located in the UserDetail entities.

```
* ...
* @Groups({
*      "ROLE_USER__Course",
*      "ROLE_USER__Course.description",
*      "ROLE_USER__CoursePackage.courses",
* })
* ...
```

Figure 13. Serialization groups annotation for the course description property.

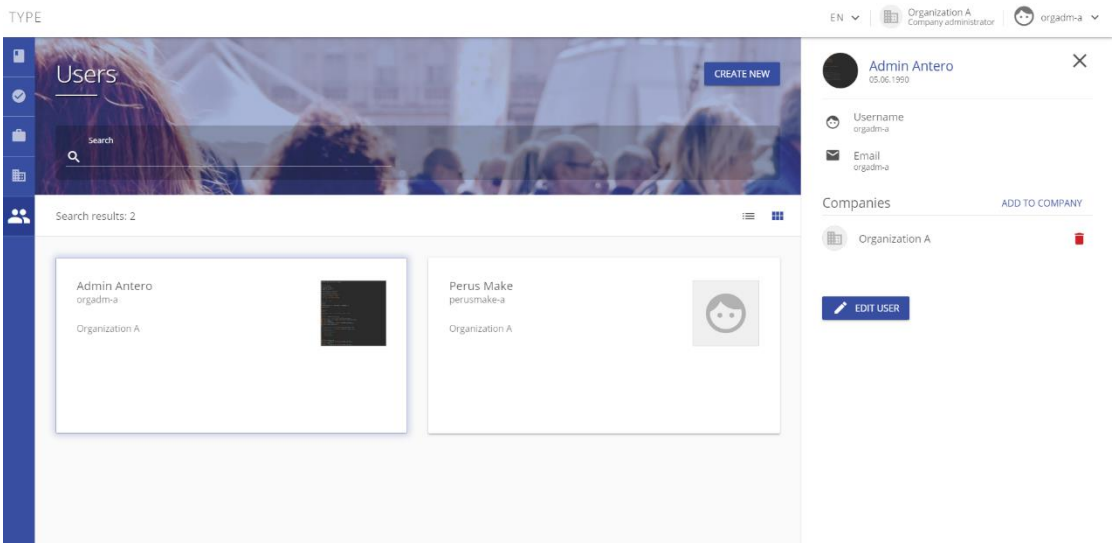


Figure 14. User list view.

4.3.2 Symfony Forms

Symfony forms are useful for validating the data received by REST endpoints. As demonstrated in Symfony's documentation (Forms n.d.), the form can be instantiated inside the controller for each entity. However, the forms should be separated into different files for better maintainability, which is exactly what was done in this project. Inside the controllers, each form type, such as create type, update type and patch type, was mapped to their specific REST endpoint (see Figure 15). The primary reason for this is to allow modifying certain properties on create or update only. As a rule of thumb, PATCH is for partially modifying an entity, PUT is for updating it entirely and POST is for creating and other miscellaneous processes. For instance, a

course can only be added to a course package after its creation by using the PATCH form type. There are many ways to implement Symfony forms, but this method is great when utilizing generic RESTful traits.

```
class CourseController extends Controller
{
    protected static $formTypes = [
        self::METHOD_PATCH => CoursePatchType::class,
        self::METHOD_CREATE => CourseCreateType::class,
        self::METHOD_UPDATE => CourseUpdateType::class,
    ];
```

Figure 15. Form type mapping.

The form types must specify which entity properties can be modified along with validation options. As shown in Figure 16, the form takes in a list of property names, types as well as optional validation options and default values. Symfony comes equipped with most of the commonly used types, but custom ones can be added as well. (Forms n.d.) Checking the documentation is essential, as many of the available options have default values that correspond with certain property types and some properties are completely ignored by Symfony in specific cases. Additionally, Symfony forms can be rendered on the frontend via HTML or Twig template files (ibid). This reduces development time, since the validation options do not have to be defined on both ends.

Forms can make use of transformers to modify the form data or match it with other data. Some useful cases for transformers include transforming a date string to a Date or DateTime object or finding an entity based on its ID, as illustrated in Figure 16 with the File class. The IdPropertyTransformer instance requires access to Doctrine's EntityManager as a parameter in order to perform the query. More specifically, transformers have a function, called *transform*, for converting the normalized data into either view data or model data depending on the use-case. Occasionally, the reverseTransform function can be used to do the opposite conversion. (ibid.)

```php
public function buildForm(FormBuilderInterface $builder)
{
    $builder
        ->add(
            'name',
            Type\TextType::class,
            [
                'label'          => 'Name',
                'min'            => '2',
                'max'            => '255',
                'required'       => true,
            ]
        )
        ->add(
            'description',
            Type\TextType::class,
            [
                'label'          => 'Description',
                'required'       => false,
                'empty_data'     => '',
            ]
        )
        ->add(
            'mainImage',
            Type\TextType::class,
            [
                'label'          => 'Image',
                'required'       => false,
            ]
        );

    $builder->get('mainImage')->addModelTransformer(
        new IdPropertyTransformer(
            $this->objectManager, File::class
        )
    );
}
```

Figure 16. Course package entity form. The properties and options are added to the FormBuilder object and the “mainImage” string value is transformed to an Entity.

4.3.3 Lifecycle Events

Doctrine’s entities have lifecycle events that occur in certain situations. Some of these events take place after and before creating, updating, deleting or persisting an entity instance. Event listeners for each of the lifecycle events are defined in the *subscribers.yaml* file, as shown in Figure 17, to hook them into the EventManager instance. An important thing to note is that events taking place before a database

query, such as an insert or update, can still result in the query failing. (Events n.d.) Therefore, calculations and other procedures should be handled in the lifecycle hooks that are executed *after* the database is changed whenever possible. It is essential to select the proper event for every use-case.

```
App\EventSubscriber\UserEntitySubscriber:
    tags:
        - { name: doctrine.event_listener, event: prePersist }
        - { name: doctrine.event_listener, event: preUpdate }
```

Figure 17. Event listener configuration in *subscribers.yaml*.

In this project, the lifecycle callbacks were utilized substantially to propagate access control rights, generally via PostPersist and PostUpdate callbacks. In general, they are also useful for sending messages via email and SMS as well as calculating numbers in the database. For instance, when a product is bought from an online store, the product's stock value could be subtracted by one in the order's PostPersist callback. One thing to note is that PostPersist is only launched after SQL *Insert* operations whereas PostUpdate is executed after Update operations (ibid). Doctrine's DQL queries, which are database operations written either in raw SQL or by using a QueryBuilder object, do not trigger the lifecycle callbacks.

5 Security

5.1 Authentication

5.1.1 JSON Web Tokens

JSON Web Tokens, or JWTs, are an alternative to session cookies, SWTs and SAML tokens. They are often utilized for logging in and authentication in web applications, as they are fast to transmit between two bodies and secure when accompanied by a checksum. In practice, the user starts by submitting their credentials, generally a username and password, to the web service. The server-side software checks that the information is correct before creating a signed token, which is sent back to the

client. Afterwards, the token is used for authentication by attaching it to the Authorization header, instead of checking the credentials on every request. (Yellavula 2018.)

A JWT token houses the following information: header, payload and signature. Generally, all of the data is encoded in Base64 and separated by dots. The header describes which algorithm is used to encode the token as well as the type, which is JWT in this case. The payload can include any customizable data such as the username in JSON format. The final part, signature, is a combination of the Base64-encoded data and the web server's secret key signed with the HMAC algorithm. (Introduction to JSON Web tokens n.d; Yellavula 2018.) See Figure 18 for an example of the encoded and decoded contents of a JSON Web Token (Nasseri 2016).

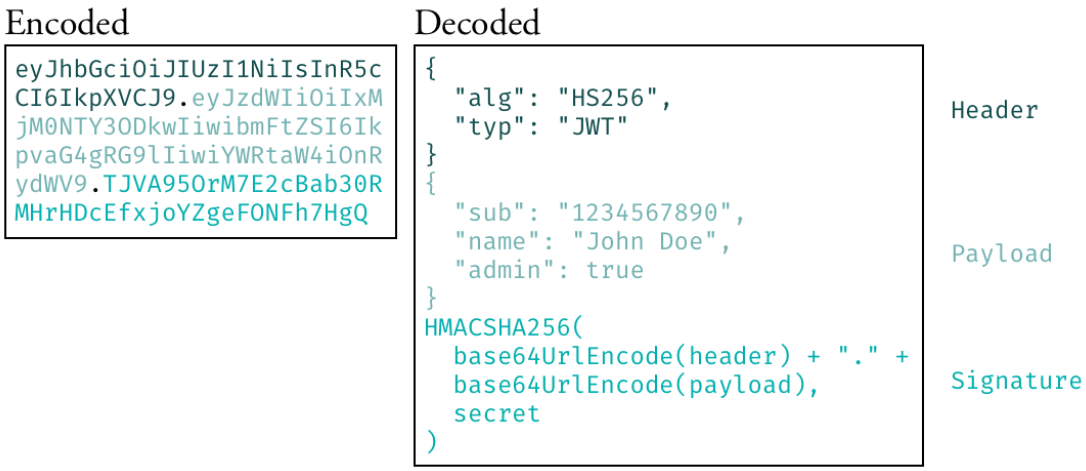


Figure 18. Encoded and decoded JWT information.

As an extra security measure, the authentication was split into access tokens and refresh tokens. The access tokens contain the aforementioned signed token and information about the user, including their active company and access role in said company. The token is attached to each backend request's header by the HTTP interceptor service on the frontend, in order to access protected resources. However, the access token has a short lifespan, which is where the refresh token comes into play. Refresh tokens are used to retrieve a new access token after it has expired and after first logging in. Although they have a much longer lifespan compared to access tokens, after it expires, the user must login to the software once more. The user's login

data is only authenticated when the refresh token is renewed, which makes the rest of the requests faster and less straining for the backend. That being said, the refresh token must be kept secure, since it can be used to renew access tokens until it expires or gets blacklisted.

Lexik JWT authentication bundle makes JWT authentication simple to implement. By modifying the configuration options in *lexik_jwt_authentication.yaml* file with the values shown in Figure 19, the token creation becomes automatized. Additionally, *Gesdinet JWT refresh token* bundle enables refresh tokens on top of Lexik's base implementation. For configuration, Gesdinet's bundle also requires the path definition in *routes.yaml* (see Figure 20). The time-to-live values can be adjusted, but it should be kept short for access tokens and somewhat longer for refresh tokens. Ten minutes and one day respectively is a good place to start. The secret and public keys are generally defined in a separate file and the pass phrase must always be kept secure along with the secret key.

```
lexik_jwt_authentication:
    private_key_path: '%kernel.project_dir%/config/jwt/private.pem
    public_key_path:  '%kernel.project_dir%/config/jwt/public.pem'
    pass_phrase:      '%env(JWT_PASSPHRASE)%'
    token_ttl:        '%env(JWT_TTL)%'

gesdinet_jwt_refresh_token:
    user_provider: App\Security\UserProvider
    ttl_update: true
    ttl: 604800 # 1 week
```

Figure 19. Configuration for Lexik JWT and Gesdinet JWT bundles. Environment variables are defined in a separate file.

```
gesdinet_jwt_refresh_token:
    path:     /auth/token/refresh
    defaults: { _controller: gesdinet.jwtrefreshtoken:refresh }
```

Figure 20. Gesdinet JWT route configuration.

The JWT creation event can be subscribed to, similarly to Doctrine entity lifecycle events. This makes it so that custom data can be added to the token between the token's creation and sending it back to the client. As shown in Figure 21, user data, such as their name, active company and access role as well as extra security measures are added to the payload. Naturally, if the user or someone else can change the information within the application, the token data must also be refreshed, despite the old token still being valid.

```
public function onJWTCreated(JWTCreatedEvent $event): void
{
    $payload = $event->getData();

    $this->setSecurityData($payload);

    $this->setUserData($payload, $event->getUser());

    $this->setCompanyData($payload, $event->getUser());

    $this->setApiKey($payload, $event->getUser());

    $event->setData($payload);
}
```

Figure 21. JWT creation event handler function.

5.1.2 Configuration

Symfony's security configuration is located in the *security.yaml* file (How to Organize Configuration Files n.d). The requirements for setting up authentication are to, first of all, install the *symfony/security-bundle* and then create a user class. Alongside that, a *user provider* is required to load user data from the request's session. The provider is used to load a user by its username, email or token information. If there are multiple varying user types, a certain user provider can be configured to only support specific ones. Each user provider must be defined under the providers section in the *security.yaml* file (see Figure 22). (Security n.d.) When using token authentication without actual users, some sort of entity implementing Symfony's UserInterface must still be created in order to use the security functionalities.

```
providers:
    chain_provider:
        chain:
            providers: [user_provider, api_key_user_provider]
    api_key_user_provider:
        id: App\Security\ApiKeyUserProvider
    user_entity_provider:
        id: App\Security\UserProvider
```

Figure 22. User provider configuration.

Other than the providers, the security configuration should contain encoders and, most importantly, *firewalls*. Encoders define the algorithm that is used to cryptographically hash the users' passwords, e.g., argon21 or bcrypt (How to Manually Encode a Password n.d). It is possible to use different algorithms for different user objects. Firewalls, on the other hand, describe security measures for all the routes that are used in the service. Each firewall, with a customizable name, must define a regular expression pattern that is used to match routes to the firewall, for instance "^/api/course$". Excluding the pattern definition matches the firewall to all routes instead. Most of the routes used in the service used the same authenticator, except the password recovery controller, file download controller (see chapter 5.2.1) and, naturally, the JWT-related endpoints. Additionally, firewalls can be limited to certain HTTP verbs such as GET, POST and PUT. When including an authentication token with each request, as is the case with this application, the stateless property should also be set to "true". (Security n.d.)

An optional *Guard* authenticator, which is a type of authentication provider, was created to handle the JWT authentication. The guard handles the actual authentication procedures such as checking that the token exists and is valid. In the configuration, the Guard requires a user provider and an authenticator class, one of which is described in more detail in chapter 5.2.1. To completely bypass security checks, it is possible to set the *security* property to "false" for a firewall, making it completely public. A fully configured firewall can be seen in Figure 23.

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    login:
        pattern: ^/auth/get_token$
        stateless: true
        anonymous: true
        json_login:
            provider: user_provider
            check_path: /auth/get_token
            success_handler: lexik_jwt_authentication.handler
                             .authentication_success
            failure_handler: lexik_jwt_authentication
                             .handler.authentication_failure
    refresh:
        pattern: ^/auth/token/refresh
        stateless: true
        anonymous: true
    api:
        pattern: ^/
        stateless: true
        anonymous: true
        guard:
            provider: user_provider
            authenticators:
                - lexik_jwt_authentication.jwt_token_authenticator
```

Figure 23. Firewall configuration.

5.1.3 Authentication Service and Interceptor

The authentication service is a singleton class on the frontend. It handles operations related to logging in and authentication by utilizing the Angular-jwt's JWTHelperService and ngx-webstorage's LocalStorageService. The user is authenticated and afterwards logged in to the application by sending a request to the access token URL on the backend and then decoding the access token into a UserProfile object with the JWTHelperService (see Figure 25). The UserProfile contains the user's name, active company, access role and API key. Subsequently, the tokens are stored in the local storage of the browser by utilizing the LocalStorageService.

```
public function authenticated(): Observable<boolean> {
  const isTokenExpired = this.jwtHelper.isTokenExpired(
    this.localStorage.retrieve('token')
  );

  if (isTokenExpired) {
    return new Observable<boolean>((observer) => {
      this.getAccessTokenWithRefreshToken().subscribe(
        (authenticated: boolean) => {
          observer.next(authenticated);
          observer.complete();
        }
      );
    });
  } else {
    return Observable.of(true);
  }
}
```

Figure 24. Access token retrieval from local storage. If the token has expired, a fresh one is retrieved by using the refresh token.

```
this.http.post(url, payload, httpOptions).subscribe(
  (token: TokenDataInterface) => {
    this.localStorage.store('token', token.token);
    this.localStorage.store('refreshToken', token.refresh_token);

    const decodeUser =
      this.jwtHelper.decodeToken(token.token)
      as UserProfileInterface;

    observer.next(decodeUser);
    observer.complete();

    this.loggedInUser$.next(decodeUser);
  },
  ...
```

Figure 25. Decoding the user data from a token after storing it in the local storage.

On the frontend, an HTTP interceptor is used to add the JWT into the header of each backend request. It makes an identical copy of all outgoing requests, modifies the header and sends the request to the original address. This is standard practice, because attaching the JWT from each component or service would be unnecessarily complex for little gain. Not only that, but it also allows for less tightly coupled design,

making it easier to refactor the logic later on. In addition, the interceptor service allows one to handle the different error cases accordingly. In the case of 401 error, which indicates *unauthorized access*, the access token is refreshed and the request sent again subsequently. The entire process is demonstrated in Figure 26.

```
private function intercept(req: HttpRequest, next: HttpHandler) {
  return next.handle(
    req.clone({
      headers: req.headers.set(
        'Authorization',
        this.injector.get(AuthService).getAuthorizationHeader())
    })
  );
  .catch(error => {
    if (error.status === 401) {
       return this.handle401Error(req, next);
    }
}

private handle401Error(req: HttpRequest<any>, next: HttpHandler) {
   return this.injector.get(AuthService)
     .getAccessTokenWithRefreshToken()
     .switchMap((isTokenRefreshed: boolean) => {
        if (isTokenRefreshed) {
          this.tokenSubject.next(isTokenRefreshed);
          return next.handle(this.addToken(req));
        }
      };
}
```

Figure 26. Cloning a request with an HTTP interceptor. The Authorization header is attached to the cloned request and a new token is retrieved once it has expired.

5.2 Firewalls and Guards

5.2.1 File Downloads

The issue with the application's HTTP interceptor was that it only works with requests sent from within the Angular instance itself. This means that it does not apply to HTML attributes that send requests, such as *src* and *href*, which are generally used for requesting image elements and links respectively. Optimally, file downloads would be handled through the href attribute, but unfortunately it bypasses the HTTP

interceptor. As a workaround, the token data could also be appended directly to the URL as GET parameters, although that is less secure since the token is in plain text and the URL becomes extremely long. Therefore, it would be less than ideal.

The most obvious solution to this problem was to manually recreate the request in the application code to make the interceptor handle it. An Angular directive, a reusable component which modifies HTML elements with behaviors and styles, was made to handle the request and either add the file into the HTML element or download it (see Figure 27). The returned file is in Blob format, meaning that it has some limitations in file size and speed. Although those limitations are insignificant in the grand scheme of things, the most glaring issue is the lack of platform support. Older browsers, such as Internet Explorer version 10 and older, do not trigger the file download with this method. On top of that, mobile devices with the iOS operating system are lacking a file system meaning that they handle file downloads differently from Android and Windows platforms. Normal href downloads still work, because the devices are designed to handle those events in a specific way, which does not apply here. Additionally, programmatically created files in Blob format do not support file downloads or opening in a new window or tab. Unfortunately, the application had to support various platforms and devices, and, therefore, this solution was insufficient.

```
@HostListener('click')
public getFile() {
  this.httpClient.get(url{ responseType: 'blob' })
    .subscribe((blob) => {
      const blobURL = URL.createObjectURL(blob);

      // Create temporary link
      const link = document.createElement('a');
      link.setAttribute('href', blobURL);
      link.setAttribute('target', '_blank');
      link.setAttribute(
        'download', this.appFileDownload.originalName
      );
      document.body.appendChild(link);

      // Activate download and remove link
      link.click();
      window.URL.revokeObjectURL(blobURL);
      link.remove();
    });
}
```

Figure 27. Using a temporary link to download a file in Blob format.

Instead, a custom firewall with a Guard authenticator was created to handle the authentication. It was configured with the settings shown in Figure 28. Since modifying the request's headers on the frontend was impossible using the aforementioned attributes, the parameters were appended to the URL itself. By using a singleton service to attach the required data as GET parameters, it was possible to submit the data to the server with each file download request. The problem with GET parameters, is that they are visible in plaintext, e.g., "http://www.webapp.com?username=user&password=secret". However, that was partially solved by using a separate API key for only downloading files as the parameter.

```
firewalls:
    ...
    file_downloads:
        methods: [GET, POST]
        pattern: ^/file/[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}...
        stateless: true
        anonymous: true
        guard:
            provider: user_provider
            authenticators:
                - App\Security\ApiKeyParamAuthenticator
```

Figure 28. Firewall configuration for file downloads.

The Guard assigned to handling the file download endpoints was the ApiKeyParamAuthenticator. It implements standard Guard authenticator methods: supports, getCredentials, getUser, checkCredentials, start and onAuthenticationSuccess. The Guard supports requests that are attached with the GET parameter “ak” meaning API key. The parameter, which is basically considered the user’s credentials, is verified by using the Request object’s get method. Subsequently, the token is authenticated by running a database query using the APIKeyRepository’s findOne method and simply retrieving the createdBy property data from the APIKey entity (see Figure 29). If the parameter does not exist or the token is invalid or expired, the Guard naturally throws an error response.

```php
public function getUser($token): ?UserInterface
{
    $apiKey = $this->apiKeyResource->findOneBy(
        ['token' => $token]
    );

    if ($apiKey === null) {
        throw new HttpException(
            Response::HTTP_UNAUTHORIZED, 'Invalid or expired token'
        );
     }

    return $apiKey->getCreatedBy();
}
```

Figure 29. GetUser function of the authenticator.

Since the API key token was included in the URL string, it was possible for it to be leaked. For example, if a user opens an image in a new tab and sends the link to another user, they could get access to that picture without signing in. Afterwards, they could use the key to access other files if they happen to know their resource URLs. This is why the API keys have a very short lifespan and have to be renewed often. This allows users to link the images to someone else, while reducing the risk of misuse. When working with tokens, there is always a risk that they can be leaked or intercepted; however, keeping user information secure is much more critical for this system than hiding images and files.

5.2.2 Password Reset

Most of the backend resources were behind some sort of authentication. However, if a user forgets their password, they must be able to reset it without logging in first. Hence, one of the controller endpoints was configured as anonymous and without a Guard, so that it can be accessed by anyone. The endpoint takes in a username, date of birth and email address, which are used to send a password reset email to the user. In this system, multiple users can potentially share the same email, although it is not endorsed. The date of birth value is utilized to more effectively distinguish the users from each other.

As for the link, a temporary API key is generated and concatenated into the web application URL, after which it is sent to the user's email address. Accessing the link begins the authentication process by connecting to another unsecured anonymous endpoint. That endpoint has, however, a light authentication mechanism in the form of an API key check. If an API key with the token specified in the URL is found and has not expired, the password encoding begins. It is triggered by changing the user's plainPassword value, after which the PreUpdate callback is executed. In the lifecycle hook, the password is hashed with a secure algorithm and the plainPassword value is erased, as it should never be stored in the database. Subsequently, the client receives a signed token and is able to login as normal. Naturally, the temporary API key is removed after it has been exhausted or if a certain amount of time has passed. The logic is illustrated in Figure 30.

```
$token = $request->headers->get('Authorization');

$apiKey = $this->apiKeyResource->getRepository()->findOneBy([
    'token' => $token,
    'description' => 'Password Recovery'
]);

if ($apiKey === null) {
   throw new HttpException(
       Response::HTTP_NOT_FOUND, 'Api key not found'
   );
}

$user = $apiKey->getCreatedBy();

$user->setPlainPassword($request->get('password'));

$this->userResource->save($user);

$this->entityManager->remove($apiKey);
```

Figure 30. Password reset action.

6 User Roles and Access Control

6.1 User Roles

6.1.1 Role Descriptions

The TyPe application was designed with multiple user roles in mind. Starting from the lowest to the highest role they are *examinee*, *work manager*, *gate keeper* and *organization admin*. Examinees are considered "normal" users, whereas the rest are administrative users. The distinction between them is that the admin users are normally part of an organization or in a management position in one of their subsidiaries and examinees are other employees and freelancers. Technically, each role has access to features that are meant for users below them in the hierarchy, e.g., work managers can also take exams.

The majority of users are *Examinees*, who only take exams in the application. They have no rights to create or modify anything except for exam entities and their own user information. Rather than being part of organizations, these users generally belong to subsidiary companies. Examinees cannot see any other users or even course packages, only courses that are assigned to them and any related materials.

Work managers are administrators designated to specific course packages. Their primary purpose is to assign examinees to course packages and occasionally create new users. They can additionally track the exam results of users who belong to their company or companies. Similarly to examinees, work managers belong to subsidiaries more often than not, although it is possible have them in an organization. Even though they are administrators of course packages, they cannot modify the entity itself in any way.

Organization admin is the highest role available. As the name suggests, it is only available for users in an organization. Organization admins cannot only create courses, course packages, companies and users but they can modify every property available. Furthermore, it is up to organization admins to create all content in the application, including other users. They also have access to exam results and their users' information similarly to the other admin roles.

Gate keepers are similar to organization admins in that they can only belong to an organization. The main difference is that they cannot create new courses, course packages or companies. They can, however, create users and manage them, which includes attaching them to companies and course packages. The gate keeper role is designed for people whose job is to check which users have successfully taken an exam.

6.1.2 Changing the Active Company

The access control list is not only based on access roles, but also on companies. This means that access control entries include the ID value of the company, through which the access right is shared. This is necessary due to some edge cases, such as when a single work manager is designated to the same course package through more than one company or when an examinee is attached to a course package in similar fashion. Naturally, users can only have access rights that are connected to companies that they belong to.

TyPe was designed to be simplistic in that each user has a single active company, while they are logged in, to reduce the amount of information they take in at once. Since users can be part of multiple companies, there had to be an option to change the active company of the user. At first, when creating a user, they are also attached to a company making that their active company at that time. Afterwards, the user can switch their active company, after which a new token containing the necessary information is generated on the backend. Every request sent from the application includes the ID of the active company, which is used in the access control filter as described in more detail in chapter 6.3.2.

The difference between administrative users and examinees is that examinees cannot change their active company. Instead, they see all the courses that they have access to concurrently. When it comes to admins, one can only switch to a company in which their access role is of a similar level; organization admins and gate keepers can only change to a role in another organization, as changing to another company under the same organization is redundant. Work managers, on the other hand, can freely switch between different companies where they exist as work managers. Whenever a user accesses the application, a request is sent to the web service retrieving a list of

companies that the user can switch to. The code for retrieving the company data is showcased in Figure 31.

```
public function getAdminCompanies(string $userId): array
{
    return $this->createQueryBuilder('accEntry')
        ->select('
            DISTINCT c.id, c.name, accEntry.accessRole,
            org.id AS organization, owner.id as ownedBy'
        )
        ->innerJoin('accEntry.company', 'c')
        ->leftJoin('c.organization', 'org')
        ->leftJoin('c.ownedBy', 'owner')
        ->where('accEntry.subjectId = :userId')
        ->andWhere('accEntry.accessRole IN (:admins)')
        ->setParameter('userId', $userId)
        ->setParameter('admins', array(
            'PACKAGE_ADMIN', 'GATE_KEEPER', 'ORGANIZATION_ADMIN')
        )
        ->groupBy('c.id')
        ->getQuery()->getArrayResult();
}
```

Figure 31. Fetching available companies with Doctrine's QueryBuilder.

To allow admin-level users to also take exams without making the user interface too confusing, another feature, known as *Examination mode*, was developed. It allows users to temporarily set their own role to “examinee” for backend requests. Examination mode makes it so that one can only receive data relevant to examinees and additionally disables company switching, until they change back to Administrative mode.

6.2 The Importance of Access Control

Requests can be sent to API endpoints via programs such as Postman. The requests can include headers, payloads and anything that web applications includes in them, including the access token. Therefore, unauthorized people can try to fish for protected data if they are aware of the resource URLs. Access control is essential to software solutions for making sure that users can only access resources available to them.

Many web services and APIs consist of both secure and unsecure endpoints that require no authentication. When designing unsecure endpoints, it is imperative to disallow access to any protected resources. For instance, when using serialization groups, they should be manually set on the backend to essentially avoid a kind of SQL injection. Otherwise, one could populate an entity and all of its relational properties to gain access to them. With large-scale modern web software consisting of huge amounts of features, it can be difficult to keep everything secured. Nonetheless, a robust access control system can thwart many unwanted breaches of security.

Naturally, keeping user information secure from unauthorized people is imperative, especially because of the European Union's General Data Protection Regulation. One of the primary adjustments that developers have to make is that user information has to be deleted from systems that have no necessity to handle it. According to GDPR's terms, personal information includes names, addresses, phone numbers, tracking information and even IP addresses among other things. (Henkilötietojen käsittelyä koskevat periaatteet n.d.) Software companies must make sure to keep this data from the hands of unauthorized people.

6.3 Base Implementation

6.3.1 Access Control List

Access control list is a sort of map that describes which users have access privileges to which resources. General permission to entity resources for all of the roles was defined in the *AccessRoles.php*. The rights are mapped by role, e.g., ORGANIZATION_ADMIN, and *permission*, which includes read, create, update, delete, share and revoke (see Figure 32). Sharing and revoking is related to adding users to companies or course packages.

```
const ORGANIZATION_ADMIN = [
    User::class =>
          Permissions::CONTENT_READ
        | Permissions::CONTENT_CREATE
        | Permissions::CONTENT_UPDATE
        | Permissions::CONTENT_DELETE
    ,
    Company::class =>
          Permissions::CONTENT_READ
        | Permissions::CONTENT_CREATE
        | Permissions::CONTENT_SHARE
        | Permissions::CONTENT_REVOKE
    ,
    'Default' =>
    ...
];
```

Figure 32. Access rights to resources for organization admins.

To enable the access control list on a controller, an annotation can be added into the controller and any resources in it, for instance "@AccessControlRequirement(permissions={'CONTENT_READ'})". Afterwards, *voters* make sure that the user, which is retrieved by utilizing the user provider, meets the authentication requirements to access the resource. Voters are officially recommended by Symfony for these types of security implementations (Security n.d). At the beginning of every resource endpoint, the AuthorizationChecker service inspects the voter results and either continues executing the code or returns a 403 forbidden error.

6.3.2 Access Control Filter

Symfony has a built in event that is dispatched after the controller for an endpoint has been resolved, immediately before executing the code. The event can be subscribed to with the *onKernelController* function. (Built-in Symfony Events n.d.) In this function, the access control service is instantiated and the access control filter is enabled. The filter requires an annotation reader, entity manager and the user entity to perform necessary authentication steps. Additionally, the active company id and access role are fetched from the query parameters and forwarded to the access control filter. If the values are missing, the request will throw an error.

The access control filter is essentially an extra SQL Inner Join for each Select query. It combines the selected table with the access control table by using the domain id. If the access control table includes a row with the specified domain ID, subject ID equals the user's ID, access role matches the user's role and company ID matches the user's active company, the query is able to find one or more rows. Inner Joins make it so that both tables must have data with the matching requirement. In addition, a Union Select on the company table makes sure that the user has the required access rights to the company. A simplified version of the access control filter logic is shown in Figure 33. In certain scenarios, the filter can be disabled temporarily to perform database queries while ignoring the access control checks.

```
public function addFilterConstraint()
{
    if ($restrict->byCommonAccess) {
        return $this->addUserFilterConstraint();
    }

    $select = "SELECT e_.id FROM
        {$targetEntity->getTableName()} e_";
    $aclJoin = "INNER JOIN access_control a_ ON (a_.subject_id =
        $userId".
    "AND a_.domain_id = $joinAlias.$joinColumnName)";
    $whereRole = "WHERE a_.access_role = '$this->accessRole'";
    $whereNotDeleted = 'AND a_.deleted_at IS NULL';
    $whereCompany = "AND a_.company_id = '$this->activeCompanyId'";

    return "$targetTableAlias.id IN (
        "$select{$joins}
        $aclJoin $whereRole $whereCompany $whereNotDeleted";
    )";
}
```

Figure 33. Simplified access control filter logic.

As for user entities, there are no access control entries with them as the domain. Instead, any administrative user can see all users in their respective companies. The user filter simply creates a Join into the access control table and fetches users that belong to the company with a raw SQL query (see Figure 34). However, the query is slightly different for work managers and other users. For organization admins and

gatekeepers, the query also includes other companies attached to the user's company, whereas work managers only have access to their own company and they can only view examinee-level users.

```
private function addUserFilterConstraint(): string
{
    ...
    switch ($accessRole) {
        case 'ORGANIZATION_ADMIN':
        case 'GATE_KEEPER':
            ...
            break;
        case 'PACKAGE_ADMIN':
            $q = "SELECT a0_.subject_id ".
                "FROM access_control a0_ ".
                "INNER JOIN access_control a1_ ON (".
                "a1_.subject_id = $userId AND ".
                "a1_.domain_id = a0_.domain_id AND ".
                "a1_.company_id = a0_.company_id AND ".
                "a1_.access_role = 'PACKAGE_ADMIN') ".
                "WHERE a0_.subject_id = $userId ".
                "OR a0_.access_role = 'EXAMINEE' ".
                "AND a0_.company_id = '$this->activeCompanyId' ".
                "AND a0_.deleted_at IS NULL ".
                "GROUP BY a0_.subject_id";
            break;
    }
    ...
```

Figure 34. User filter constraint for work managers (PACKAGE_ADMIN).

6.4 Access Control Entries

6.4.1 Entity Design

The access control entries work the same way as other Symfony entities. They are mapped to a single table in the database with the most important properties being the subject, domain, company and access role. The properties are connected to the original entities by Foreign Keys making it easier to create complex SQL queries with multiple Joins (see chapter 6.5.1 for an example). All of the properties, except company, are required to map the entries onto the appropriate entities.

Subject refers to the user who has an access control right to an entity. The target entity, on the other hand, is known as the *domain*. It consists of two properties, domain

id and domain name, which are the Primary Key of the domain entity and its name, e.g., “App\Entity\Course”. Most access control entries also have a company property, which indicates the company through which the right is given. When accessing backend resources, the user’s active company is used to filter access control entries based on the company ID. Lastly, access role is self-explanatory in that it simply specifies the role as a string value, for instance “EXAMINEE” or “ORGANIZATION_ADMIN”.

Access rights are reliant on many modifiers. In this application, users can gain access by way of a certain role, company or a course package. For example, a single user can be an administrator of a course, even though they cannot take the exam for that course without having examinee-level access rights. All of the variable data is included in the access control entry increasing its complexity. Otherwise, the access control rights would be impossible to manage afterwards.

6.4.2 Creating Entries

The access control rights are generally created in entity lifecycle callbacks (see chapter 4.3.3), mostly in PostPersist and PostUpdate. In the case of a new course, its creator gets access control entries for that course with both an “OWNER” role as well as an “ORGANIZATION_ADMIN” role. Every other organization admin in the same company also obtains the latter. The same principle applies to materials such as files and images, as shown in Figure 35. Rights to the materials are inherited from the parent entity, for instance a course or course package. In the case of course packages, the rights are propagated to courses included in it, so that the work managers and examinees are allowed to access it.

```
foreach ($entity->getFiles()->getIterator() as $file) {
    $accessControlService->extendRights(
        $entity, $file, null, AccessRoles::all(true)
    );
}
```

Figure 35. Propagation of access rights to child entities.

Access rights are mainly shared via a user's company or organization. Depending on if the company is an organization or not, the logic is slightly different, as demonstrated in Figure 36. Occasionally, access control entries are also added or removed from users in another company, for instance when a single course package is shared between two or more companies. Whenever a new user is added to a company, they also get access to the necessary data depending on their access role. Work managers and examinees, however, do not get access to anything by default.

```
if (
    !$currentUserCompany->isOrganization() &&
    $currentUser->getHighestRole() === 'PACKAGE_ADMIN'
    )
) {
    $accessControlService->extendRights(
        $currentUserCompany,
        $entity,
        $currentUserCompany->getOwnedBy()->getCompany()->getId(),
        ['ORGANIZATION_ADMIN', 'GATE_KEEPER', 'PACKAGE_ADMIN']
    );
}

else if ($currentUserCompany->getId() !== $company->getId()) {
    $accessControlService->extendRights(
        $currentUserCompany,
        $entity,
        $currentUserCompany->getId(),
        ['ORGANIZATION_ADMIN', 'GATE_KEEPER', 'PACKAGE_ADMIN']
    );
}
```

Figure 36. Creating entries depending on the access role and company type.

6.5 Problematic Cases

6.5.1 Retroactive Access Rights

In the application, users' contact information is tied to their company. Therefore, since any user can belong to multiple companies, they could also have more than one email and phone number. The contact information is stored in UserDetail entities, which are associated with the user and the company. Originally, the data was simply stored on the user entity instead; however, it was deemed necessary to allow

multiple sets of contact information. Another benefit of a separate entity is that the *Blameable* trait can be utilized. It saves the ID of the user who created the entity, so that their information can be tracked later on. Using this method, it is possible to retrieve the contact information of the work manager who attached an examinee to a course package, and show it to the user. Furthermore, the same principle applies to updating and deleting.

Due to changes in the application code after the TyPe application product was released, certain administrative-level users lost their access rights to some users' contact information. In order to adjust the entries to the revamped access control logic, it was necessary to create a migration that would executed in the production database. Normally, migrations are used to alter tables or modify the data using Where conditions and Joins. However, this problem could not be solved with a simple Update query using Joins.

The issue was fixed by using an SQL Stored Procedure, which is similar to a Function. However, it is more flexible, as it allows multiple Select, Case and If statements inside of it (Wenzel n.d). The procedure makes use of SQL Cursors, which retrieve a single row at a time from the database using complex Select queries. The rows can then be iterated through and used in declaring variables. This allows creating new access control entries inside the iterative loops using those variables. The logic for creating the necessary entries works as follows:

1. Cursor finds all UserDetail entities and iterates through them in a loop. Each of the entities has a companyId value for later use.
2. Inside the loop, another Cursor Selects all users with an administrative access role in the company attached to the UserDetail and iterates through them as well. This is done by adding Inner Joins from the user table through the company_has_user table into the company table and checking that the user has administrative rights to said company. Additionally, a Left Join is used to check if the company has a relationship to an organization entity. The Cursor utilizes values retrieved from the previous Cursor Select query.
3. Inside the inner loop, a new access control entry entity is created for that user to that UserDetail with the same access role and the same company. In the case of organization admins, the companyId property must be the organization's ID instead of

the company's. The IDs for new entries are created by combining SQL's functions, such as HEX, FLOOR, RAND, LPAD and CONCAT, to achieve the semi-random UUIDv4 format. This is done, because SQL's generic UUID function creates a predictable, less random UUIDv1.

Although, this algorithm results in some duplicate entries, they can be easily removed afterwards. Additionally, it is worth noting that executing the procedure with approximately 100 users can take up to 15 seconds, due to amount of queries and Joins. For the full procedure code, see Appendix 2.

6.5.2 Creating Duplicate Entities

Each real-life company can only have a single company entity matching it in the application. However, multiple organizations can still "own" the same company. When creating a company, one of the required properties is the business ID, which is used as a unique constraint in the database. After typing in the business ID in the correct format, a request is sent in the background. The request checks whether a company with the business ID already exists, ignoring the standard access control procedures. Instead, the endpoint only checks that the user is an organization admin. Although it is not ideal to be able to find entities without proper access, it was essentially the only way handle this case without confusing the users.

If the database query finds an existing company entity, it is added under the user's organization. The difference from normally creating a company is that the organization user do not receive owner rights. Therefore, only the first organization to create a company can modify its properties, although others can still add users to it. In that case, both organizations have access to those users.

7 Discussion

7.1 Conclusions

7.1.1 Accomplishments

From the clients' perspective, the application had many praiseworthy features. Specifically, it received positive feedback for its ease-of-use and intuitiveness, especially on the course form shown in Figure 37. The credit for that certainly goes to the development team's user experience expert. Moreover, it was a good idea to wait for clients' feedback and then refactor certain components and features with the feedback in mind. The course form used for creating and editing courses was especially well-received.

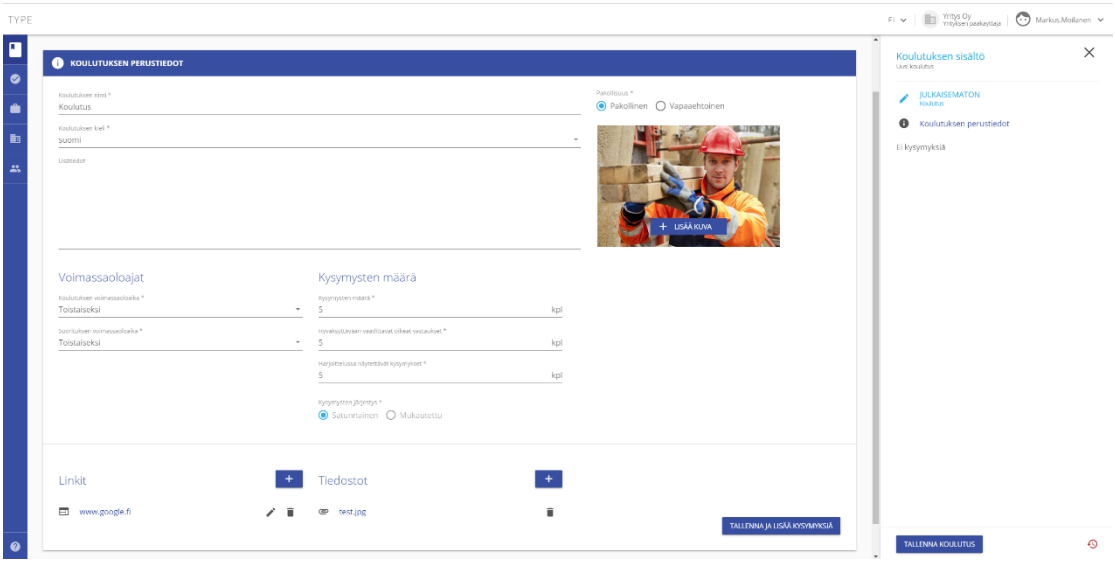


Figure 37. Course form view.

Although the access control implementation was not without flaws, it certainly worked as intended. The base implementation, including the entity, access control filter, propagation functions and the inheritance of access rights, was intelligently designed. It can surely be utilized again in future projects due to its flexibility. There is also a lot to learn from developing it.

The application was designed with best practices in multiple areas. Thanks to the knowledge from the team's most experienced developers, many pitfalls were avoided. Issues were spotted in code review and quality assurance testing and promptly fixed afterwards. The team followed naming conventions for files, resources such as endpoints, tables and components. Although the developers did not have much experience with Angular or state management, they handed Observables, the application state and the ngrx middleware appropriately and efficiently.

7.1.2 Design Flaws

The controllers in the web service contained many actions, which are endpoints that handle various operations. It would have been better to implement generic REST endpoints by using a bundle like the FOSRestBundle. They allow easily modifying entities if implemented properly. Additionally, array type properties should be configured to use *adder* methods instead of *setter* methods, so that every related child row does not have to be altered on each update. That was how the controller endpoints worked at first, but Symfony forms and the access control implementation created some issues it. There might be better input validation bundles available or perhaps Symfony forms could be customized somehow.

As mentioned in chapter 4.2.24.2.2, organizations can have other organizations as their subsidiaries. Taking it one step further, a subsidiary organization could technically have another organization under it. This results in an incredible amount of database queries and long iterative loops when creating access control entries for those companies. Looking back at this functionality, it would have been better to disallow subsidiary organizations in favor of a two-level design with organizations on top and normal companies below them. Even if the clients actually made use of this feature, as it stands, it would be extremely confusing to them.

7.1.3 Improvement Possibilities

The access role names are generic. One way to cater to different clients' needs would be to allow customizable role names and even names for entities. For instance, some client might prefer to call companies subsidiaries instead. That way, it would be less

confusing for the end-users, since they can use terminology and jargon from their respective occupations. The main issue, however, is that Finnish language has many conjugations and they would have to be configurable as well.

The complexity of the access control service's logic makes some of the queries take a long time. During this time, the users have to wait for the response before continuing to use the application. It would definitely be better to run those tasks asynchronously by adding them to a process queue, which would be executed with a first-in-first-out logic. Granted, if there are many tasks in the queue, it might take a while for users to see the changes in the application, but it would be far better than forcing them to wait. Similarly, the process queue could also be utilized by the email and text message services, which can take a while to execute the message delivery.

Serializing entities to response payloads takes not only time, but also processing power. Optimizing the population options would be the first step to achieving faster performance in the system. For instance, when populating course entities in a Course package, the population options could be changed from "CoursePackage.courses, Course" to "set.CoursePackageList". Then, the set group can be added to each property that should be included in the set. Not only does this make it faster in certain situations, but it also makes it easier to maintain the software in the long term. Another improvement to property population would be to include certain *count* values. In the previously mentioned course package list example, instead of populating each course entity, it might be better to include only the number of courses and possibly the number of unpublished courses as well. Knowing other information about the courses is unnecessary. Other than the optimizations for serialization, this also reduces the time it takes for the access control filter to run, since it is executed for each processed entity.

There is a generic search feature in each list view. The search is limited to string-based properties, such as names and descriptions. Instead, it would be far better to be able to filter based on other criteria, including relations and Boolean properties. Some examples of this include finding courses based on course packages, users based on companies, published or unpublished courses, exam results that are valid

or expired, et cetera. The search criteria should optimally be implemented in a generic way, with a trait or service, to reduce development time and not hinder the maintainability.

7.2 Lessons to Learn

The main problem that slowed down the development was the less than adequately designed database schema. At first, users could have a single set of emails and phone numbers, which was changed later on. Another big change was when course packages could be shared between organizations, which meant that the relation had to be changed from one-to-many to many-to-many among other issues. Similar code refactoring had to be made often, resulting in extra development time.

Deadlines are something that developers have to constantly deal with at work. Since the development of the system took longer than expected, the project's release date came quickly. The team knew that they had a hard time creating a stable product on release, which meant that certain shortcuts had to be taken to reach that state. This ended up hurting the development in the long run, due to the technical debt of having to refactor code. Instead of developing features with potential programming flaws, it would have been better to spend the extra time and even delay the release if necessary. This also comes back to the planning issues.

When developing a feature, it is common to split it into smaller tasks. Generally, it is a good thing, although it can also create issues later on. Other code changes might affect that feature or make it more difficult to implement in some way. In the worst case scenario, it might even get forgotten or pushed lower into the backlog of tasks. In addition, the developer or developers who originally worked on a feature might not be working on the project anymore, meaning that programming the feature takes more time. The split tasks should be prioritized well and preferably developed right after the main task in many cases.

Most problems that developers run into have already been solved in the past. Therefore, it is better to reuse existing solutions, rather than try to reinvent them. It is important to use libraries and bundles that are publically available, as long as they are

secure and do not pose other problems. Although it is difficult to say what kind of packages could have been used in the project, there are surely a few of them.

References

10 best Node.js frameworks in 2018. 2018. Blog post on DA-14's website. Accessed on 02.11.2018. Retrieved from https://da-14.com/blog/10-best-nodejs-frameworks.

Anser, M. 2017. GraphQL *:: A data query language.* Article on Medium's website. Accessed on 27.10.2018. Retrieved from https://medium.com/@ansertechgeek/graphql-a-data-query-language-e22e2d2f8eeb.

Box, D. 2001. *A Brief History of SOAP.* Article on XML.com website. Accessed on 12.10.2018. Retrieved from https://www.xml.com/pub/a/ws/2001/04/04/soap.html.

Built-in Symfony Events. N.d. Symfony's official documentation. Accessed on 30.12.2018. Retrieved from https://symfony.com/doc/current/reference/events.html.

Carpenter, J. 2018. *Why you should use a graph database.* Article on InfoWorld's website. Accessed on 03.11.2018. Retrieved from https://www.infoworld.com/article/3251829/nosql/why-you-should-use-a-graph-database.html.

Databases and Doctrine ORM. N.d. Symfony's official documentation. Accessed on 13.01.2019. Retrieved from https://symfony.com/doc/current/doctrine.html.

DB-Engines Ranking - Trend Popularity. N.d. Statistical distribution on DB-Engines' website. Updated in autumn of 2018. Accessed on 28.10.2018. Retrieved from https://db-engines.com/en/ranking_trend.

Difference Between RPC and RMI. 2017. Article on TechDifferences' website. Accessed on 28.10.2018. Retrieved from https://techdifferences.com/difference-between-rpc-and-rmi.html.

Digitalisaatio. N.d. Accessed on 13.01.2019. Retrieved from https://www.protacon.com/digitalisaatio/.

Events. N.d. Doctrine's official documentation. Accessed on 13.01.2019. Retrieved from https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html.

Forms. N.d. Symfony official documentation. Accessed on 25.12.2018. Retrieved from https://symfony.com/doc/current/forms.html.

Gilmore, J. 2018. *The importance of loose coupling in REST API design.* Article on dreamfactory's website. Accessed on 27.10.2018. Retrieved from http://blog.dreamfactory.com/the-importance-of-loose-coupling-in-rest-api-design/.

Hammink, J. 2018. *The Types of Modern Databases.* Blog post on alooma's website. Accessed on 28.10.2018. Retrieved from https://www.alooma.com/blog/types-of-modern-databases.

Hansen, S. 2017. *7 Good Reasons to Use Symfony Framework for Your Project.* Article on Hackernoon's website. Accessed on 28.10.2018. Retrieved from

https://hackernoon.com/7-good-reasons-to-use-symfony-framework-for-your-project-265f96dcf759.

Harris, D. 2016. *MongoDB co-creator explains why 'NoSQL' came to be, and why open source mastery is an elusive goal.* Article on Medium's website. Accessed on 02.11.2018. Retrieved from https://medium.com/s-c-a-l-e/mongodb-co-creator-explains-why-nosql-came-to-be-and-why-open-source-mastery-is-an-elusive-goal-3a138480b9cd.

Helfer, J. 2016. *GraphQL vs. Falcor.* Blog post on Medium's website. Accessed on 27.10.2018. Retrieved from https://blog.apollographql.com/graphql-vs-falcor-4f1e9cbf7504.

Henkilötietojen käsittelyä koskevat periaatteet. N.d. Article on tietosuojamalli's website. Updated on 21.05.2017. Accessed on 13.01.2019. Retrieved from https://fakta.tietosuojamalli.fi/gdpr-asetus/5-henkilotietojen-kasittelya-koskevat-periaatteet.

Homan, J. 2014. *Relational vs. non-relational databases: Which one is right for you?* Blog post on Pluralsight's website. Accessed on 02.11.2018. Retrieved from https://www.pluralsight.com/blog/software-development/relational-non-relational-databases.

Hoppe, T. 2015. *REST APIs Today and Tomorrow*. Presentation on a GitHub pages website. Accessed on 05.01.2019. Retrieved from http://vanthome.github.io/rest-api-essay-presentation/rest_apis.html.

How to Manually Encode a Password. N.d. Symfony's official documentation. Accessed on 26.01.2019. Retrieved from https://symfony.com/doc/4.0/security/password_encoding.html.

How to Organize Configuration Files. N.d. Symfony's official documentation. Accessed on 13.01.2019. Retrieved from https://symfony.com/doc/current/configuration/configuration_organization.html.

Introduction to JSON Web Tokens. N.d. Manual on JWT.io website. Accessed on 17.01.2019. Retrieved from https://jwt.io/introduction/.

Kumar, D. N.d. *Best practices for building RESTful web services.* Pdf document on Infosys' website. Accessed on 21.10.2018. Retrieved from https://www.infosys.com/digital/insights/Documents/restful-web-services.pdf.

Lease, D. 2018. *TypeScript: What is it & when is it useful?* Article on Medium's website. Accessed on 10.11.2018. Retrieved from https://medium.com/front-end-hacking/typescript-what-is-it-when-is-it-useful-c4c41b5c4ae7.

Mueller, J. 2013. *Understanding SOAP and REST Basics And Differences.* Blog post on Smartbear's website. Accessed on 11.10.2018. Retrieved from https://smartbear.com/blog/test-and-monitor/understanding-soap-and-rest-basics/.

Nasseri, A. 2016. *JSON Web Tokens: Artsy's* Journey. Blog post on Artsy's GitHub pages. Accessed on 26.01.2019. Retrieved from http://artsy.github.io/blog/2016/10/26/jwt-artsy-journey/.

Nayyar, A. 2018. *Why MariaDB Scores Over MySQL*. Article on OpenSourceForU's website. Accessed on 02.12.2018. Retrieved from https://opensourceforu.com/2018/04/why-mariadb-scores-over-mysql/.

Patel, P. 2018. *What exactly is Node.js?* Article on Medium freeCodeCamp's website. Accessed on 02.11.2018. Retrieved from https://medium.freecodecamp.org/what-exactly-is-node-js-ae36e97449f5.

Pohjolainen, J. 2016. *Java RMI.* Lecture material on Tampere University of Applied Sciences' website. Accessed on 28.10.2018. Retrieved from http://koti.tamk.fi/~pohjus/java/lectures/rmi.html.

Reeve, A. 2012. *Big Data and NoSQL: The Problem with Relational Databases.* Article on InFocus DellEMC's website. Accessed on 02.11.2018. Retrieved from https://infocus.dellemc.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases/.

REST in peace, SOAP. 2010. Article on royal.pingdom.com website. Accessed on 06.10.2018. Retrieved from https://royal.pingdom.com/2010/10/15/rest-in-peace-soap/.

Rytov, O. N.d. *Why I Decided To Embrace Laravel.* Article on Toptal's website. Accessed on 28.10.2018. Retrieved from https://www.toptal.com/laravel/why-i-decided-to-embrace-laravel.

Sakhibgareev, K. N.d. *PHP Frameworks: Choosing Between Symfony and Laravel.* Article on Toptal's website. Accessed on 28.10.2018. Retrieved from https://www.toptal.com/php/choosing-between-symfony-and-laravel-frameworks.

Sasaki, B. 2018. *Graph Databases for Beginners: Why Graph Technology Is the Future*. Blog post on neo4j.com website. Accessed on 06.01.2019. Retrieved from https://neo4j.com/blog/why-graph-databases-are-the-future/.

Sanjna, V. 2018. *APIs versus web services.* Blog post on MuleSoft's website. Accessed on 11.10.2018. Retrieved from https://blogs.mulesoft.com/dev/api-dev/apis-versus-web-services/.

Security. N.d. Symfony's official documentation. Accessed on 13.01.2019. Retrieved from https://symfony.com/doc/current/security.html.

Shah, App. N.d. *Basic WSDL Structure Understanding – (Web Service Description Language) Explained.* Blog post on Crunchify's website. Updated on 14.07.2018. Accessed on 20.10.2018. Retrieved from https://crunchify.com/basic-wsdl-structure-understanding-wsdl-explained/.

Skvorc, B. 2015. *The Best PHP Framework for 2015: SitePoint Survey Results.* Article on SitePoint's website. Accessed on 28.10.2018. Retrieved from https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/.

SOAP Vs. REST: Difference between Web API Services. N.d. Tutorial on Guru99's website. Accessed on 12.10.2018. Retrieved from https://www.guru99.com/comparison-between-web-services.html.

Sturgeon, P. 2016. *Understanding RPC Vs REST For HTTP APIs.* Article on Smashing magazine's website. Accessed on 27.10.2018. Retrieved from

https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/.

The History of REST APIs. 2016. Blog post on readme.io website. Accessed on 27.10.2018. Retrieved from https://blog.readme.io/the-history-of-rest-apis/.

The Serializer Component. N.d. Symfony's official documentation. Accessed on 19.01.2019. Retrieved from https://symfony.com/doc/current/components/serializer.html.

The structure of a SOAP message. N.d. Documentation on IBM's website. Updated on 19.09.2018. Accessed on 21.10.2018. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSMKHH_10.0.0/com.ibm.etools.mft.doc/ac55780_.htm.

The ultimate guide to API architecture: REST, SOAP or GraphQL. 2018. Blog post on DA-14's website. Accessed on 27.10.2018. Retrieved from https://da-14.com/blog/ultimate-guide-api-architecture-rest-soap-or-graphql.

Tutorial – Example of a SOAP message. 2012. Shared file on GitHubGist.com website. Accessed on 21.10.2018. Retrieved from https://gist.github.com/lamprosg/2151619.

Wenzel, K. N.d. *Learn about Stored Procedures*. Blog post on essentialSQL.com website. Accessed on 19.01.2019. Retrieved from https://www.essentialsql.com/what-is-a-stored-procedure/.

What Are Web Services and Where Are They Used? 2013. Article on Segue Technologies' website. Accessed on 29.09.2018. Retrieved from https://www.seguetech.com/web-services/.

What is a Column Store Database? 2016. Article on Database.guide's website website. Accessed on 03.11.2018. Retrieved from https://database.guide/what-is-a-column-store-database/.

What is a Database Schema? 2016. Article on Database.guide's website. Accessed on 02.11.2018. Retrieved from https://database.guide/what-is-a-database-schema/.

What is REST? N.d. Article on Code academy's website. Accessed on 12.10.2018. https://www.codecademy.com/articles/what-is-rest.

Why the Hell Would You Use Node.js. 2017. Article on Medium's website. Accessed on 02.11.2018. Retrieved from https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e.

Wieruch, R. 2018. *Why GraphQL: Advantages, Disadvantages & Alternatives.* Article on Robin Wieruch's website. Accessed on 21.10.2018. Retrieved from https://www.robinwieruch.de/why-graphql-advantages-disadvantages-alternatives/.

Wodehouse, C. N.d. *SOAP vs. REST: A Look at Two Different API Styles.* Article on Upwork's website. Accessed on 20.10.2018. Retrieved from https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/.

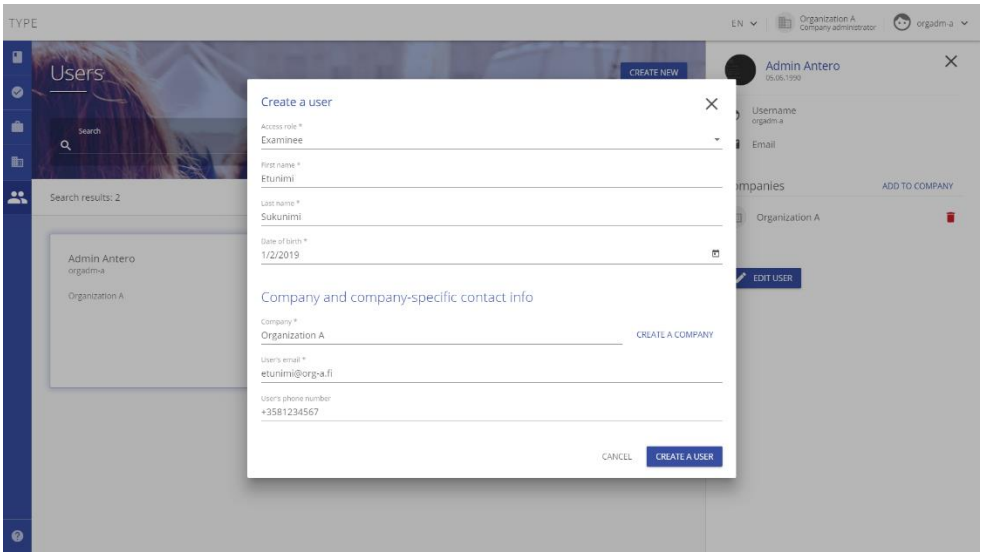
Working with Associations. N.d. Doctrine's official documentation. Accessed on 13.01.2019. Retrieved from https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/working-with-associations.html.

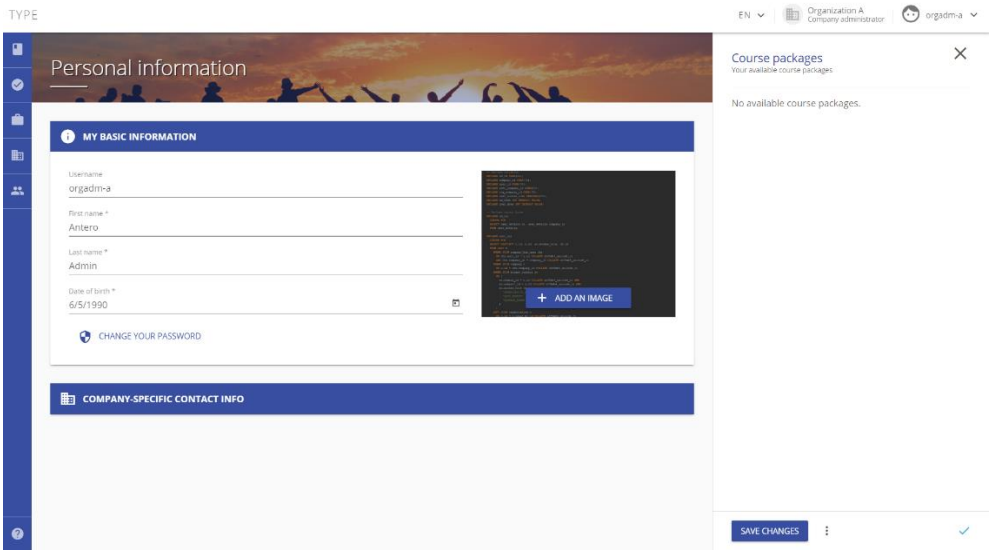
Yegulalp, S. 2017. *What is NoSQL? NoSQL databases explained.* Article on InfoWorld's website. Accessed on 03.11.2018. Retrieved from https://www.infoworld.com/article/3240644/nosql/what-is-nosql-nosql-databases-explained.html.

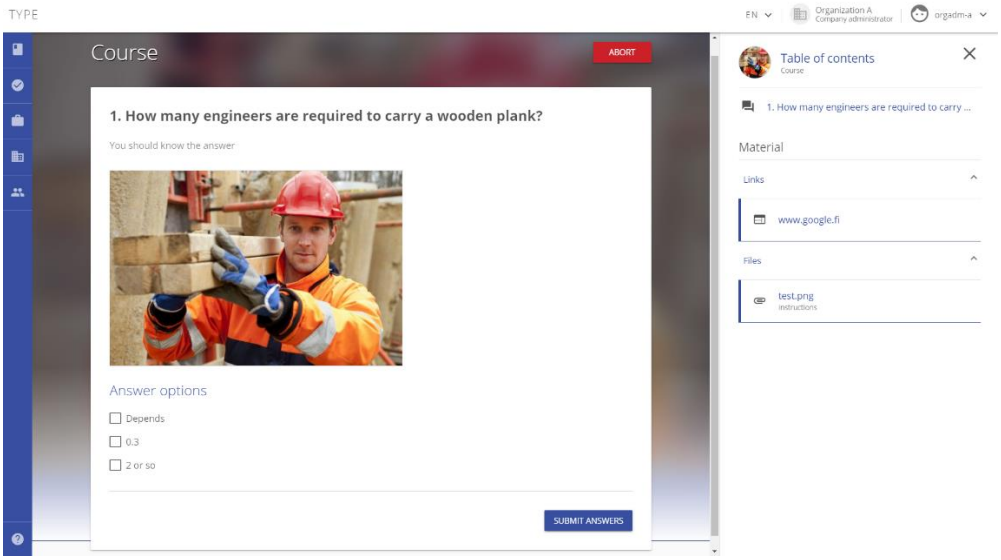
Yellavula, N. 2018. *A guide for adding JWT token based authentication to your single page NodeJS applications*. Article on Medium's website. Retrieved from https://medium.com/dev-bits/a-guide-for-adding-jwt-token-based-authentication-to-your-single-page-nodejs-applications-c403f7cf04f4.

Appendices

Appendix 1. Screenshots of the TyPe web application.







Appendix 2. Stored Procedure for adding access rights to user information.

```
delimiter //

CREATE PROCEDURE add_user_detail_rights()
BEGIN
-- Declare variables
DECLARE ud_id CHAR(36);
DECLARE company_id CHAR(36);
DECLARE user_id CHAR(36);
DECLARE user_company_id CHAR(36);
DECLARE org_company_id CHAR(36);
DECLARE user_access_role VARCHAR(255);
DECLARE ud_done INT DEFAULT FALSE;
DECLARE user_done INT DEFAULT FALSE;

-- Declare cursor loops
DECLARE ud_cur
  CURSOR FOR
  SELECT user_details.id, user_details.company_id
  FROM user_details;

DECLARE user_cur
  CURSOR FOR
  SELECT DISTINCT c.id, u.id, ac.access_role, c2.id
  FROM user u
    INNER JOIN company_has_user chs
      ON chs.user_id = u.id
      AND chs.company_id = company_id
    INNER JOIN company c
      ON c.id = chs.company_id
    INNER JOIN access_control ac
      ON (
        ac.domain_id = c.id AND
        ac.subject_id = u.id AND
        ac.access_role IN (
          'ORGANIZATION_ADMIN',
          'GATE_KEEPER',
          'PACKAGE_ADMIN'
        )
      )
    LEFT JOIN organization o
      ON o.id = c.owned_by_id
    LEFT JOIN company c2
      ON c2.id = o.company_id

DECLARE CONTINUE HANDLER
FOR NOT FOUND
SET ud_done = TRUE;

-- Outer loop block. Goes over each user detail.
OPEN ud_cur;
ud_loop: LOOP
  FETCH ud_cur
  INTO ud_id, company_id;
```

```
  NESTED_LOOP: BEGIN
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET user_done = TRUE;

    OPEN user_cur;
    user_loop: LOOP
      FETCH user_cur
      INTO user_company_id, user_id, user_access_role, org_company_id;

      -- Create a new entry for each role associated with a company
      IF user_id IS NOT NULL THEN
        INSERT INTO access_control (
          id,
          subject_id,
          domain_id,
          domain_name,
          access_role,
          company_id,
          created_at
        )
        VALUES (
          -- Create a semi-random UUID (v4)
          LOWER(CONCAT(LPAD(HEX(FLOOR(RAND() * 0xffff)), 4, "0") ...)),
          user_id,
          ud_id,
          'App\\Entity\\UserDetail',
          user_access_role,
          user_company_id,
          NOW()
        );

        -- Same values, except the company is an organization
        IF (org_company_id IS NOT NULL
        AND user_access_role = 'ORGANIZATION_ADMIN') THEN
          INSERT INTO access_control (
            ...
          )
          VALUES (
            ...,
            org_company_id,
         );
        END IF;
      END IF;

      IF user_done THEN LEAVE user_loop;
      END IF;
      END LOOP user_loop;
    CLOSE user_cur;
  END NESTED_LOOP;

    -- Reset nested 'not found handler' after each outer loop cycle
  SET user_done = 0;

  IF ud_done THEN LEAVE ud_loop;
  END IF;
  END LOOP ud_loop;
CLOSE ud_cur;

END
//

delimiter ;
```