Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Web Application onlajny.com with Support for Real-time Communication and Machine Readable Data

Bc. Michal Toman

Supervisor: Ing. Martin Ledvinka, Ph.D.
Field of study: Open Informatics
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Toman Michal**	Personal ID number:	**483502**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Web Application onlajny.com with Support for Real-time Communication and Machine Readable Data

Master’s thesis title in Czech:

Webová aplikace systému onlajny.com s podporou real-time komunikace a strojově čitelných dat

Guidelines:

1. Become familiar with the latest trends in web application development, bi-directional client-server communication and machine readable data support.
2. Analyze suitable technologies and approaches to creating a new version of onlajny.com. Pay particular attention to real-time client side data updates as well as machine readable backend APIs.
3. Design the new onlajny.com system. The new application should besides the new machine readable API also support legacy clients using the old XML-based API.
4. Implement the newly designed version of onlajny.com, including a new user interface (whose graphical design may be provided by a web designer).
5. Evaluate your solution by user testing with at least five subjects. Ensure also backwards compatibility of your API with respect to the current clients of onlajny.com.

Bibliography / sources:

[1] Wood D., Zaidman M., Ruth L., Hausenblas M.: Linked Data, Manning, 2013.
[2] Fielding R.: Architectural Styles and the Design of Network-based Software Architectures, 2000.
[3] Ater T.: Building Progressive Web Apps: Bringing the Power of Native to the Browser, O'Reilly Media, 2017

Name and workplace of master’s thesis supervisor:

Ing. Martin Ledvinka, Ph.D. Knowledge-based Software Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **26.01.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Martin Ledvinka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank the supervisor, Ing. Martin Ledvinka, Ph.D., for his patience and helpfulness during ongoing consultations throughout the writing of this thesis. Also, I would like to thank my colleagues at eSports.cz for the opportunity to implement the Onlajny project as part of my master's thesis.

Declaration

Abstract

This master’s thesis focuses on the revitalization of the Onlajny.com system. The new version of the system was requested by the company eSports.cz. The application will support real-time communication and machine-readable data. This document consists of an analysis of the current state of the system, an analysis of the technology to be used for the new version, the design of the new system, its implementation, and the evaluation of the created solution. The main goal of the thesis is to create a new Onlajny application, which can be further developed into a comprehensive system and will replace the current version.

Keywords: Node.js, Next.js, JavaScript, TypeScript, REST, web, JSON-LD, semantic web, WebSockets

Supervisor: Ing. Martin Ledvinka, Ph.D.

Abstrakt

Tato diplomová práce se zaměřuje na revitalizaci systému Onlajny.com. Novou verzi systému si vyžádala společnost eSports.cz. Aplikace bude podporovat komunikaci v reálném čase a strojově čitelná data. Dokument obsahuje analýzu stávajícího systému a analýzu technologií, které budou použity v nové verzi. Dále je součástí textu popis implementace nového systému, jeho návrh a vyhodnocení vytvořeného řešení. Hlavním cílem práce je vytvořit novou aplikaci Onlajny, jež může být dále rozvinuta a nahradí stávající verzi.

Klíčová slova: Node.js, Next.js, JavaScript, TypeScript, REST, web, JSON-LD, sémantický web, WebSockets

Překlad názvu: Webová aplikace systému onlajny.com s podporou real-time komunikace a strojově čitelných dat

Contents

Figures

Tables

Chapter 1
Introduction

When choosing the topic for my master's thesis, I turned to my job and searched for some projects that would be satisfactory. And so I discovered the web application Onlajny.com[1], which is outdated and could be improved.

The application provides live statistics and other live data about various sports events. In addition, it serves as a platform for writing and reading online reports from matches.

Currently, there are two primary sports data services on the Czech market - Onlajny.com and Livesport.cz[2].

Onlajny is an older service developed by the company eSports.cz[3] and has not had major visual or technical updates in many years. Livesport was, on the other hand, visually overhauled in the recent past and is growing faster than Onlajny, according to our company's data. Therefore, I want to improve our company system Onlajny, so that we could compete more successfully in the Czech sports data market.

One of the reasons why Livesport is so popular is that it has a mobile application as well as a classic web application. On the contrary, our service does not have a mobile application and native feeling on mobile devices is missing in our web application.

I would like to develop a new web (as a part of my master's thesis) and mobile (out of the scope of my master's thesis) application. The new applications will also require a complete rework of the outdated backend service so that they can receive data according to the modern trends.

In addition, the Onlajny system provides an export service for other businesses which are using our company's data. This backend service is also very old and would need to be redesigned.

This thesis focuses on the revitalization of the Onlajny system. The document describes the current state of the application, the design of the new version, its implementation, and the evaluation of the created solution.

One of the main goals of the thesis is to add support for the semantics of the data. This requirement emerged when we discussed the topic with my supervisor. We settled on introducing the data semantics to the exports and

[1]`https://www.onlajny.com/` [cit. 2023-05-16]
[2]`https://www.livesport.cz/` [cit. 2023-05-16]
[3]`https://esportsmedia.com/` [cit. 2023-05-16]

also to the frontend of the application.

Another goal of the master's thesis is to improve the real-time data communication in the application. Today, communication is done in almost real-time with old technology, which will be changed as a part of the thesis.

A new web frontend would also be developed within the scope of the master's thesis. However, the design of the frontend User Interface (UI) is created by an external designer.

This text first focuses on the analysis (Chapter 2) of the thesis topic. We look at the basic principles and technologies needed to understand the topic. Next, the thesis continues with the design of the newly created system for the Onlajny application (Chapter 3). The text also describes the implementation of the designed solution in Chapter 4. And last but not least, the created solution is evaluated in Chapter 5.

Chapter 2
Analysis

This chapter first discusses the core technologies needed for understanding the thesis topic. Then the chapter describes the current state of the Onlajny application. And last but not least, the text deals with the requirements for the newly developed system.

2.1 Background

First of all, we need to define an overview of some of the main principles and technologies which are relevant to the topic and which are going to be referenced later in the thesis.

2.1.1 REST

The most common thing to develop when building a web application is an Application Programming Interface (API). That is the way our frontend[1] can communicate with our backend[2]. One of the most popular architectures for web application API development is called Representational State Transfer (REST).

It was defined by Roy Fielding in his dissertation thesis in the year 2000 [3]. So, it is more than 22 years old, but still very popular because of its low technological overhead and very easy implementation. The main REST principles are listed below.

- Client-server
- Uniform interface
- Stateless interactions
- Cacheable
- Layered system

[1]Frontend allows users to interact with a software system. From a developer's point of view, the frontend is the design and code that make User Interface (UI) work [1].

[2]Backend denotes any part of a software system which is not interactive and visible to the end user. Common backend processes are those that use a web server to run [2].

- Code on demand (optional)

First, REST API needs to work in a client-server environment, i.e., clients communicate with the server. In addition, it works with a basic unit called a *resource*. A resource is any information that can be transferred by API. Each resource should be uniquely identified, for example, with URL.

When someone wants to access a resource, they need to do so while using the so-called *resource methods*. Roy Fielding does not specify which meaning belongs to which method. It is up to the developer of the API to decide on this problem. However, every method in one API must have a clear and unique meaning.

The communication between the client and the server must be stateless. It means that each request from the client has to contain all information needed to understand the request and must not take advantage of the information stored on the server.

The REST API also needs to support cache to improve network efficiency. If the information is marked as cacheable, there is less need to make requests for the data on the way to the client.

As another principle, Roy Fielding states the so-called layered system. This system is the same as the layered architecture in software engineering. The application is split into services existing in multiple layers, and services from one layer provide the functionality to the layer above and use the functionality of the layer below.

The last of the REST principles is code on demand. This principle is often not implemented and can be omitted. It says that the client can ask for a resource that will return a code that can be executed on the client. This can simplify client development in some cases [3].

In addition, REST uses some basic principles of HTTP communication. Therefore, it is mostly adopted by applications that run on top of HTTP so the protocol's functionality can be utilized in the communication, e.g. the HTTP verbs and status codes.

When building REST API on top of HTTP it is common practice to use GET, POST, PUT, and DELETE methods. Meanings of the particular methods are listed in Table 2.1.

Method	Meaning
GET	retrieve an existing item
POST	create a new item
PUT	replace an existing item
PATCH	modify an existing item
DELETE	remove an existing item

Table 2.1: Most used HTTP methods and their meanings

The HTTP also provides the so-called *status codes*. These codes represent the result of the request processing by the server and can also be used to represent the transfer state. Yet again, it is up to the developer which code will have which meaning. However, it is common practice to use status codes as described in Table 2.2 [4].

Code	Meaning	Usage
200	OK	successful operation with data
201	Created	successful creation
204	No Content	successful modification
404	Not Found	resource not found, for all methods
405	Method Not Allowed	method is not defined for the resource
409	Conflict	operation failed due to data conflict
500	Internal Server Error	operation failed due to server error

Table 2.2: HTTP status codes and their usage

2.1.2 JSON

When we build an API, we need a data format to exchange information between two communication points. Two main formats are most often used in REST for this purpose: the older Extensible Markup Language (XML) standard and the newer and more popular JavaScript Object Notation (JSON) standard.

JSON is data-interchange format based on the JavaScript (JS) programming language standard introduced in 1999. It is a text format that is language-independent. However, it uses conventions very similar to those of the C family of languages.

The popularity of this standard is due to its simplicity. It is easy to read for machines and also for humans, and it is also easily generated. JSON is also less verbose than XML.

Let us look at the example document in Listing 2.1. JSON document is using two main data structures - *objects* and *arrays*. An *object* is a collection of name/value pairs. Such pair can be seen on line 2 in the example. Objects can also be nested, which is depicted on lines 3-6. The other data structure is an *array* which is an ordered list of values (line 7 in the example) [5].

```
1 {
2     "name": "Jaromir Jagr",
3     "team": {
4         "id": 1,
5         "name": "Rytiri Kladno"
6     },
7     "photos": ["photo1.jpg", "photo2.jpg]
8 }
```

Listing 2.1: Example of a JSON document

JSON Schema

JSON is missing a standardized schema or metadata definition that would allow developers to specify the structure of a JSON document. This functionality can be added with the usage of JSON Schema [6].

One can also use JSON-LD to define the document's metadata. However, these data are used for context definition rather than validation, like in the JSON Schema. JSON-LD is discussed in more detail in Section 2.1.5 of this document.

JSON Schema is, as its name suggests, a schema language that allows developers to constrain the structure of a JSON document. Users can therefore define, for example, what properties of an object are mandatory or how many elements are allowed to be in an array at maximum [7].

Example JSON Schema can be seen in Listing 2.2. The key `$schema` describes according to which version of the JSON Schema was the schema written. The type is a validation keyword and constrains the type of the document - in our case, it is an object.

Next, we define the properties of the object. In this part of the schema, the user enumerates all the possible keys of the JSON document and for each of them, the user describes their type, properties, etc. recursively.

And last but not least, there is a `required` keyword. Its value is an array with names of the properties of the document that must be present at all times [8].

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "name": {
          "type": "string"
        },
        "team": {
              "type": "object",
              "properties": {
                "id": {"type": "integer"},
                "name": {"type": "string"}
              },
              "required": ["id", "name"]
        },
        "photos": {
            "type": "array",
            "items": {"type": "string"}
        }
    },
    "required": ["name", "team", "photos"]
}
```

Listing 2.2: Example of a JSON Schema

2.1.3 JavaScript frontend

Today's trend in web development is to divide applications into a backend part and a frontend part. These two parts are typically separated, so they can be developed by different people.

Like in the development of a backend, there are also many new techniques for developing a frontend part of the application. The time of static Hypertext Markup Language (HTML) documents is long gone, and the modern web page must be interactive. The most popular way to achieve this is with the usage of JavaScript (JS).

JS is a lightweight interpreted programming language developed to script web pages. It has support in all modern web browsers, although it can also be used on different platforms than the Web [9].

When we develop a new JS frontend, we have two main approaches: client-side rendering and server-side rendering.

Client-side

JS web applications in the modern world must handle many simultaneous connections, and that brings a large overhead for the server side of the application. This problem can be tackled by moving the rendering of the HTML document to the client and achieving it with the JS code that runs on the client device. This approach is called client-side rendering.

The server typically sends to the client an empty HTML document on the first request alongside the JS code. Then the JS code is run on the client and renders the page's content into the HTML. All changes to the page are then rendered on the client. The server needs to make the data available, e.g., in the JSON format.

This approach, on the other hand, is not ideal for search engines. A search engine is a web-based application that browses different data available over the Internet and displays the results to its users as links to relevant web pages.

Search Engine Optimization (SEO) is a technique that aims to improve the relevance of the developed web page in search engine results. In other words, this means increasing the visibility of a web page by growing its SEO rank. SEO also helps the search engine to show its users more relevant results [11].

When we develop a JS web application with client-side rendering, the search engine needs to execute the shipped JS code in order to see the full page content. And because of that, the website gets a lower SEO rank from the search engine. This is not ideal when we want our page to be shown as early and as often as possible in the search results. This problem is solved by rendering on the server side [10].

Server-side

Server-side rendering is an approach in which the server executes the populating JS code before sending it to the client. Therefore, the client receives not only the JS code itself, but also the HTML document with the rendered page content on the first request.

This is important for search engines and boosts the website's Search Engine Optimization (SEO) rank. The engines crawl the web and parse the received HTML documents, so they need to get the content right away and not wait for the JS code to render the page on the client.

Most server-side frontend frameworks render the page on the server only on the first request. After that, the JS code takes over and renders the content on the client machine. The server-side rendering, therefore, has almost the same performance as the client-side rendering and adds the benefit of supporting search engines [10].

2.1.4 WebSockets

When building an API, one may need to communicate the data in real time. The WebSocket API was built for this purpose. It enables the creation of a two-way interactive session between the client and the server. The messages sent by the server can be based on events that occurred, without the need to ask for them directly [12].

The WebSocket protocol functions on top of the existing HTTP protocol because one of its goals is to use the existing HTTP infrastructure. It is designed to work on standard HTTP ports 80 or 443. It also supports HTTP proxies and other technologies, although it adds some complexity to the currently used environment.

The protocol consists of two parts, a handshake and a data transfer. If the client and the server support the protocol, its usage is determined in the handshake phase. The client sends the request with specific HTTP headers that can be seen in Listing 2.3 on lines 1-7. The server response then includes the header `Sec-WebSocket-Accept` as depicted on lines 9-12 in Listing 2.3. After a successful handshake, the WebSocket connection is established and either of the two sides can independently send the data [13].

```
 1  GET /match/18 HTTP/1.1
 2  Host: onlajny.fake
 3  Upgrade: websocket
 4  Connection: Upgrade
 5  Sec—WebSocket—Key: dGhlIHNhbXBsZSBub25jZQ==
 6  Origin: http://onlajny.fake
 7  Sec—WebSocket—Version: 13
 8
 9  HTTP/1.1 101 Switching Protocols
10  Upgrade: websocket
11  Connection: Upgrade
12  Sec—WebSocket—Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Listing 2.3: Example of a WebSocket handshake

Before the WebSocket protocol, there was another technique to achieve almost real-time data transfer. It is called HTTP polling. HTTP polling is based on a sequence of request-response messages. The client sends a request to a server. After receiving the request, the server responds with a new message. The new message contains the data for the client if there are any. If there are no new data available for the client, the response is empty. After a short time (the polling interval), the client sends another request to the server to see if any new data are available. This is not ideal because many requests have to be sent over the network.

The disadvantage of a large number of request-response exchanges was tackled by a technique called HTTP long polling. The difference is that the server does not send the empty message immediately after realizing that there are no new data for the client. The server rather holds on to the request and sends the response after the data is available. This reduces the number of messages sent. However, it is not a full duplex channel that is offered by the WebSocket protocol, therefore, not optimal [14].

Efficiency is very important in real-time data transfers. Qigang Liu and Xiangyang Sun have conducted a test to monitor WebSocket performance. They compared a WebSocket implementation of asynchronous communication with HTTP polling implementation. Based on their results, WebSocket performance is much better than polling in terms of network traffic and delay, especially when dealing with large concurrency [15].

2.1.5 Semantic web

The term semantic web was first used by Tim Berners-Lee who described it as a web of data and information that can be processed by machines [16]. One can also say that the data must be machine readable. Semantics also provides an account of the meaning of the data. Therefore, machines need to understand the logical connection between particular information found on the Web and what they represent.

The definitions of the categories, properties and relations between data and information are called *ontologies*. An *ontology* formally describes an entity or a relation between two entities and because of its formality can easily organize data and therefore reduce the complexity of operations performed with those data [18].

To model an ontology on the Web, Web Ontology Language (OWL) was developed. An OWL document consists of optional ontology headers and any number of axioms and facts. Ontologies modeled in OWL are serialized to the RDF standard [19].

Another language that supports ontologies on the Web is called the RDF Schema. It is an extension of the basic RDF vocabulary which is described later in this text in Section 2.1.5. The RDF Schema is not as expressive as OWL, however, it also contains mechanisms to describe groups of related resources (classes) and their relationships [20].

RDF

For encoding the data, the Resource Description Framework (RDF) is used. It is based on statements about resources that are interconnected, and thus can be viewed as a directed graph. Every RDF triple consists of a subject, a predicate, and an object [17].

All three main RDF parts can be seen in the example figure 2.1. There are three RDF triples in the graph. They have the same subject, which is depicted by the rounded node. The arcs in the graph illustrate the predicates, and the rectangles are the objects.

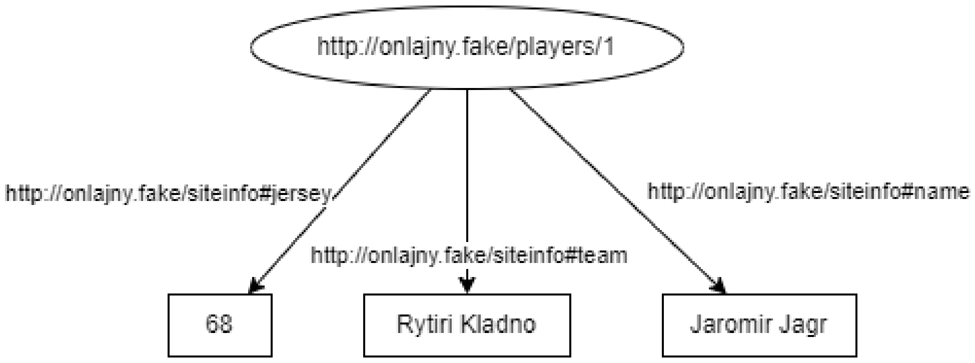
http://onlajny.fake/players/1
http://onlajny.fake/siteinfo#jersey
http://onlajny.fake/siteinfo#team
http://onlajny.fake/siteinfo#name
68
Rytiri Kladno
Jaromir Jagr

Figure 2.1: RDF graph example

The same example can be also written in Terse RDF Triple Language (Turtle) as can be seen in Listing 2.4. On line 5 we define the subject, i.e. we are talking about a player defined by Internationalized Resource Identifier (IRI) given by concatenating the prefix `p` and the ID 1.

Next, on lines 6, 7, and 8, we define the predicates and objects. For example, on line 6, the predicate is `si:full-name`, which corresponds to the IRI `http://onlajny.fake/siteinfo#name` and the object is the string `Jaromir Jagr`.

```
1  @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
2  @prefix si: <http://onlajny.fake/siteinfo#> .
3  @prefix p: <http://onlajny.fake/players/> .
4
5  p:1
6      si:name "Jaromir Jagr" ;
7      si:team "Rytiri Kladno" ;
8      si:jersey "68" .
```

Listing 2.4: Example of RDF in Turtle

JSON-LD

RDF statements can be written in many different syntax notations. We have already seen the Turtle notation in Section 2.1.5. However, in the application Onlajny, we want to mainly work with the JSON format. Therefore, we will take a look at the JavaScript Object Notation for Linking Data (JSON-LD) format.

JSON-LD offers the tools needed to work with RDF and encapsulates them into the JSON data format. One of the main features is the `@context` keyword. It is a tool with which users can define the context for the data in the JSON document.

Users can make references to custom data types that are not available in the JSON format by default. JSON-LD also offers the option to support multilanguage fields. A JSON document can have, for example, a field named `name` that should be provided in multiple languages. Users can achieve this with an object containing key/value pairs, where keys are the language tags, and values are the translations. However, the information that the field contains such an object should be included in the context.

We also need to reference a node in the RDF graph. To do so, we need to give a unique identifier to the nodes. JSON-LD has a keyword `@id` for this purpose. Furthermore, we have a keyword `@type` at our disposal. This keyword allows us to define the type of JSON object which we could not have done otherwise. In addition to that, it uses plain JSON structures, which can be seen in Listing 2.5 [21].

```
 1  {
 2      "@context": {
 3          "name": "http://onlajny.fake/siteinfo#name",
 4          "team": "http://onlajny.fake/siteinfo#team",
 5          "jersey": "http://onlajny.fake/siteinfo#jersey"
 6      },
 7      "@id": "http://onlajny.fake/players/1",
 8      "@type": "http://onlajny.fake/siteinfo#player",
 9      "name": "Jaromir Jagr",
10      "team": "Rytiri Kladno",
11      "jersey": 68
12  }
```

Listing 2.5: Example of a JSON-LD

Listing 2.5 contains the Turtle example from Listing 2.4 rewritten to JSON-LD. One can observe the key `@context` in which the user maps the short terms (JSON keys) to the definition IRIs.

The subject itself is defined as the root JSON document with the `@id` key on line 7. The predicates are defined by the keys of the document on lines 8-11, and the objects are defined as the values on those lines. With the keyword `@type`, we can also define which type the subject is. The type is determined by IRI.

Linked data

When we apply the relations between the individual data entities and collections of data entities, we get something called linked data. Linked data aims to connect data through data-level links from different sources into a single global data space.

Linked data are primarily based on open web standards. This means that data consumers can use generic tools to access the data and work with them. Additionally, search engines can read the data more efficiently and process it to provide better services to their users [22].

2.2 Current state

Today, there are three parts of the Onlajny system as depicted in Figure 2.2. First of all, there is the administration system for writing the online articles, which is well tested and functional, and we plan to keep it as is in the new version. Then there is a web application rendered on the server side for data presentation available to all users. And lastly, there is an API used by other companies that pay for our generated data.

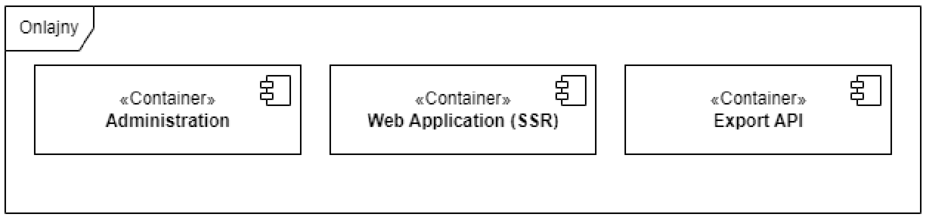

Figure 2.2: Current parts of the Onlajny system

The data pieces are stored in different places. The primary data generated by the authors are stored in the MySQL databases on our server. There are now two main databases. The first one contains the live data, which are expected to change. The other one contains the archived data. Such data are not changeable and are stored in a separate Database (DB) so that the one with the live data has a smaller workload.

Those two databases are connected via daily cron job, which takes all the information about matches older than 48 hours, copies them to the archive, and deletes them from the live DB. This operation is managed by the database provider, and the developer just needs to make sure that the data models of the two databases are the same.

Moreover, for some sports, we also have additional data generated by our other systems. These data are available in several Amazon Web Services (AWS) buckets and can be accessed via JSON API.

The export API now provides very basic endpoints about events and their matches, and also individual match statistics. Data are distributed in XML format, which has a lot of boilerplate code. Thus, it is not as efficient for simple data exchanges as JSON [23].

The system also has the administration of the export API which is currently done in many different places and, therefore, not very convenient. A new centralized administration should be developed as part of the revitalization of the Onlajny system. However, this application is outside the scope of this thesis.

Last but not least, the main part of the system is the web application available to regular users. This web application is now written in PHP and is being rendered on the server.

Nevertheless, the Onlajny system provides live data, so we cannot rely on static generation of the pages. The data shown must be redrawn on the client when they change in the database. This is now done via HTTP long polling executed by a JS code shipped with the page rendered on the server.

The JS code asks the server for any changes in the data shown on the current page and rerenders affected parts of the page accordingly. However, this approach is not ideal, as discussed in Section 2.1.4.

The current state of the Onlajny system is also in decent technical debt. The term technical debt is a designation for the consequences of poor software development that can lead to many obstacles in the future maintenance and growth of the created system [24].

In particular, the system lacks any automated tests or continuous integration features. All the testing is done manually by the product owners, which is not sufficient in a project of this scale.

The system also has no project documentation. All the requirements are now being put to a task manager, but they are not documented in a concise manner. Thus, it is very hard to navigate in the system and its features without consultations with the senior developer of the application.

Furthermore, a developer documentation is missing, too. When browsing the current codebase, one cannot understand the functionality very easily. It is because the code is mostly not self-explanatory and also does not contain explaining comments or even used data types.

All of those difficulties lead to the fact that no one except the main senior developer wants to collaborate on this system. This is a common consequence of technical debt, as it causes slower development and kills the productivity of the team [24].

It also means that the company and the system are completely dependent on the one developer who knows all parts of the system. This is an issue similar to the vendor lock-in problem in cloud computing [25] and should be addressed by the executive officers.

2.3 Requirements

The goal of the revitalization project is to rewrite the Onlajny system according to the current trends in web development. The new application should be modern, scalable, and address the problems that stem from technical debt.

2.3.1 Frontend

As far as the frontend goes, we would like to add client-side rendering to the web application so that the server is less burdened. However, one of the crucial aspects of the web is SEO which requires server-side rendering. So, the best approach would be hybrid rendering, when the server renders the page only on the first request, and then the client takes over the rendering process.

In addition, Google started to use JSON-LD in its search engine. And its usage can help to improve the SEO. To take advantage of this feature, the web page must contain a `script` tag with a type of `application/ld+json` inside of a `head` HTML tag [26].

An example can be seen in Listing 2.6. There is a script tag with the proper type and its content is a JSON-LD document. Google recommends using the schema.org vocabulary, as they are involved in its maintenance. Also, Google Search provides its own rules for structured data such as required, recommended, or optional properties on particular types [26].

```
<script type="application/ld+json">
    {
      "@context": "https://schema.org/",
      "@type": "Game",
      "name": "Los Angeles Kings — St. Louis Blues",
      "author": {
        "@type": "Person",
        "name": "Jan Novak"
      },
      "datePublished": "2012—05—10",
      "description": "A game of NHL playoffs between Los Angeles Kings and St. Louis Blues."
    }
</script>
```

Listing 2.6: Example of a JSON-LD usage for Google SEO

The Onlajny web frontend also needs to work with live data. The client application must reflect any changes that occur in the database. This means that we need to manage real-time communication between the frontend application and the backend server. To optimize this dual communication, it would be best to use WebSockets as described in Section 2.1.4.

In addition, the new web frontend should have all the functionality of the old one. It is also expected that more functionality will be added in the future, such as user accounts and personalizing of the content. However, those features are beyond the scope of this thesis.

The project also includes the development of a mobile application that needs to run on Android and iOS devices. Therefore, the preferred way to achieve this is to implement the application with some cross-platform technology. The mobile application is out of the scope of this master's thesis. However, in addition to the aforementioned frontend, the mobile application would also need to consume data from somewhere, so we need to prepare a backend service.

2.3.2 Backend

The backend of the new Onlajny system can be split into two main services - the export API and an internal API for web frontend and mobile application.

The export service must have backward compatibility due to the current API endpoints that our customers consume. These endpoints must also remain in their current form, which means that we need to enable the option of using XML as the data format.

We also need to create the frontend API with new endpoints for the new web application and the mobile application. These endpoints should support nowadays favorite JSON data format and should be built according to the REST principles.

In addition, when we discussed the topic with my supervisor, we came up with an idea to enrich the API endpoints with semantic data so that the data would be machine-readable.

For the purpose of machine readability of API, I chose to use the JSON-LD format. Users can specify the type of content they want to receive, while `application/ld+json` is one of the available. The returned JSON-LD document contains a link to the external context that contains JSON-LD definitions of the types and properties used and can be retrieved if needed.

Listing 2.7 shows a simplified example of a JSON-LD document returned by the new Onlajny export API. Here, we want to get information about a detail of a match, so we get back a JSON object with the `@context` property set to an IRI describing the `MatchDetail` type.

```
{
    "@context": "http://onlajny.fake/api/v1/schema/types/MatchDetail",
    "id": 299126,
    // rest of the properties
}
```

Listing 2.7: Example of a JSON-LD API

2.3.3 Functional requirements

The functional requirements are mainly the same as in the old Onlajny system. The list of the main ones is given below.

- FR1 - The system must allow users to list played, live, and upcoming games
- FR2 - The system must allow users to filter games by date
- FR3 - The system must allow users to filter games by sport
- FR4 - The system must allow users to filter games by league
- FR5 - The system must allow users to find player rosters in a particular match
- FR6 - The system must allow users to find scored goals in a particular match
- FR7 - The system must allow users to find given penalties in a particular match

2.3.4 Nonfunctional requirements

The list of the main nonfunctional requirements of the system is given below.

- NR1 - The system shall be split into backend and frontend parts
- NR2 - The system shall adopt REST principles in API
- NR3 - The system shall support JSON data format in its APIs

- NR4 - The system shall have backward compatibility of the export API
- NR5 - The system shall support machine-readable data
- NR6 - The system shall have clearer URLs to improve SEO
- NR7 - The system shall have clearer meta information to improve SEO
- NR8 - The system shall use WebSockets to optimize live-data transfer

Chapter 3

Design

In this chapter, the design of the developed system is discussed. First, the overall picture of the system is depicted. Then the individual parts of the system are drawn and explained.

3.1 Overview

The overview of the system is shown in Figure 3.1. The server side of the application is divided into two parts, the export service and the front server.

The export service provides the export API. For that, it needs to read the data from the two main sources which are AWS and internal MySQL database.

The exposed API is open to customers consuming our data exports. When making the request, users can specify the desired format in which an export should be returned. Available formats are JSON, XML, and JSON-LD.

The exports are currently managed in the Onlajny administration application which stores the configuration in the MySQL database alongside the other data.

This could be improved by creating a new administration system just for exports management. The system would allow the configuration of consumers and available data. In addition, an API would be provided from which the export configuration data could be retrieved. However, such application is out of scope of this thesis, so it is visualized as a node with a regular border in the diagram.

The other part of the server provides an API marked as internal in the diagram. It is meant to be consumed by the newly created frontends, the web JS application and the mobile application. The JSON format was chosen for this API.

To improve the SEO of the web frontend, the web client application also needs to initially be rendered on the server side. This functionality is also provided by the front-server component, as depicted in the diagram.

In the first iteration of the system design, there was only one server which included export functionality as well as internal API for the frontend. However, after a discussion, I decided to split the server into two separate server applications due to the expected load.

I was concerned that one server would not be sufficient to manage the rendering of the web client, requests to the internal API, and the requests to the export API. Because of that, I decided to move the export API to its own server application.

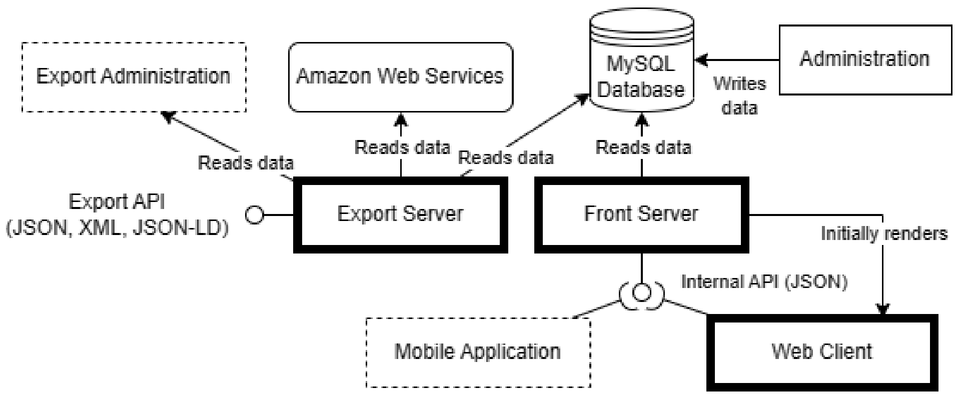

Figure 3.1: System overview diagram

The boldly visualized components depicted in the diagram in Figure 3.1 are being developed as part of this master's thesis. It means the *Export Server*, the *Front Server*, and the *Web Client*.

The other parts of the system shown in the picture either already exist, which is depicted by rectangle nodes with a solid regular border, or are put out of the scope of this thesis. This is depicted by rectangle nodes with a dashed regular border.

The rounded nodes in the diagram have the meaning of external services used by the main system. The cylinder node represents a data source, and the provided APIs are shown with the Unified Modeling Language (UML) lollipop notation.

3.2 Data

The Onlajny system has two main data sources - the internal MySQL database and the AWS buckets.

In the AWS buckets, external statistical data can be found generated by other systems created by the company eSports.cz. However, those data vary from sport to sport and also from league to league. Thus, for the sake of brevity, their model is not shown in this document.

The other data source is the MySQL database, which is divided into two parts. One database manages live data, which are expected to change. The other one is managing the archived data which are not changeable and would slow down the operations on the live database. Nevertheless, the data models of both parts of the database are the same.

The basic data model of the Onlajny system is very simple as it consists of six entities. These entities are depicted in the Entity Relationship (ER) diagram in Figure 3.2.

The core entity is an *Event*, which represents the messages written in the online article through administration. Events must be created with a *Match* assigned to them. When the user opens a detail of a match in the current system, all its events are listed.

Every event also has an enumerated type that helps users understand what happened, so it gives a bit of semantics to the events. The event can, for example, be of type goal, which means that a team scored a goal. This type of event has a *Team*, which scored the goal, and a *Player*, who scored the goal, assigned.

Players can be assigned to teams if they play a team sport. If they play an individual sport, e.g. tennis, they can be assigned to matches directly.

Matches are also grouped into leagues. *League* can have many matches, and users can find detailed statistics about teams and players on their detail pages in the current system.

Furthermore, leagues are grouped by sports, so that users can look only at sports in which they are interested. *Sport* can be, for example, ice hockey, football, or handball.

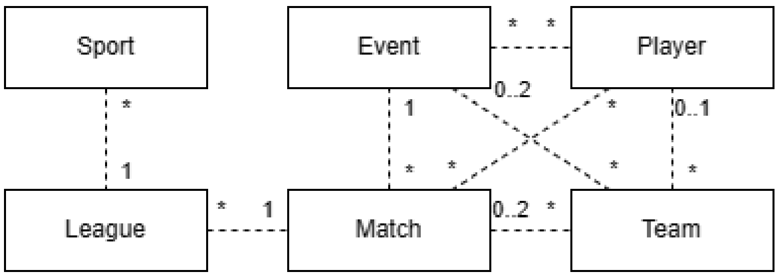

Figure 3.2: ER data overview diagram

In addition, some entities such as sports, leagues, or teams support versioning and translation. It means that they have an extra entity for each of their versions or language mutations. For example, a team exists in the Czech and Slovak language, and so has entities for both of those languages.

However, this thesis focuses only on creating the new system for Onlajny.com which works in the Czech language, so the additional entities are not included in this chapter.

3.3 Backend

The new system is built according to trends in modern web development. Therefore, its backend and frontend parts are divided into separate applications. In addition, the backend part is further split into two separate applications.

The first part of the backend is the *Export Server*. This application provides an API for customers that consume data exports from the Onlajny system. The components of this part of the backend are depicted in Figure 3.3.

The core component of the export application is the *API Service*. This service runs on the HTTP server that handles all requests from customers that want the export data.

All REST endpoints in this service use the two available data sources through the *AWS Data Service* and the *Internal Data Service*. The first service is calling the AWS, the second one is communicating with the internal MySQL database.

The data pieces retrieved from the data sources are then passed to the *Data Transformer Service*. This component provides the functionality of transforming the data to different data formats according to the headers received from the HTTP request.

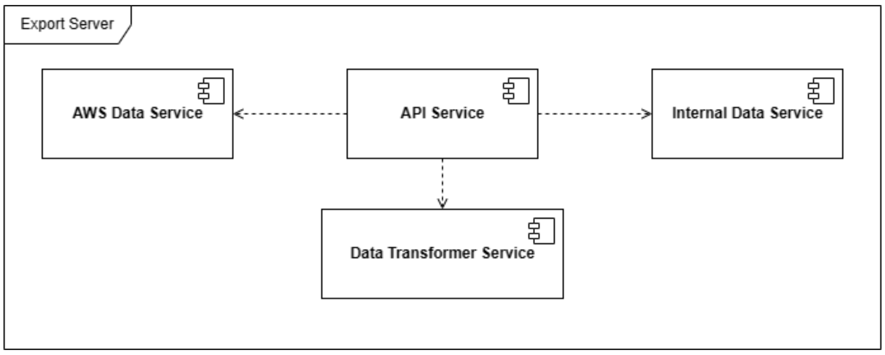

Figure 3.3: Export Server components diagram

The *Front Server* is the second part of the backend. This service is responsible for the server side functionality of the web application available for end users. The figure 3.4 shows all the components of this part of the system.

This backend part has two main services. The *Backend Service* is responsible for the initial rendering of the JS web client. This is important for the SEO of the application.

The *API Service* is exposing endpoints for the JS web frontend. This API provides the data that are displayed on the page. Some endpoints also support WebSocket connection due to the real-time communication requirement.

Both discussed services are using the *Internal Data Service* which retrieves data from the internal MySQL database.

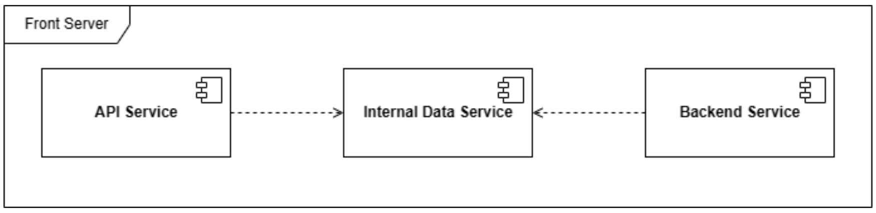

Figure 3.4: Front Server components diagram

3.4 Frontend

Two new frontends will be created in the Onlajny revitalization project. The JS web application and a mobile application. However, the mobile application is beyond the scope of this thesis.

The components of the JS web application are depicted in Figure 3.5. There are two components that represent the two main pages of the website which can be opened in the browser by users: *Homepage* page and *Match Detail* page.

The homepage is the base page of the application and is described by the root URL `/`. It contains a list of games that can be filtered by date, league or sport.

Another important page is a match detail. All the match data are shown within this page, e.g. rosters, comments, score, goal scorers, etc.

In addition, pages are expected to be added with details for individual leagues, teams, and players. However, these parts of the frontend are put outside the scope of this thesis and, therefore, have dashed borders in the diagram.

All pages use *HTTP API Client*. This component communicates with the backend API and contains the functionality for data retrieval.

Another part of the frontend is a *WebSocket Client* which is communicating with the WebSocket servers exposed via backend API endpoints.

The important part of the web application is also a *State Manager*. This component is needed because we need to have a central place to encapsulate the application's state and the logic associated with it. It is good practice to extract this code into a separate component and reuse it when necessary.

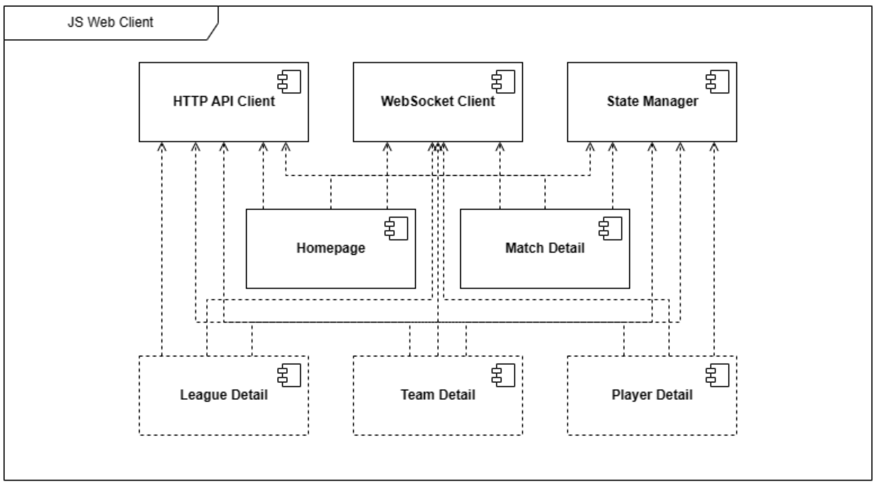

Figure 3.5: JS Web Client components diagram

Chapter 4
Implementation

This part of the document focuses on the implementation of the designed new version of the Onlajny system. This chapter describes languages and libraries used during development. In addition, interesting design patterns and parts of the code are noted in this section.

4.1 Platform comparison

Before the implementation started, the proper platform for its development and runtime had to be selected. The two main options available were Hypertext Processor (PHP) and the most popular JS runtime Node.js.

PHP is vastly used in our company and on the Web, so it had to be considered. However, it provides a synchronous execution with multi-threaded blocking input/output processing. With every request, a new thread is created, which runs the script and returns a result. This is not great for applications that need to stream live data to many clients, for example, with WebSockets [27].

On the other hand, the JS programming language and its Node.js runtime are great for big application load and live data transfer. JS is also very good for client-side rendering and the development of a frontend web application, so the backend and frontend code would be in the same language. I have experience in JS development, too. Thus, I chose to implement the new version of the Onlajny system in JS.

After choosing JavaScript as the platform for the implementation, the decision of the appropriate framework had to be made.

When developing an application of such size, the framework is a must. If we write the entire application in plain JS or any other programming language, we would simply reinvent the wheel.

Plenty of functionality needed for web development, such as networking, file system handling, routing, rendering, etc., have already been implemented in many frameworks which are widely used in production and well tested. On the other hand, our code could never be as bulletproof as the code of the framework. It is thus better to choose one of them and use its functionality then to implement the needed features from scratch.

There are many JS frameworks and it is not easy to choose one. New frameworks are also constantly being created. However, the common and most used choices would be, for example, React, Angular, Vue, or Svelte.

Angular is backed by Google, but has a reputation for being not very user-friendly and is therefore not popular. In the year 2022, Angular had only 47.2% retention according to the State of JavaScript survey [28]. This is not very good, as Vue had almost the same usage percentage as Angular, but had about 80% retention. One of the most popular frameworks is Svelte, which can be seen in its retention of 89.7%. However, it is a young framework and is not yet popular in production, as its usage was 21.2% in 2022. So, only one framework from the earlier offer remained: React, whose usage and retention were above 80% in 2022 [28].

4.2 Node.js

When creating a server application in JS, developers need to choose the runtime by which it will be executed.

Whereas the main runtime for the frontend JS code is the web browser, the backend code cannot run there. For example, the server application needs to work with the file system, networking, and other hardware components. And because of that we need a server runtime as browsers cannot do that [30].

There are many JS runtimes which run on top of the Google's open-source scripting engine V8. Node.js, Deno, or Bun, just to name a few [31].

Node.js[1] was selected as the application runtime for the new Onlajny system. It is an asynchronous event-driven JavaScript runtime designed to handle many concurrent connections. Thus, it is very good for building scalable network applications.

In other systems, there is typically a blocking call to start the event loop. The user first needs to define callbacks describing the application behavior, then a server is started like, for example, `Server::run()`. In Node.js, there is no such call. It simply enters the event loop after executing the input script and exits when there are no more callbacks waiting. Like in the browser JavaScript, the event loop is hidden from the user [29].

Node.js also comes with a very popular npm package manager[2]. It is a tool that manages the dependencies of the Node.js project so that developers do not need to manually download the additional libraries they want to include in their tech stack [32].

Npm also helps with the management of the project itself and its structure with workspaces, which is further discussed in Section 4.6.

[1]`https://nodejs.dev/en/` [cit. 2023-05-16]
[2]`https://www.npmjs.com/` [cit. 2023-05-16]

4.3 React

React[3] is a very popular JS library for building User Interface (UI). It is used extensively in production when it comes to web development because of its ease of use and versatility.

The UIs in React are made up of components. These components are single units that can manage their own state and be composed to make complex UIs. The React code is also declarative, which makes the code more understandable and easier to debug.

React-built applications can be rendered both in browsers and on the server using Node.js. React code can also power mobile apps using React Native [33].

4.4 Next.js

However, React is only a basic UI library that does not contain all the components needed to build an entire web frontend. Developers need to add more tools and libraries to the tech stack in order to fulfill all the requirements. The most common tasks to solve are, for example, routing, data fetching, or state management.

To address the issue of missing tools in React, the full-stack framework Next.js[4] was created to facilitate the start of the application's development. It is widely used in production and provides tools for server-side rendering, TypeScript support, API routes building, and more [34].

Recently, Next.js introduced version 13 with new features improving server-side rendering and SEO. The components of the application's UI are now located in the `app/` directory where all components are server-rendered by default. This reduces the size of JavaScript code sent to the client and enables faster initial page loads.

In addition, the new Next.js supports layouts. The layout component is common for more routes and allows for sharing UI between them without expensive re-renderings of the whole page.

And last but not least, Next.js 13 provides a new `fetch` API which extends the native JS `fetch` API. It allows developers to fetch data at the component level while caching the same requests to preserve high performance [35].

4.5 TypeScript

The new Onlajny system is a big project, and its codebase, while already very vast, can grow quickly even more. Thus, the implementation in a loosely typed language such as JavaScript could be problematic.

In loosely typed languages, variables are not required to be declared with a specific data type. This can lead to problems such as type confusion, which

[3]`https://react.dev/` [cit. 2023-05-16]
[4]`https://nextjs.org/` [cit. 2023-05-16]

can produce errors that are difficult to find and detect. And so developers need to write extensive tests to ensure that the application is working correctly [36].

However, in strongly typed languages, developers must declare variables with a specific data type. This means that type errors can be caught at build-time. And because of that, strongly typed languages are more reliable and easier to debug than loosely typed languages. Maintenance of projects in strongly typed languages is also easier as types of variables often help developers understand what code is meant to do [37].

Fortunately, there is a syntactic superset of JS called TypeScript (TS)[5] which makes it possible to code in a strongly typed language while using the JS ecosystem.

TypeScript is a strongly typed programming language that is based on JavaScript and gives developers a better tool for writing code. It adds additional syntax to the JavaScript code to support tighter integration with the code editors. Users can, thanks to that, catch errors very early in the development cycle.

Users can describe the shape of objects and functions in their TypeScript code. This makes it possible to see documentation and issues directly in the code editor.

Code written in TypeScript needs to be converted to JavaScript before it is run. It means that it can run anywhere where JavaScript runs - browsers and other JS runtimes [38].

4.6 Monorepo

It was decided that the front and the export modules of the new Onlajny system will be separated into two standalone applications. However, they share the model, and so they need to share a part of a codebase.

Traditionally, this situation would be solved using a polyrepo approach. This means that every application would have its own repository, and the shared modules would also have their repository. The development process then becomes very tedious.

The alternative to the polyrepo is a monorepo setup. In a monorepo, the shared modules and the applications are located in the same repository, so no additional steps are required when building and deploying the finished product.

The main block of the monorepo is a workspace. Every application and package is located in its own workspace. However, workspaces can depend on each other, and the monorepo manager will automatically resolve those dependencies. Each monorepo also has a root workspace in which users can define things that are common for the entire codebase [39].

In this project, I chose to use Turborepo[6] as a monorepo manager. It is a build system written in Rust and optimized for JavaScript and TypeScript

[5]`https://www.typescriptlang.org/` [cit. 2023-05-16]
[6]`https://turbo.build/repo` [cit. 2023-05-16]

codebases. It also uses caching to speed up local development and deployment.

I am using Turborepo with npm, and in this case, the root workspace is located in the root folder of the project with its `package.json` file. The individual applications and packages are then located in their npm workspaces that correspond to the monorepo workspaces and are configured in their own `package.json` files. [40]

4.7 WebSockets

When I was choosing a library to implement real-time communication via WebSockets, I immediately stumbled upon Socket.IO[7]. It is a very popular library with features such as rooms, namespaces, and sessions with Node.js support [41].

However, when I was browsing the documentation of this library, I discovered something strange. While it uses the WebSocket protocol under the hood, it adds additional information to the protocol, creating size and speed overhead, and sometimes the library falls back to the long polling [42].

I didn't like that, so I tried to implement my solution with plain WebSocket using the `ws`[8] module from npm. However, I was unable to get this library to work with the Next.js framework I was using. So after a long time, I returned back to the Socket.IO library.

I also found that Socket.IO was used by Trello in their massive real-time application. And so I comforted myself with the fact that the Onlajny system is not expected to grow as much, and thus the Socket.IO library should be sufficient [43].

4.8 Express.js

In addition to the frontend application, the Onlajny system contains also an export service. I wanted to implement this service as light as possible.

I already had a Next.js framework in my tech stack running the frontend web application. While it could certainly be also used to build the export API, I figured that it has too much functionality that I would not use. In other words, I wanted something simpler for the implementation of the exports.

Thus, I chose the simplest and most popular JS web framework which is also used by Next.js under the hood, Express.js[9].

Express.js is a minimalist web framework for Node.js with simple, yet powerful routing API. Developers can create an app instance and then add handlers for web requests with different HTTP methods and different URL paths.

Another great feature of Express.js is the middleware. Middleware is a function that can be added to particular routes and can then read and

[7]`https://socket.io/` [cit. 2023-05-16]
[8]`https://www.npmjs.com/package/ws` [cit. 2023-05-13]
[9]`http://expressjs.com/` [cit. 2023-05-16]

modify requests going to the route. So developers can greatly customize the request processing pipeline, which is useful for tasks like user authentication, authorization, or custom error handling [44].

4.9 Code structure

The code structure of the Onlajny revitalization project is heavily influenced by the Turborepo recommendations for organizing a monorepo. The main parts of the root repository are listed below.

- *apps* - code of all launch-able applications
  - *export* - code for the export API server
  - *front* - code for the frontend server and the JS web client
- *packages* - reusable packages included in developed applications or other packages
  - *logger* - code for logging
  - *onlajny-model* - code encapsulating the communication with the database and transforming data
- *.env.template* - template for the configuration `.env` file
- *package.json* - management of the global dependencies and workspaces as the project uses npm
- *turbo.json* - configuration of the Turborepo

Every application and package has its own project structure, and so I described them in the following lists. The first depicts the export server application, although it has a common JS project structure.

- *cypress* - API tests written in Cypress, has its own project structure
- *src* - source code of the export application
  - *api* - definition of the export API endpoints
  - *middleware* - code for the Express.js middleware
  - *model* - definition of the export data model
  - *index.ts* - entry point of the export application
- *cypress.config.ts* - Cypress configuration
- *package.json* - management of the export dependencies
- *tsconfig.json* - TypeScript configuration

The next application is the server for frontend which is written in Next.js. This had an impact on the code structure, so it is a bit different from the one for the export. It is shown below.

- *app* - routes written in the new Next.js 13 App Router[10]
- *cypress* - UI tests written in Cypress, has its own project structure
- *pages* - routes written in the old Next.js Pages Router[11], used for the internal API because of the incomplete Next.js 13 support
- *public* - contains static files to be served
- *styles* - global styling of the frontend web application
- *types* - custom TS types needed for the application development
- *cypress.config.ts* - Cypress configuration
- *next.config.js* - Next.js configuration
- *tailwind.config.js* - Tailwind configuration as it is used as an application styling framework
- *package.json* - management of the front dependencies
- *tsconfig.json* - TypeScript configuration

Furthermore, the main package, which is not launch-able, is the model shared between both developed applications. Its project structure is very simple, but is listed below for the sake of completeness.

- *src* - source code of the package
  - *__tests__* - Jest test files
  - *db* - code responsible for the database communication, also contains database model definition for ts-sql-query
  - *model* - definition of the data model operations
  - *types* - additional types and data transformation helpers
  - *cache.ts* - contains data caching logic
  - *index.ts* - exports of the public parts of the package
- *package.json* - management of the onlajny-model dependencies
- *tsconfig.json* - TypeScript configuration

[10]`https://nextjs.org/docs/app` [cit. 2023-05-19]
[11]`https://nextjs.org/docs/pages` [cit. 2023-05-19]

4.10 Interesting patterns

In this section, the design patterns used in the implementation are introduced. Not all patterns are described for the sake of brevity. However, the most interesting ones are included.

4.10.1 Template method

The first design pattern implemented in the project is the template method. This pattern lets developers define a skeleton of a particular algorithm in the parent class. The classes that extend this skeleton class can then override or implement particular steps of the algorithm [45].

An example of a usage in the model package is shown in Listing 4.1. I defined an abstract class for a data transformer which is responsible for sending export data in a proper format. The desired data format is determined from the received HTTP accept header on lines 9-24.

The abstract class has a transform method for each supported format. These are defined on lines 2-4 and need to be implemented in the subclasses. The parent class uses the methods and their implementation at runtime to properly transform the export data.

```
 1 export abstract class DataTransformer<T, U> {
 2   abstract getXml(data: T, params?: U): string;
 3   abstract getJsonLd(data: T, params?: U): string;
 4   abstract getJson(data: T, params?: U): string;
 5
 6   sendTransformedData(req: Request, res: Response, data: T, params?: U): Response {
 7     let type: string;
 8     let toSend: string;
 9     switch (req.headers.accept) {
10       case AcceptHeaderType.Xml:
11         type = AcceptHeaderType.Xml;
12         toSend = this.getXml(data, params);
13         break;
14
15       case AcceptHeaderType.JsonLd:
16         type = AcceptHeaderType.JsonLd;
17         toSend = this.getJsonLd(data, params);
18         break;
19
20       default:
21         type = AcceptHeaderType.Json;
22         toSend = this.getJson(data, params);
23         break;
24     }
25     return res.set("Content-Type", type).send(toSend);
26   }
27 }
```

Listing 4.1: Example of template method

4.10.2 Singleton

Another used design pattern is a singleton. Singleton belongs to a group of creational patterns. It gives developers the ability to ensure that only one instance of a class can exist. Due to this, global access to this instance can be granted [46].

Traditionally, a static method, e.g., `getInstance`, would be created in the class, and a class would have a private constructor. The function `getInstance` would return a static private instance of the class or create a new instance if it does not already exist [46].

However, when using the new JS import/export syntax, developers can implement this a little easier. An example can be seen in Listing 4.2. A DB connection pool is created at line 1. This instance is created only once in the run of the application. The export statement on line 10 then exposes a global function that provides an access to the single `pool` instance via more `DBConnection` instances (requirement of the ts-sql-query).

Singleton instances can be difficult to test as many test frameworks use inheritance to create mock objects, and singletons have private constructors [46]. However, in my implementation, the singleton instance is wrapped in a getter function which can be easily mocked to return a DB connection which does not use the production database. The mocking process is discussed in Section 5.1.1.

```
 1  const pool = createPool({
 2    host: process.env.DB_HOST,
 3    user: process.env.DB_USER,
 4    password: process.env.DB_PASSWORD,
 5    database: process.env.DB_NAME,
 6    connectionLimit: 100,
 7    socketTimeout: 1000,
 8  });
 9
10  export const getConnection = () =>
11    new DBConnection(new MariaDBPoolQueryRunner(pool));
```

Listing 4.2: Example of singleton

4.10.3 Builder

Last but not least, a builder pattern was utilized during the implementation. This was caused by the usage of the ts-sql-query which is managing the DB query creation. Builder lets developers create complex objects step by step. This is done by chaining function calls while every call adds a piece to the created object [47].

This approach is used in the ts-sql-query as depicted in Listing 4.3. A simplified version of a query to get all comments from a match is given for the sake of brevity.

The database connection is retrieved on line 4. Then the query is created incrementally by chaining the calls of the utility functions on lines 5-14.

Finally, the `executeSelectMany` function is called on line 15 which generates the query and executes it while returning the data.

```
1  export const findByMatch = async (id: number) => {
2    const tComment = new CommentsTable();
3    const tFlag = new FlagsTable();
4    const match = await getConnection()
5      .selectFrom(tComment)
6      .innerJoin(tFlag)
7      .on(tComment.idFlag.equals(tFlag.id))
8      .where(tComment.idMatch.equals(id))
9      .and(tComment.deleted.equals(0))
10     .select({
11       id: tComment.id,
12       label: tFlag.long,
13     })
14     .orderBy(tComment.order, "desc")
15     .executeSelectMany();
16
17   return match;
18 };
```

Listing 4.3: Example of builder

Chapter 5
Evaluation

After the development of the new version of the Onlajny system, we needed to evaluate the results. Testing of the individual parts of the system is discussed in this chapter as well as the User Acceptance Testing (UAT) of the frontend web application.

5.1 Testing

The old Onlajny system is very poorly tested. The system currently lacks unit tests, integration tests, and automatic UI tests. Everything is being tested by hand and often in production. This contributes to the technical debt (2.2) of the project and should be changed.

As part of this thesis, the new testing methods for the Onlajny system are designed and suggested. Nevertheless, testing is not the main goal of this thesis, so tests are not its crucial part. Thus, they are not that extensive.

5.1.1 Model

The database communication and the business logic that works with the data are separated in a special package. This model package builds the SQL queries, executes them and transforms the retrieved data into a proper form. Such package should be tested because it is used by the export service and by the frontend service.

Unit tests are a clear solution to this problem as they focus on the individual components of the system. Components are also called units and are the basis for unit testing. The testers define the inputs for each of the units tested and check the outputs given regardless of the rest of the system [48].

There are some functions in the model package that modify the data received from the DB. These functions were the main targets of the unit tests implemented.

In addition to that, integration tests are also good practice. Integration testing combines individual units of software into working modules. Then those modules are tested, rather than individual statements as in unit testing. The main goal of integration tests is to validate interactions between modules [49].

In this case, the model integration tests cover the communication with DB. Tests can check whether proper driver functions were called with the correct arguments.

I chose Jest[1] for the integration tests and it makes it very easy to implement them. Especially in combination with the ts-sql-query[2] library which was used to build the data queries.

An example implementation can be seen in Listing 5.1. The developer can mock functions using the `jest.fn` utility which is used on lines 1 and 11.

In addition, whole modules or their parts can be mocked using the function `jest.mock`. Lines 5-15 show how one would change a single function, in this case `getConnection`, exported from a module `db`.

Moreover, the used query builder library provides a `MockQueryRunner` which can accept a mocked function as a query executor. Therefore, it is very easy to check whether the query executor is called with proper parameters, as one would just ask the `queryExecutor` defined on line 1.

```
 1  const queryExecutor = jest.fn(() => {
 2    return null;
 3  });
 4
 5  jest.mock("../db", () => {
 6    const originalModule = jest.requireActual("../db");
 7
 8    return {
 9      __esModule: true,
10      ...originalModule,
11      getConnection: jest.fn(
12        () => new DBConnection(new MockQueryRunner(queryExecutor))
13      ),
14    };
15  });
```

Listing 5.1: Example of mocks in Jest

5.1.2 Export

The export API in the current version of the Onlajny system is not tested. This is set historically because the functionality had to be delivered quickly, and the tests were done on the consumer's side.

A possible solution to this issue would be to implement a mock data source and test only the small data transformations that are happening in the export service.

The other possibility is to test the export documents given as outputs. This is already happening in the current system, but it has to be done manually inside a web browser.

To address this problem and to demonstrate the functionality of the export service, a Postman[3] collection was created as part of this thesis. The collection

[1]`https://jestjs.io/` [cit. 2023-05-19]
[2]`https://ts-sql-query.readthedocs.io/en/stable/` [cit. 2023-05-19]
[3]`https://www.postman.com/` [cit. 2023-05-17]

should provide basic documentation of the available endpoints and make it easier for testers to execute the requests, although the outputs still need to be validated manually.

However, automation of the output testing process could be done. For example, the JSON format could be validated via the JSON Schema. I have implemented this approach using the Cypress testing framework, which is described further in Section 5.1.3 and Ajv JSON Schema validator[4].

5.1.3 Frontend

The most important part of a frontend is UI because users interact with the system through it. Therefore, I decided to focus on automating the UI testing process to make it easier.

For this purpose, I chose to use a library called Cypress[5] because I had experience with it. It is a JS utility that provides the functionality to write and execute tests in a browser environment.

Cypress, unlike, for example, Selenium, runs inside of the automated browser. Testing scripts run alongside the executed application in the same run loop. Therefore, there is no need to execute remote commands in the browser, as Cypress has direct access to it. It can also interact with the tested frontend code, which can be useful when a tester wants to manually set up a state of the application under test. However, access to the tested code is not enabled by default and must be allowed by the developer [50].

There is also a Node.js server behind Cypress, which makes it possible to interact with the backend of the tested application. Thanks to a server process, Cypress can also take screenshots and record videos of the tests performed [50].

```
describe("Match detail spec", () => {
  it("shows cards", () => {
    cy.visit("/match/144668");

    cy.get("strong")
      .contains("Karty")
      .parent()
      .should("contain.text", "8. Pisczcek (POL)");

    cy.get("span")
      .contains("yellowCard")
      .parent()
      .should("contain.text", "8")
      .parent()
      .should("contain.text", "Pisczcek");
  });
});
```

Listing 5.2: Example of a Cypress test

[4]https://ajv.js.org/ [cit. 2023-05-23]
[5]https://www.cypress.io/ [cit. 2023-05-18]

With the help of this testing framework, I was able to automate all the scenarios that were created for UAT (5.2.2). The implementation of the fifth scenario can be seen in Listing 5.2.

The core function is given as the second parameter to the function `it` on line 2. This callback contains a code for one test. We get access to the Cypress object via the variable `cy`. We can navigate to a particular route as seen on line 3. The tester can then select Document Object Model (DOM) elements (e.g. lines 5-7) and perform various assertions on them (e.g. line 8).

5.2 UAT

As an evaluation method for the created frontend web application, User Acceptance Testing was chosen. During UAT the end users validate the software and test whether the implemented solution satisfies their needs [51].

There are many ways to do UAT. For example, in the black-box approach, users and developers work together on the design of the test cases. Both groups are reviewing the specification of requirements for the information needed to design tests [52].

However, the acceptance criteria are often ambiguous or not very easily determined by the user. Therefore, developers must participate in the whole testing process [52].

For the purpose of this thesis, a different approach was selected. In a behavior-based or scenario-based UAT tests are designed from the user's perspective so that they test the behavior of the system. It means that the scenarios describe common use cases of the tested system. The user then takes these scenarios and solves the given tasks. After that, the user gives feedback that is evaluated by the testers [52].

The scenarios for the new Onlajny system were created in the form of a Google form[6]. This form was sent to the users who attempted to solve the tasks and filled in the responses.

5.2.1 Users

There were five users in total who took place in UAT of the new Onlajny web frontend.

Two users were selected from the project team that is managing the current version of the Onlajny system. They are not developers, but they know how the current application behaves and expect the new version to have a similar behavior.

One user is my supervisor, who is a developer and is, therefore, an experienced computer user. However, he has not used the old system, so his expectations were not based on previous experiences with Onlajny.

And last but not least, the other two users are common computer users who also occasionally use the old Onlajny web application. That makes them less

[6]`https://forms.gle/5uRNnZTjS5AMECbc8` [cit. 2023-05-16]

experienced than the previous two groups, but they have moderate experience in both fields - computers and Onlajny.

5.2.2 Scenarios

When creating the scenarios for UAT of the Onlajny web frontend, I focused on the common tasks that users perform in the application.

I created five tasks for the users to complete. Every task deals with a particular part of the web application, and users must find the appropriate information on the page.

Each assignment has an answer field in the Google form. Because of that, we can evaluate whether users were able to find the right data on the page.

All users must also fill in how much time they spent on each task. This is required because we want to know if users encountered any problems during the task solving.

In addition, users may leave a note for any problems they have discovered during the performance of the task.

Task 1 - Games

The first scenario focused on searching for games on the homepage. Users were first asked to find games played on 15. 5. 2012. After that, they were given a task to find which team scored the most goals in hockey.

This task was created to test whether users can navigate to a particular date using the filter located at the top of the main section of the homepage. In addition, it tested whether users can find the proper sport section, the results of the matches, and differ between the home and visitor teams.

Task 2 - League games

Task number two was built on top of the first assignment. Users should find games that were played on 9. 6. 2012. Furthermore, they should locate matches from that date within the Česká fotbalová liga league.

In this scenario, we tested whether users can navigate to a league profile with games from a particular date. Users also needed to find the proper teams and write their names down.

Task 3 - Rosters

The third assignment dealt with match rosters. Users were asked to find a game from 8. 8. 2018 between Finland and Czechia in the Hlinka Gretzky Cup. In that game, they should then locate the rosters and write down who started in the goal for Czechia.

This scenario refreshed the knowledge from the first two with the beginning when users had to find a proper game. Then we focused on the roster information on the match detail page. Users should locate it and understand how to learn who is the goaltender.

Task 4 - Goals

In the fourth task, users were given a link to a particular match between Czechia and Russia. After that, users were asked to find out who scored the first goal of the game and at which time.

Here, we tested whether users could locate the section with goals and assists in the match details. Also, users should find the comment describing the goal to find the exact time of the event.

Task 5 - Penalties

And last but not least, the fifth scenario focused on the penalties. Users got a link to a football match between Poland and Germany and were asked to write down how many cards the referee gave in this game.

This assignment tested if users can find a section with the information about given cards and understand it. Or, with more time, users could find the same information in the match comments, which we could tell by the time spent on this task.

5.2.3 Limitations

The UAT of the front web application in this project has some limitations.

The first is the problem with the external graphics. I did not receive all the necessary materials from the external graphic designer, so I had to build most of the current application UI design on my own.

It means that it does not look very professional today. This could also influence the results of UAT because some of my User Experience (UX) decisions might confuse users and they could have problems with some of the tasks.

However, UX should improve in the future version as I receive all the materials. So, when we get a UX related issue in the feedback from the users, we can just ignore it for now and get back to them when we iterate on the new look of the application.

We are also not testing live games and live data transfers in UAT because it would be much harder to manage the testing.

As I stated earlier, it is not the goal of this thesis to provide extensive and thorough tests, so we decided to test the system with the use of old data.

If we were to test live games, we would need to gather the users in one time. A user would then write a live commentary in the test administration system, which would cause our application to render live data. However, this would be difficult to manage and evaluate, so it is not included in this thesis.

5.2.4 Evaluation

After creating the scenarios (5.2.2), the Google form was distributed between the selected users (5.2.1). When we got answers from all the users, we could evaluate the results.

Scenario 1

The first scenario (5.2.2) revealed a problem with UX in the date picker. Some of the users could not understand at first sight that they can manually edit the date and tried to click back to 2012 in the selector pop-up. This is a problem related to incomplete design materials (5.2.3) and will be addressed in the new version.

Other than that, no issues were discovered. All users were eventually able to complete the task and answered correctly.

As far as time goes, the average was 52 seconds. One can see in Figure 5.1 that the people who misunderstood the date picker spent more time on this task than the other people who picked the date manually.

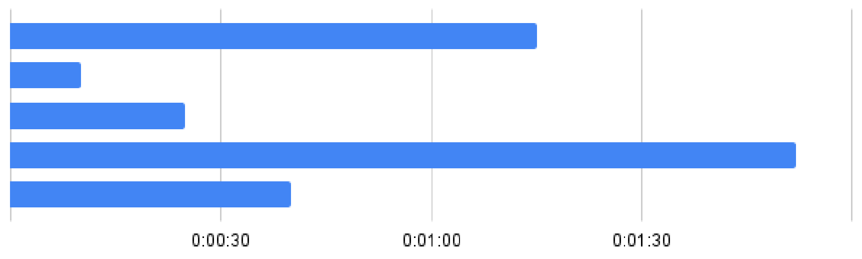

Figure 5.1: Scenario 1 - Time

Scenario 2

The second assignment (5.2.2) was the most complex, and can be seen in time in Figure 5.2. Three users needed two minutes to complete the tasks, and the fastest needed one minute.

That is expected as we wanted people to write all the teams that played on the given date in the given league and it was many.

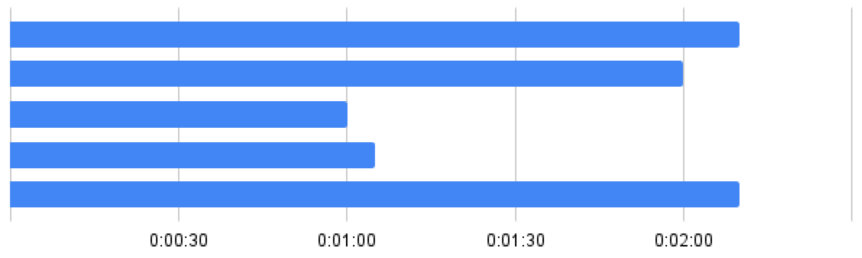

Figure 5.2: Scenario 2 - Time

All the users managed to write the correct answer. However, two of them reported an error when picking the date. This error was caused by an incorrect configuration of the database connection tool and is fixed in the new version.

Scenario 3

In the third task (5.2.2), we encountered the first wrong answer. One user misunderstood the rosters section. They found the section but picked the first Czech defender instead of the goaltender.

This problem could be fixed by providing a legend to the rosters section explaining which positions are listed on which places in the rosters.

The user with the wrong answer also spent the most time solving this task, so this is another indication that they had problems finding the roster section and understanding it.

Other than that, the average time of this task was 32 seconds which is not bad considering the one especially long time.

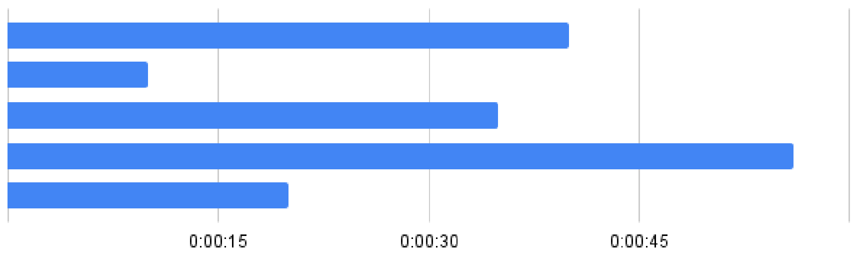

Figure 5.3: Scenario 3 - Time

Scenario 4

Scenario number four (5.2.2) was correctly solved by all the users. However, one of them forgot to write the time of the goal, and the other one wrote only a minute in which the goal was scored.

The first user probably did not read the question properly. The second one found the goal only in the goals and assists section, which is fine, but we could improve the UX so that, for example, the comments could be filtered only to goals.

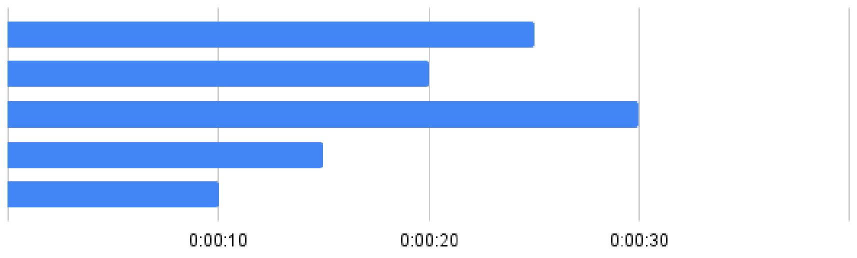

Figure 5.4: Scenario 4 - Time

This was also suggested by the user with the longest time spent on this task. They answered correctly; nevertheless, they suggested the filtering feature so that they would not need to scroll all the way to the first goal.

The average time spent on this assignment was 20 seconds, which could be improved by the filtering feature.

Scenario 5

The last scenario (5.2.2) was also solved correctly by all users and in very fast time. The average time was 12 seconds.

However, one of the users stated that they had a problem understanding the card section without any icon indicating the meaning of the section. This could be improved by adding a card icon to the section.

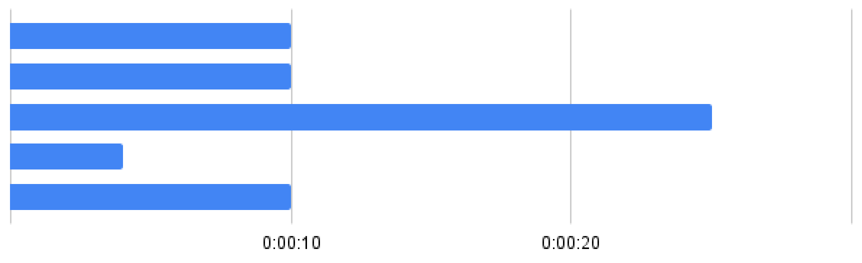

Figure 5.5: Scenario 5 - Time

Chapter 6
Conclusion

This thesis dealt with the revitalization of the Onlajny system. Such assignment consists of four main parts: analysis, design, implementation, and evaluation.

First, it was necessary to analyze the current state of the application and the technology that could be used to improve the Onlajny system. The analysis revealed some issues with the current implementation.

The old system had been built as a monolith where the backend and frontend were closely coupled. It was therefore decided to split the application into backend and frontend parts, as we wanted to develop the new version of the system according to modern web development trends.

In addition, real-time communication had been implemented with a long polling technique, which is not really real-time and tends to load the server unnecessarily. For the purpose of upgrading this functionality, WebSockets were chosen for live data transfer between the backend and frontend parts.

When designing the architecture of the new system, it needed to be taken into account that the backend part should handle frontend requests and provide an export API. After a discussion, it was decided to split the backend into two separate applications. One has export capabilities and the other serves as the backend part of the frontend, allowing the load to be divided between the two.

Implemented export API[1] supports the semantics of the data and its machine readability, as this requirement was one of the main goals of the thesis. JSON-LD was chosen as the technology to achieve this functionality for its simplicity.

Export API was also implemented with respect to the requirement to support the old XML data format. This was done by analyzing the old exports and implementing the new ones in the same structure.

The frontend web application[2] was developed as a single page JS application that is being pre-rendered on the server. The server-side rendering was included in the system due to SEO requirements.

Last but not least, the implemented solution had to be evaluated. User Acceptance Testing was used as the main technique to test the frontend web

[1]`http://test-export-onlajny.esports-13-www5.superhosting.cz` [cit. 2023-05-17]
[2]`http://test-onlajny.esports-13-www5.superhosting.cz` [cit. 2023-05-17]

application.

Moreover, the discovered technical debt of the old system was tackled by designing and partly implementing automatic testing of the individual parts of the system. Unit tests, integration tests and also the UI tests were included in the thesis.

Throughout the implementation of the new Onlajny system, I stumbled upon some difficulties. The biggest one was the technical debt of the old system. I could not look into any documentation, as it is completely missing. The only source of information was the senior developer and the source code. It was very hard to navigate the old code as it was probably never refactored, so I needed to make appointments with its creator, which slowed me down.

However, I was eventually able to understand the system, which can help the company with technical debt a bit. I also suggested that the project managers create a business documentation for the system, so that new developers can more easily understand the application and its parts.

Another problem was the external source for the design of the new web frontend. The UX designer was aware of the deadline of this thesis; nevertheless, he did not deliver the complete draft, so I had to create a large part of the frontend on my own. This was not ideal because I am not an UI expert which could be seen on some of the results in UAT.

To conclude, the current version of the new Onlajny system is planned to be expanded. The frontend application needs to be improved according to a design made by the external UX expert, which has not yet been delivered. In addition, an extensive user testing and load testing is planned in the future.

The new Onlajny system is expected to be released in a beta version in spring 2024. This means that the new system will run alongside the old system so that both of them can be compared. After that, the new system should take over in the autumn of 2024.

Appendix A

Glossary

API Application Programming Interface. 3–5, 8, 12, 14–21, 25, 27–29, 34, 43

AWS Amazon Web Services. 12, 17, 18, 20

DB Database. 12, 31, 33, 34

DOM Document Object Model. 36

ER Entity Relationship. 18

HTML Hypertext Markup Language. 7, 8, 13

HTTP HyperText Transfer Protocol. viii, 4, 5, 8, 9, 12, 20, 21, 27, 30

IRI Internationalized Resource Identifier. 10, 11, 15

JS JavaScript. 5, 7, 8, 13, 17, 20, 21, 23–28, 31, 35, 43

JSON JavaScript Object Notation. vii, 5–7, 10–12, 14, 15, 17, 35

JSON-LD JavaScript Object Notation for Linking Data. 6, 10, 11, 13–15, 17, 43

OWL Web Ontology Language. 9

PHP Hypertext Processor. 12, 23

RDF Resource Description Framework. 9–11

REST Representational State Transfer. vii, 3–5, 14, 15, 20

SEO Search Engine Optimization. 7, 8, 13, 16, 17, 20, 25, 43

SQL Structured Query Language. 33

TS TypeScript. 25, 26, 28, 29

Turtle Terse RDF Triple Language. 10, 11

UAT User Acceptance Testing. vii, 33, 36–39, 41, 43, 44

UI User Interface. 2, 3, 25, 29, 33, 35, 38, 44

UML Unified Modeling Language. 18

URL Uniform Resource Locator. 4, 16, 21, 27

UX User Experience. 38–40, 44

XML Extensible Markup Language. 5, 12, 14, 17, 43

Appendix B

Listings

Appendix C

Bibliography

[1] Frontend Definition, c2023. *TechTerms.com* [online]. [cit. 2023-05-22]. Available from: https://techterms.com/definition/frontend

[2] Backend Definition, c2023. *TechTerms.com* [online]. [cit. 2023-05-22]. Available from: https://techterms.com/definition/backend

[3] FIELDING, Roy, 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Available also from: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Dissertation. University of California, Irvine.

[4] HTTP response status codes. *MDN Web Docs* [online]. [cit. 2022-11-20]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[5] Introducing JSON. *JSON* [online]. [cit. 2022-11-20]. Available from: https://www.json.org/json-en.html

[6] The home of JSON Schema. *JSON Schema* [online]. [cit. 2022-12-17]. Available from: https://json-schema.org/

[7] PEZOA, Felipe, Juan L. REUTTER, Fernando SUAREZ, Martín UGARTE, and Domagoj VRGOČ, 2016. Foundations of JSON Schema. *Proceedings of the 25th International Conference on World Wide Web*. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 263–273. Available from: doi:10.1145/2872427.2883029

[8] Getting Started Step-By-Step. *JSON Schema* [online]. [cit. 2022-12-03]. Available from: https://json-schema.org/learn/getting-started-step-by-step.html

[9] JavaScript. *MDN Web Docs* [online]. [cit. 2022-12-17]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[10] MILLER, Jason, and Addy OSMANI. Rendering on the Web. *Web.dev* [online]. 6 February 2019 [cit. 2022-12-03]. Available from: https://web.dev/rendering-on-the-web/

[11] SHARMA, Dushyant, Rishabh SHUKLA, Anil Kumar GIRI, and Sumit KUMAR, 2019. A Brief Review on Search Engine Optimization. *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. IEEE, 687-692. Available from: doi:10.1109/CONFLUENCE.2019.8776976

[12] The WebSockets API. *MDN Web Docs* [online]. [cit. 2022-10-23]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`

[13] FETTE, Ian, and Alexey MELNIKOV. The WebSocket Protocol. *RFC Editor* [online]. December 2011 [cit. 2022-10-23]. Available from: `https://www.rfc-editor.org/rfc/rfc6455.html`

[14] PIMENTEL, Victoria, and Bradford G. NICKERSON, 2012. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*. IEEE, **16**(4), 45-53. ISSN 1089-7801. Available from: doi:10.1109/MIC.2012.64

[15] LIU, Qigang, and Xiangyang SUN, 2012. Research of Web Real-Time Communication Based on Web Socket. *International Journal of Communications, Network and System Sciences*. **5**(12), 797-801. Available from: doi:10.4236/ijcns.2012.512083

[16] BERNERS-LEE, Tim, James HENDLER, and Ora LASSILA, May 2001. The Semantic Web. *Scientific American*. **284**(5), 34-43.

[17] WOOD, David, Marsha ZAIDMAN, Luke RUTH, and Michael HAUSENBLAS, 2013. *Linked Data: Structured data on the Web*. Manning. ISBN 9781617290398.

[18] SHADBOLT, Nigel, Tim BERNERS-LEE, and Wendy HALL, 2006. The Semantic Web Revisited. *IEEE Intelligent Systems*. IEEE, **21**(3), 96-101. ISSN 1541-1672. Available from: doi:10.1109/MIS.2006.62

[19] BECHHOFER, Sean, Frank VAN HARMELEN, Jim HENDLER, Ian HORROCKS, Deborah L. MCGUINNESS, Peter F. PATEL-SCHNEIDER, and Lynn Andrea STEIN, DEAN, Mike a Guus SCHREIBER, ed., 2004. OWL Web Ontology Language Reference. *W3C* [online]. 10 February 2004 [cit. 2022-12-23]. Available from: `https://www.w3.org/TR/owl-ref/`

[20] BRICKLEY, Dan, R. V. GUHA, and Brian MCBRIDE, ed., 2014. RDF Schema 1.1. *W3C* [online]. 25 February 2014 [cit. 2022-12-23]. Available from: `https://www.w3.org/TR/rdf-schema/`

[21] SPORNY, Manu, Dave LONGLEY, Gregg KELLOGG, Markus LANTHALER, Pierre-Antoine CHAMPIN, and Niklas LINDSTRÖM, 2020. JSON-LD 1.1: A JSON-based Serialization for Linked Data. *W3C* [online]. 16 July 2020 [cit. 2022-10-23]. Available from: `https://www.w3.org/TR/json-ld/`

[22] BIZER, Christian, 2009. The Emerging Web of Linked Data. *IEEE Intelligent Systems.* IEEE, **24**(5), 97-92. ISSN 1541-1672. Available from: doi:10.1109/MIS.2009.102

[23] KRISHNAN, Abhimanyu. JSON vs XML in 2023: Comparing Features and Examples. *Hackr.io* [online]. 05 May, 2023 [cit. 2023-05-16]. Available from: `https://hackr.io/blog/json-vs-xml`

[24] TOM, Edith, Aybüke AURUM a Richard VIDGEN, 2013. An exploration of technical debt. *Journal of Systems and Software.* **86**(6), 1498-1516. ISSN 01641212. Available from: doi:10.1016/j.jss.2012.12.052

[25] OPARA-MARTINS, Justice, Reza SAHANDI a Feng TIAN, 2014. Critical review of vendor lock-in and its impact on adoption of cloud computing. *International Conference on Information Society (i-Society 2014).* IEEE, 2014, 92-97. ISBN 978-1-9083-2038-4. Available from: doi:10.1109/i-Society.2014.7009018

[26] Introduction to structured data markup in Google Search. *Google Developers* [online]. 2023-02-20 [cit. 2023-05-14]. Available from: `https://developers.google.com/search/docs/appearance/structured-data/intro-structured-data`

[27] PRUCIAK, Michał. Node.js vs PHP For Backend Development: Which One Should You Pick?. *Ideamotive* [online]. May 6, 2022 [cit. 2023-05-16]. Available from: `https://www.ideamotive.co/blog/nodejs-vs-php-for-web-development`

[28] State of JavaScript 2022: Front-end Frameworks. *State of JavaScript* [online]. 2022 [cit. 2023-04-12]. Available from: `https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/`

[29] About Node.js. *Node.js* [online]. [cit. 2022-10-23]. Available from: `https://nodejs.org/en/about/`

[30] Differences between Node.js and the Browser. *Node.js* [online]. [cit. 2023-05-16]. Available from: `https://nodejs.dev/en/learn/differences-between-nodejs-and-the-browser/`

[31] POWELL, Alex. Everything You Should Know About JavaScript Runtimes. *Level Up Coding* [online]. Aug 15, 2022 [cit. 2023-05-16]. Available from: `https://levelup.gitconnected.com/dev-explainer-javascript-runtimes-8e4d1e3af405`

[32] About npm. *Npm Docs* [online]. [cit. 2023-05-16]. Available from: `https://docs.npmjs.com/about-npm`

[33] React: A JavaScript library for building user interfaces, c2023. META PLATFORMS, INC. *React* [online]. [cit. 2023-01-15]. Available from: `https://reactjs.org/`

[34] Next.js by Vercel - The React Framework, c2023. VERCEL. *Next.js* [online]. [cit. 2023-01-09]. Available from: `https://nextjs.org/`

[35] Blog - Next.js 13, c2023. VERCEL. *Next.js* [online]. [cit. 2023-05-13]. Available from: `https://nextjs.org/blog/next-13`

[36] Understanding Loosely Typed Programming Languages: A Beginner's Guide. *Coderslang* [online]. Jan 10, 2023 [cit. 2023-05-16]. Available from: `https://coderslang.com/articles/Understanding-Loosely-Typed-Programming-Languages-A-Beginn\ers-Guide/`

[37] Strictly Typed Programming Languages: Advantages and Disadvantages. *Coderslang* [online]. Jan 10, 2023 [cit. 2023-05-16]. Available from: `https://coderslang.com/articles/Strictly-Typed-Programming-Languages-Advantages-and-Disadv\antages/`

[38] TypeScript: Typed JavaScript at Any Scale, c2012-2023. MICROSOFT. *TypeScript* [online]. [cit. 2023-01-09]. Available from: `https://www.typescriptlang.org/`

[39] What is a Monorepo?, c2023. VERCEL. *Turborepo* [online]. [cit. 2023-04-12]. Available from: `https://turbo.build/repo/docs/handbook/what-is-a-monorepo`

[40] Turborepo Quickstart, c2023. VERCEL. *Turborepo* [online]. [cit. 2023-04-12]. Available from: `https://turbo.build/repo/docs`

[41] RAI, Rohit, 2013. *Socket.IO Real-time Web Application Development.* Birmingham: Packt Publishing. ISBN 978-1-78216-078-6.

[42] Introduction. *Socket.IO* [online]. c2023 [cit. 2023-05-13]. Available from: `https://socket.io/docs/v4/`

[43] The Trello Tech Stack, c2023. *Trello Blog* [online]. January 19, 2012 [cit. 2023-05-13]. Available from: `https://blog.trello.com/the-trello-tech-stack`

[44] Express/Node introduction - Learn web development, c1998–2023. *MDN* [online]. Feb 24, 2023 [cit. 2023-05-13]. Available from: `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction`

[45] Template Method, c2014-2023. *Refactoring.Guru* [online]. [cit. 2023-05-19]. Available from: `https://refactoring.guru/design-patterns/template-method`

[46] Singleton, c2014-2023. *Refactoring.Guru* [online]. [cit. 2023-05-19]. Available from: `https://refactoring.guru/design-patterns/singleton`

[47] Builder, c2014-2023. *Refactoring.Guru* [online]. [cit. 2023-05-19]. Available from: `https://refactoring.guru/design-patterns/builder`

[48] RUNESON, Per, July 2006. A survey of unit testing practices. *IEEE Software.* IEEE, **23**(4), 22-29. ISSN 0740-7459. Available from: doi:10.1109/MS.2006.91

[49] LEUNG, H. K. N. a L. WHITE, 1990. A study of integration testing and software regression at the integration level. *Proceedings. Conference on Software Maintenance 1990.* IEEE Comput. Soc. Press, 290-301. ISBN 0-8186-2091-9. Available from: doi:10.1109/ICSM.1990.131377

[50] Key Differences, c2023. *Cypress Documentation* [online]. [cit. 2023-05-18]. Available from: `https://docs.cypress.io/guides/overview/key-differences`

[51] GANESH, K., Sanjay MOHAPATRA, S. P. ANBUUDAYASANKAR a P. SIVAKUMAR, 2014. User Acceptance Test. In: *Enterprise Resource Planning* [online]. Springer, Cham, s. 123–127 [cit. 2023-05-16]. ISBN 978-3-319-05927-3. Available from: `https://doi.org/10.1007/978-3-319-05927-3_9`

[52] LEUNG, Hareton K. N. a Peter W. L. WONG, June 1997. A study of user acceptance tests. *Software Quality Journal.* **6**(2), 137-149. ISSN 09639314. Available from: doi:10.1023/A:1018503800709