BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ADMINISTRATION INTERFACE FOR INFORMATION SYSTEM FOR MUSICIANS

ADMINISTRÁTORSKÉ ROZHRANÍ INFORMAČNÍHO SYSTÉMU PRO HUDEBNÍ UMĚLCE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. VÍT SIKORA**
AUTOR PRÁCE

SUPERVISOR **Doc. Ing. JAROSLAV ZENDULKA, CSc.**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

22125

Student: **Sikora Vít, Bc.**
Programme: Information Technology Field of study: Information Systems
Title: **Administration Interface for Information System for Musicians**
Category: Information Systems
Assignment:

1. Get to know all requirements for an application able to run in the web browser, capable of managing an information system for a choir of artists, including their contacts, web presentation and the possibility to generate artist contracts and concert tickets.
2. Analyze requirements for this application including requirements to persist data in a database. Use UML modelling techniques for the analysis.
3. Design and implement front-end part of the application using React.js framework. Use KORES application (created as a bachelor's thesis) to manage concert hall configuration.
4. Design and implement back-end part of the application in PHP language with MariaDB database.
5. Test the application functionality on a properly chosen set of data.
6. Review achieved results and discuss future continuation of the project.

Recommended literature:

- Grässle, P., Baumann, H., Baumann, P.: UML 2.0 in Action: A Project Based Tutorial. Packt Publishing. 2005. 229 s. ISBN 1-904811-55-8.
- Skotskij, S.: Managing user permissions in your React app. Available at https://medium.com/dailyjs/managing-user-permissions-in-your-react-app-a93a94ff9b40.

Requirements for the semestral defence:

- Items 1, 2, and the design part of item 3.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Zendulka Jaroslav, doc. Ing., CSc.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: October 23, 2018

Abstract

This thesis describes the implementation of a web application that enables central administration of services for a choir of artists or a chamber orchestra. This administration particularly includes the management of website content, artists and their contracts and royalties, concerts, compositions and online tickets reservations and orders. The application integrates the Configuration and Reservation System for Concert Halls (KORES) created as a bachelor's thesis and further manages its reservations.

Abstrakt

Tato práce popisuje realizaci webové aplikace, která umožňuje centrální administraci služeb pro sbor umělců, jako je například komorní orchestr. Do této administrace patří zejména správa obsahu webové prezentace, umělců a jejich smluv a honorářů, koncertů, skladeb a rezervací a objednávek online vstupenek. Aplikace integruje vestavný Konfigurátor a rezervační systém koncertních síní (KORES), vytvořený v rámci bakalářské práce a spravuje dále jím vytvořené objednávky.

Keywords

administration system, CMS, CRM, online tickets, reservation system, Node.js, Javascript, ECMAScript, JSX, React, Redux, PHP, API, dependency injection

Klíčová slova

administrační systém, CMS, CRM, online vstupenky, rezervační system, Node.js, Javascript, ECMAScript, JSX, React, Redux, PHP, API, vkládání závislostí

Reference

SIKORA, Vít. *Administration Interface for Information System for Musicians.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Jaroslav Zendulka, CSc.

Administration Interface for Information System for Musicians

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Mr. Jaroslav Zendulka. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Vít Sikora
May 21, 2019

Acknowledgements

I would like to thank my supervisor, doc. Ing. Jaroslav Zendulka, CSc., for his professional guidance and valuable advice. I would also like to thank the SOVA NET, s.r.o. company for the possibility to develop and deploy the server-side part of the application on their powerful servers with high bandwidth.

Contents

Chapter 1

Introduction

In the following chapters I will describe how to build a modern information system to meet the real-world needs of a choir of artists, in this case a chamber orchestra. There are special requirements for this kind of system — there has to be content management, customer relationship management, a reservation and ticketing system and management of performer contracts and royalties.

It is impossible to find a system that will fully and perfectly fit these needs on the market. Especially, if those are slightly more specific and unique — just like for a chamber orchestra. Typically, the client would need to utilize a combination of existing solutions, usually the website CMS[1] would be standalone and its data like events and performers would not be connected to the rest of the system. CRM[2] would also be standalone, royalties would likely be in a separate financial system, online tickets would be reliant on 3rd party services like Ticketpro or Ticketportal (which is the current situation), and so on.

A perfect solution is then to build a single specialized system tailored to customer's needs.

The customer also wants to integrate a Configuration and Reservation System for Concert Halls [10] (so-called KORES[3]) which is an embedded application that I have created as a bachelor's thesis. It can be easily integrated into any system, but its job ends upon creating a client reservation, so for a deeper interaction (reservation management, online tickets), its backend part has to be customarily implemented. Brief description of KORES can be found in section 2.2.

The most important thing to start is to know, what should be the result. In chapter 2 I am putting customer requirements and goals together with standard requirements a modern application should fulfill in order to be well-maintainable, well-performing and in general, nice to use.

These requirements are further analyzed in chapter 3 and the results are comprehensive UML diagrams that describe the system's use cases and requirements for the data persistence layer.

[1] Content Management System — A system for management of digital content, usually websites (for example, WordPress is a large-scale CMS)

[2] Customer Relationship Management — Company-driven interaction between current and potential customers

[3] from Czech abbreviation „Konfigurátor a rezervační systém koncertních síní“

Chapter 4 is dedicated to creating a design of the whole system, from architecture to prototypes, and chapter 5 takes on from there with implementation.

In chapter 6 I am describing the testing methods used throughout the development and also final usability testing.

Chapter 7 discusses future continuation of the project and the last chapter sums up the achieved results and gained knowledge.

Chapter 2

Requirements and goals definition

This chapter summarizes all requirements the resulting application must fulfill. These requirements will be often referenced and will shape the following development.

2.1 Customer informal specification

Generally the customer focuses on certain functionalities and goals he wants to reach, so in some ways the specification is straightforward, but otherwise also very relaxed (in the meanings of how to reach those goals). There is no technology requirement, comprehensive behavioral guide or a strict specification that would reduce the development phase to a mere step-by-step implementation. Even a use case diagram will first need to be determined from those requirements.

Currently, the customer uses a very dated (10 years or more), simple application, that allows him to manage (old) website content and some simple entities such as events, photos, composers, compositions and performers. This system will be completely shut down, once the new application is deployed. No parts of code or database designs are going to be reused, due to their age and not very high satisfaction of current needs. However, core entities and their attributes will stay the same (e.g. events with name, date, place, etc.) and are mostly only going to be augmented, which will allow a data import from the old system, before it is shut down entirely. The system holds events and photo galleries from as long ago as 1998.

The full, informal customer specification read as follows:

The system must allow to:

- manage web-page's content, at minimum:
  - information pages,
  - news,
  - photo galleries,
  - composers,
  - compositions,

  - performers,
  - events (concerts),
  - places,
  - partners,
- configure concert halls using KORES[1] application [10] (see section 2.2 below),
- manage reservations (KORES-originated) to concert halls with an export to spreadsheet possibility,
- export reservations and customers to a vCard[2] format,
- manage payments for reservations,
- generate online tickets (e-tickets) for events (for customers who purchase them but also manually in administration for use by third-party re-sellers) with some form of validation (e.g. QR, bar code, etc.),
- manage connections among entities (for example events: manage connections to a place, concert hall, performers linked with an instrument, photo gallery and compositions played),
- manage performer contracts and royalties for performing on events,
- manage agencies that arrange performers for events,
- generate royalty overviews (per year and agency) for performers,
- generate royalty liquidations (a document linked to an event with royalty for each performer),
- keep a list of all customers with full history to allow for further engagement (GDPR[3] should be properly handled),
- have a special user type for an accountant or web content manager (limited access user roles),
- enter all data in Czech, English and possibly any other language (the application itself should, however, be in Czech only),
- a data import from a former system (see above) must be carried out at least for events, performers, composers, compositions and photo galleries.

2.2 Configuration and Reservation System for Concert Halls

One of the requirements is to integrate a Configuration and Reservation System for Concert Halls (KORES from now on) [10] for hall configuration in the administration system and for user reservation on the web presentation (which is out of scope of this project). It is a system created as a bachelor's thesis and here is just a brief description.

[1] KORES — Configuration and Reservation System for Concert Halls
[2] en.wikipedia.org/wiki/VCard
[3] GDPR - General Data Protection Regulation

2.2.1 Introduction

KORES is an application designed to be embedded into websites. Its front-end part is written in ECMAScript 2016 and LESS styles and is compiled to single a JavaScript bundle (which includes the styles for convenience) of around 880 kB (220 kB when gzipped[4]). Its usage is simple, just include the bundle and call a single method on a globally exported object „kores“.

To function properly, however, KORES also needs its back-end part, which is a thin REST API with several presenters and models. The API handles mostly data persistence and validation (section & reservation validity, conflicts, etc.). This functionality can be copied to a custom project for a deep system integration (direct model calls), but can be also left standalone.

KORES can work in two modes — configuration and reservation. Configuration mode is typically only available to administrators, who can manage hall events by defining sections of seats with a certain price and availability. Reservation mode is on the other hand usually publicly available and allows its users to create reservations in the predefined halls. Typically, there will be two deployed instances of KORES — one on the public website and one inside some administration.

This project will integrate KORES in configuration mode only.

2.2.2 Example

Few screenshots of a deployed KORES application.

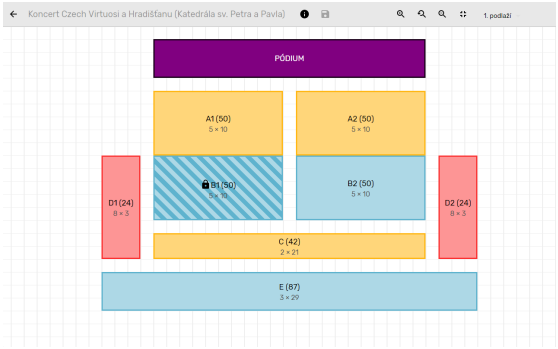

Figure 2.1: KORES in hall configuration mode.

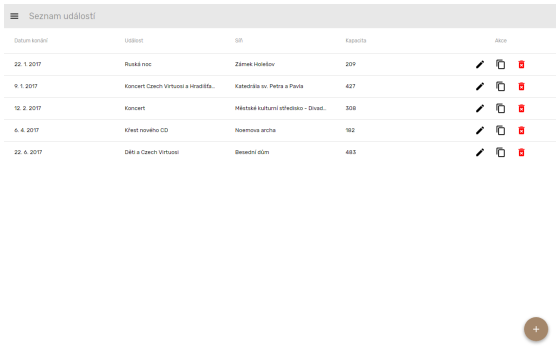

Figure 2.2: KORES showing list of all hall events.

[4] en.wikipedia.org/wiki/Gzip

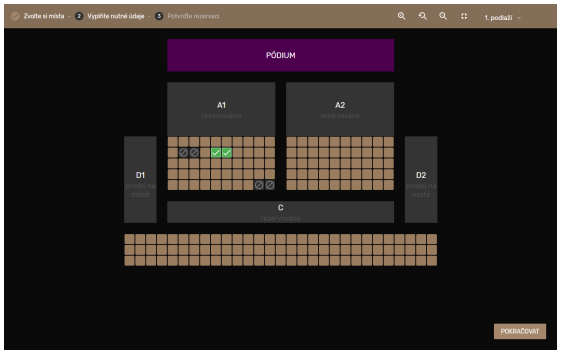

Figure 2.3: KORES in reservation (client) mode.

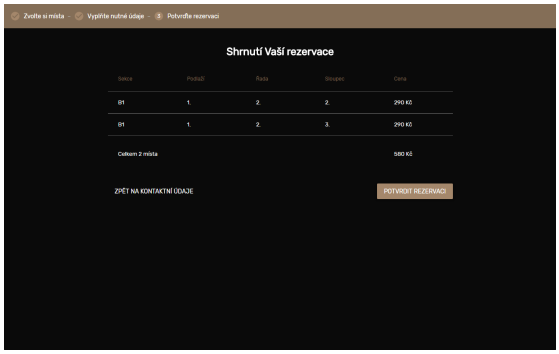

Figure 2.4: KORES showing final reservation step (confirmation).

2.3 Modern web application requirements and goals

These are application requirements the customer understands as implicit in any high quality web application. In my opinion, it is worthwhile to mention them explicitly, as to not forget any of them and refer back to them during decisions in analysis and design.

A modern web application to meet the standard of being usable and high quality should be: [1] [8]

- responsive (not restricted to a single or fixed list of viewport resolutions),
- multi-platform (performing well in all major browsers),
- fluent (no delays for basic actions, asynchronous processing of complex actions, small bundle size, etc.),
- delivering a good user experience (intuitive design, simplicity),
- adequately secured,
- flexibly designed (easy to add new functions, components, to change the way of rendering, etc.),
- well-maintainable (use of good libraries with strong community, clean code, component design, etc.).

These goals are generally accepted as desirable and the customer agrees to try to achieve them as close as possible.

Chapter 3

Analysis of requirements

This chapter combines all requirements defined in the previous one and describes their analysis. Results of the analysis are diagrams and technology restrictions coming directly from requirements.

3.1 Specification analysis

3.1.1 Basic entities

The informal requirements specification (see section 2.1) lists a number of entities that will have various attributes and internal connections. Exact attributes will be specified later, because it is not that important in the analysis or design phase.

The specification explicitly mentions entities as website content pages, news, photo galleries, composers, compositions, performers, events (concerts), partners, reservations, payments, agencies, performer contracts, royalty overview, royalty liquidations and customers. Additional entities like users, roles and permissions, will be added as a result of a user role management requirement. More entities are likely to be added after some time (new feature requests), so defining one should be as straightforward as possible.

For these requirements to be met, the system must be able to provide a way of offering CRUD[1] operations on every one of them. This typically includes a list view (table view) that works as a read operation for a number of entries and a detail view that carries out the rest on a particular entry.

3.1.2 KORES integration

As integrating KORES is directly required, new restrictions automatically arise from this requirement. As documented in the thesis [10], KORES requires:

- JavaScript support on the front-end.
- MariaDB database (if created reservations are going to be further processed).

[1] CRUD — four basic functions of persistent storage: Create, Read, Update and Delete

- REST API with PHP 7 (in case of full back-end integration, see below).

Full back-end integration means that the back-end part of KORES (REST API) would be fully integrated into this system. This should be further analyzed during the design, if beneficial, or it is better to stick with a standalone version with just little database table cross-reference.

3.1.3 Reservation management

This requirement is related to KORES integration, as reservations will be only KORES-originated, however the client also stated, that it must be possible to edit these reservations in the administration.

Therefore, the system must be able to properly manage (basically in a CRUD manner) all user reservations. This is linked to how deep should KORES be integrated in the back-end part and that should be decided later during design phase.

Additionally, the system must allow an export of all reservations in an XLSX format with customer-defined columns.

3.1.4 Payments

The specification mentions reservations payment management. Currently, the last step of creating a reservation in KORES is sending a notification with payment information (in case the customer had chosen to pay in advance). Here it ends, there is no track of the payment, except for the created reservation. The administrator must track the payments himself.

The system must therefore add a way a transaction can be set as paid or even better, integrate it with a bank account and set it automatically. Credit card payment support would also be beneficial, since these are better automated and instant. This is, however, considered a „nice to have“ feature and it is not necessary to fulfill this requirement.

3.1.5 E-tickets

The main disadvantage of seat reservation is that there is still a lot of manual overhead — manual confirmation, pickup at the box office with queues, necessary validation and printing confirmed tickets in advance. Generating tickets online with some sort of an easily validated signature (bar code, QR code or just a secret text) would fully automate this process and also give customers a feel of professionalism.

However, it is still necessary to keep the standard reservation as an option, as not every user might have the needed equipment (computer with a printer or just a smartphone) or skills to grasp the innovation.

As KORES does not support direct ticket purchase, changes would have to take place there as well. There is also a lacking support for hall seat numbering (and this is difficult to make universally applicable).

Every event has its own name, theme and sometimes a logo. Ticket design must reflect that and be at least somewhat prone to customization. Standard design with just a changing logo and text is fine.

Together with ticket validation signature, the system must provide a way how to easily validate any ticket directly at the venue. Creating a custom standalone web or mobile application for this purpose is currently out of scope of this project, however, at least a simple way must be provided.

Tickets will have to be automatically generated upon placing a new reservation in KORES with a delivery selection of e-tickets, that will be free of charge. Currently, it is only possible to pickup tickets at the box office at the venue, or order tickets via Czech Post, which costs the customer 50 CZK (or 100 CZK, if he chooses to pay cash on delivery). So it will have its economical and ecological (if the e-ticket does not end up printed) benefits as well.

To summarize, fulfilling this requirement demands:

- Updating KORES so it supports hall event numbering and offers direct ticket purchase.
- Support for definition of ticket design (PDF, HTML or else) for each venue at least by a logo and textual information.
- Generation of the tickets for printing or just showing on a handheld device (PDF recommended).
- Option to generate a higher number of tickets at once manually in the administration (as stated in customer requirements) — can be satisfied by placing a single big order and setting it as paid.
- Standardized but simple way of ticket signature validation (bar code, QR code, secret code, etc.) for example by providing an endpoint that can perform this validation.

3.1.6 Virtual contact file export

For all entities that it is feasible (they track contact data), it must be possible to export a Virtual contact file (VCF).

3.1.7 Management of relations among entities

This requirement refers to the need for explicit connections between certain entities. For example, events need to be linked to performers in a many-to-many relationship and for every relation there must be an information about a musical instrument used and whether it is a solo performance.

Many of these relationships also need to carry an information about order of the items referenced. For example, compositions on a concert always appear in a defined order.

So to fulfill this requirement, the system must provide standard ways to manage relations between entities (1:1 — typically a simple select box, 1:N, M:N). For 1:N and M:N allow defining order of appearance, and for M:N allow to define additional information.

3.1.8 Performer contracts and royalty documents

The system must allow to set up a contract with any performer for a given event. This contract is valid for one event and one agency and defines the royalty the performer should receive.

Furthermore, the system must be able to generate official documents with all royalties belonging to a specific performer and a specific year (royalty overview), and all royalties within a specific event (royalty liquidation).

3.1.9 Document generation

As is previously, the system must be able to generate some documents. In the future, more document types might be added, so a general approach should taken.

There are currently 3 types of documents:

- **Performer contracts** — for one performer and event.
- **Performer royalty overviews** — for one performer and year.
- **Event royalty liquidations** — for all performers of one event.

The system should offer user-editable templates for each document type. Based on the final template complexity, some might have editable only headers and footers and some will be fully editable. This should be dynamically defined, along with some short manuals for each type.

3.1.10 Customer relationship management

This part of customer requirements describes the possibility to track all customers and the history of their purchases to better craft email campaigns and improve customer engagement and satisfaction. Currently, the requirement will be satisfied by just tracking all customers in the database, but the system should be prepared for future connections to third-party services for CRM, for example SmartEmailing[2].

On the other hand, this customer data storage leads to an obligation to comply with GDPR[3]. Generally, all customers will have to explicitly give their consent with a special privacy statement document prior to their first purchase. This document must fully describe the way customer data are going to be treated and who can possibly access them.

Furthermore, GDPR requires that: [2]

- every customer must be able to access all stored information concerning them,
- every customer have a right to be completely removed from the system in an automated way.

[2] smartemailing.cz
[3] GDPR - General Data Protection Regulation

3.1.11 User and role management

According to customer specification, the administration account must have the possibility to create additional accounts with a different set of privileges. These privileges are defined by a list of entities the role can read or modify.

The system must therefore allow administrators to manage users and user roles in the same way as other entities (CRUD), and enforce the defined rules for each user role.

Technically, the customer requirement calls only for 3 statically defined roles (administrator, web content manager and an accountant), but there is also the generic requirement for good maintainability (section 2.3) so user roles should be completely dynamic and user-defined. This will reduce maintenance costs (there are always new requests) and will also help keep the code clean from expressions like „if (isAdmin() or isMaintainer()) “ that should never be in the front-end code [1]. The responsibility should be always on the back-end only and never hard-coded like that. A final proposal on how to exactly perform user role management and user permissions will be created during design phase, when it will be late enough to consider all other parts of the system, as this will have an impact on everything else.

3.1.12 Internationalization

The specification mentions support for content internationalization — possibility to localize all managed content to more languages. Textual information will need to be stored on a per-locale basis to allow for an unlimited number of languages. The specification explicitly mentions just two languages (with a small possibility of extension in the future), however it is never a good idea to implement an application for a fixed number of languages, so this requirement should be taken as „store data in N defined languages“.

For this requirement to be met, the administration will need to allow editing of every data input in all available languages and persist this information in the same way.

3.1.13 Data import

Customer requirements include that a part of the old database data needs to be migrated to the new system. No matter the design of the old database, this requirement must not have any impact on the design of the new one. Even if it would make the import very complicated and require a custom non-SQL script to be carried out. The new database must be designed to best fulfill current requirements and not carry the burdens of an outdated design.

Because of that, the import will be carried out later and can now be safely taken out from requirements that need to be processed. It should just only be assured, that no important entity attribute is completely removed (not just renamed, split or moved) in the new design.

3.2 Use case diagram

Based on the analysis of requirements conducted above, particular user roles and their respective use cases can be presented in the form of a use case diagram [3] (follows on the

next page). This simply illustrates the interactions the system will provide. As is stated in the bottom of the diagram, roles other than Admin and User are exemplary only — unlimited number of roles with any kind of individual permissions can be created (see section 3.1.11).

The final full-page diagram (fig. 3.1) follows on the next page.

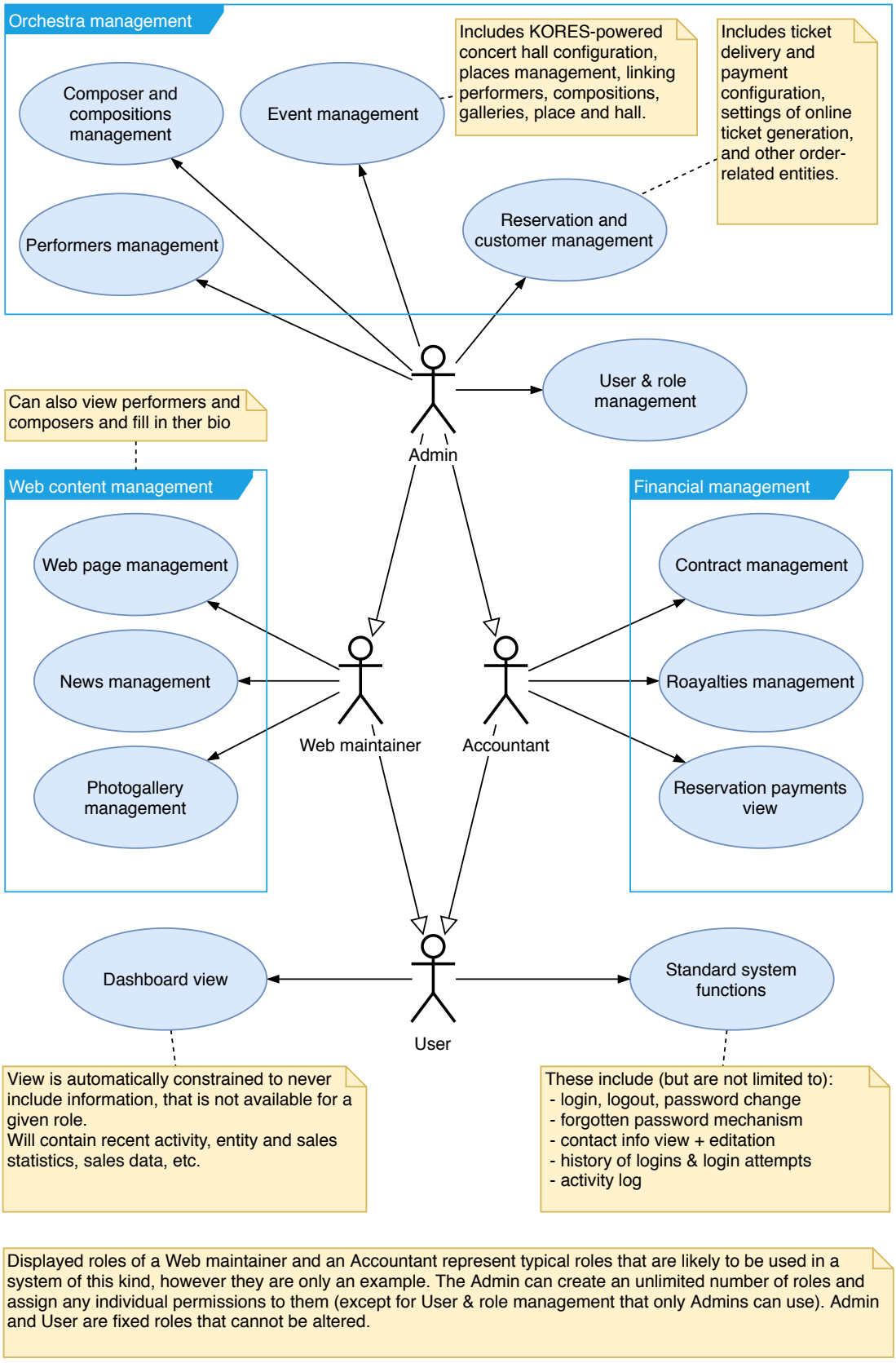

Figure 3.1: Use case diagram

Some unimportant use cases were omitted, for example the management of enumeration lists (countries, currencies, etc.).

3.2.1 Description of individual use cases:

Dashboard view — the default view users will see after logging into the application. Will likely contain quick actions and statistics of all sorts (usage and financial). As it is available to everyone, it will need to automatically constrain the information displayed according to the current user privileges.

Standard system function — these are the base functions the system offers for all users, like logging in and out, resetting a password (through e-mail or manually), changing account details, viewing the history of login attempts, activity log, switching source data language, and so on.

Web content management — these are use cases for editing information available on the public website. Each page, photo gallery and news story has its own URL address and is immediately published so it is advised to grant these permissions with caution.

Financial management — these use cases cover operations over the financial part of the system like payments, royalties and documents so it is advised to grant them as read-only to accountants or other staff.

Orchestra management — these use cases are core operations over the full data of the orchestra, like events, performers, compositions, reservations, etc. It should usually be left for administration accounts only because of its critical value and also for the presence of sensitive customer personal data.

User & role management — management of all users including administrators and definitions of their roles. This use case is fixed for only administration accounts.

All entity management — this applies to all generic entity management, including all of the above mentioned except for dashboard view and standard functions. The use case will consist of displaying a list view of entries with the possibility to sort and filter them, delete them, and to switch to a detail view. On the detail view, the user will be able to edit attributes of the entry or create a new entry.

Reservation and customer management — above standard mentioned CRUD editation of reservations and customers, this use case also covers online ticket configuration (design) and generation for mass printing and payment management.

Event management — this use case also covers hall configuration using embedded KORES.

3.3 Data model

Based on all requirements and some of the assumptions made during analysis, an entity-relationship diagram (fig. 3.2) [3] can be created to capture the way data are going to be

persisted. It is clearly marked what entities belong to the administration system and which come from KORES. Currently there is a single clearly marked link (dependency) between those groups.

For clarity, the model simplifies many-to-many relationships by amending the binding table (that would appear in the database schema) and many entity attributes are illustrative only.

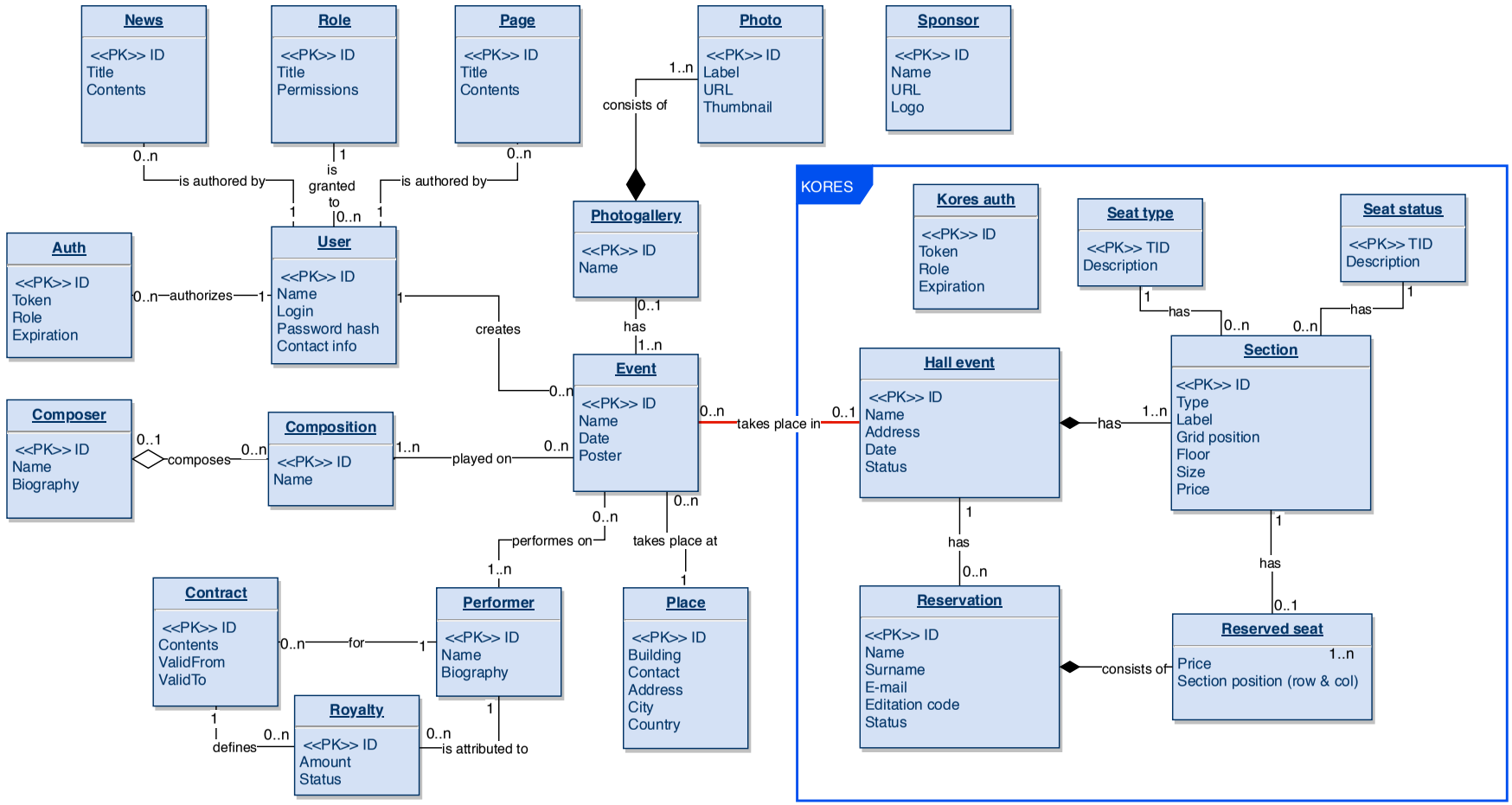

Figure 3.2: Entity-relationship diagram.

3.4 Generic requirements and goals analysis

As stated in section 2.3 of requirements, it is implicitly expected that a good application fulfills a set of generic requirements. All of these are feasible if proper techniques and technologies are used.

All of them are processed in detail in the design phase, since it is not up to analysis to decide technologies, but rather to formulate the exact definition and declare feasibility.

3.4.1 User experience

This topic touches more of the generic goals. The application must definitely be functional at any resolution (the best results, however, can be achieved on only a subset of them) and on all major browsers.

All operations must seem instant or otherwise be accompanied by a clear indication about an operation in the background (data download for example) and this indication must appear immediately. This will give users a feeling of integrity of the system and will boost the overall satisfaction.

Also, the application must be intuitive to use with the smoothest possible user interaction. This means that it must be always clear to the user what action to do to proceed with any given task. Fulfillment is easily tested on users not knowing the system prior to the testing and presenting them with a set of tasks to do. It can be then deduced, what interactions can be further improved and what is working well. [8]

As stated in a great book on user experience (introduction of [8]): „Web user interfaces are much more than buttons, menus and forms for users to fill out. It is the connection between the user and the experience, the first impression, and a lasting impression that either makes a website feel like an old friend or a forgettable passerby. Great web UI design must strike a perfect balance between captivating aesthetics and effortless interactivity. Like an invisible hand, a web interface should guide users through the experience at the speed of thought.“

3.4.2 Adequate security

The application must be fully resistant to all types of common attacks (XSS, CSRF, SQL injection, etc.). OWASP Top 10[4] is a good baseline of what is an absolute must in security of a web application.

3.4.3 Good maintainability and component design

Generally, this means that during the design and implementation of the system, solutions that will reduce maintenance costs and increase the cleanliness, readability or re-usability of the code should always be preferred, even if they seem more time consuming.

[4] OWASP Top 10 Most Critical Web Application Security Risks

Chapter 4

Design

This chapter describes the design of the system. It is a step prior to implementation that is about careful thinking of the system as a whole. Finding out that a wrong technology or procedure was used in the middle of the implementation phase is a bad mistake and costs a lot of time and energy. Proper planning and choice of technologies ahead of time can critically help with avoiding such a scenario.

In this chapter, it is first decided how to integrate KORES into the back-end part of the system. Then, technologies and libraries are selected, followed by a description of handling each requirement from the specification analysis for the front-end application, database, and back-end REST API.

The result of the design stage is a working prototype of the application, in which key design challenges were decided. It is important to make these decisions early, before main implementation phase, but it is also vital to make them in an application-like environment, rather on a whiteboard.

4.1 KORES integration

As was mentioned in sections 3.1.2 and 3.1.3, it must be decided, whether the back-end part of KORES should be fully integrated and become part of the newly created system.

KORES is a standalone embedded application that consists of a compiled JavaScript bundle and a thin REST API [10]. The REST API can be made abstract so that any system wanting to integrate it, can implement it itself (this is the so-called full integration).

Circumstances:

- The system will definitely need to access KORES reservations and to modify them (see section 3.1.3).
- The system will need to read the list of KORES hall events[1] to link them to events.

[1] Hall and event merged in one entity. New event in the same hall is a new entry. Described more in chapter 2.1 in the KORES thesis [10], however only in Czech.

- It is possible that in the future more KORES entities will be needed to have access to.

Advantages of the full integration:

- Direct communication with all KORES models (method calls), instead of intra-server API calls or even direct database access (which is definitely an anti-pattern).
- Possibility to easily edit KORES models for minor new functionalities (minor, because otherwise an update to the front-end client would be necessary too).

Disadvantages of the full integration:

- Inability to easily update to a new version of KORES by simply updating the files. Instead, the models would have to be carefully updated one by one and then audit or test the rest of the system, whether it is not broken (because of the direct model calls).
- Split API — two groups of models and presenters which must be kept in separate namespaces and carefully think through any new cross-dependency.
- It is a more work to do than to just plug in the whole solution.

After careful consideration of both the advantages and disadvantages, it is clear that is **necessary to proceed with full integration**. The reason is to have a robust system that cannot be threatened by simply adding a new requirements that would lead to a tighter integration of the modules.

User reservations are a key functionality that must not be limited by chosen architecture. It is more important to properly handle them, than to keep an embedded module in one piece. Moreover, the API part of KORES is still rather thin and updating it will not yield much hassle, even if it has dependencies.

4.2 Technologies

Based on previous experience with web application development and the results of requirement analysis, I have decided to build the application on the following technologies:

- Front-end part of the application will be created entirely in ECMAScript 8 (compiled to JavaScript) with React[2] and Redux[3] libraries and Material-UI[4] as a UI framework.
  - React will present a convenient way to build a component-driven application that is easily maintained.
  - Redux is great in abstracting the management of application state and as such is also a great tool in increasing code readability and flexibility.

[2] reactjs.org
[3] redux.js.org
[4] material-ui.com

  - Material UI is a themeable and customizable framework and is a full-feature tool for creating immersive user interfaces with focus on user experience. More details in section 4.5.2.
  - LESS[5] will be used to style the application.
  - Webpack[6] and Babel[7] will be used to compile and bundle it together.
- Back-end part of the application will be a PHP 7 REST API integrating the functionality of KORES API (will contain models for both systems).
  - Additional specification of frameworks and technologies is not yet finally decided.
- Database will run MariaDB for its innovations and performance domination over MySQL.

The front-end application will also utilize a LokiJS[8] in-memory database engine for storing data from API offline for fast retrievals and even sorts and look-ups.

The technology stack is similar to the one used in KORES development, largely because it is working well together and I have more experience and knowledge of the inner-workings of some of the libraries.

4.3 Architecture

For clear understanding how different parts of the system relate, an architecture diagram (fig. 4.1 below) might provide aid. Having chosen technologies and the level of KORES integration, this will reflect the final system architecture (although it is very concise and does not show many details, just the big picture).

KORES modules are shown in red, models in green, presenters blue and the remaining packages are yellow.

In the front-end part, the KORES app is just simply rendered on one of the detail views, there is no complex dependency or interaction. Even the KORES model interactions are going to be limited to a necessary minimum to keep KORES as isolated as possible.

[5] A CSS pre-processor, see lesscss.org
[6] webpack.js.org
[7] babeljs.io
[8] lokijs.org

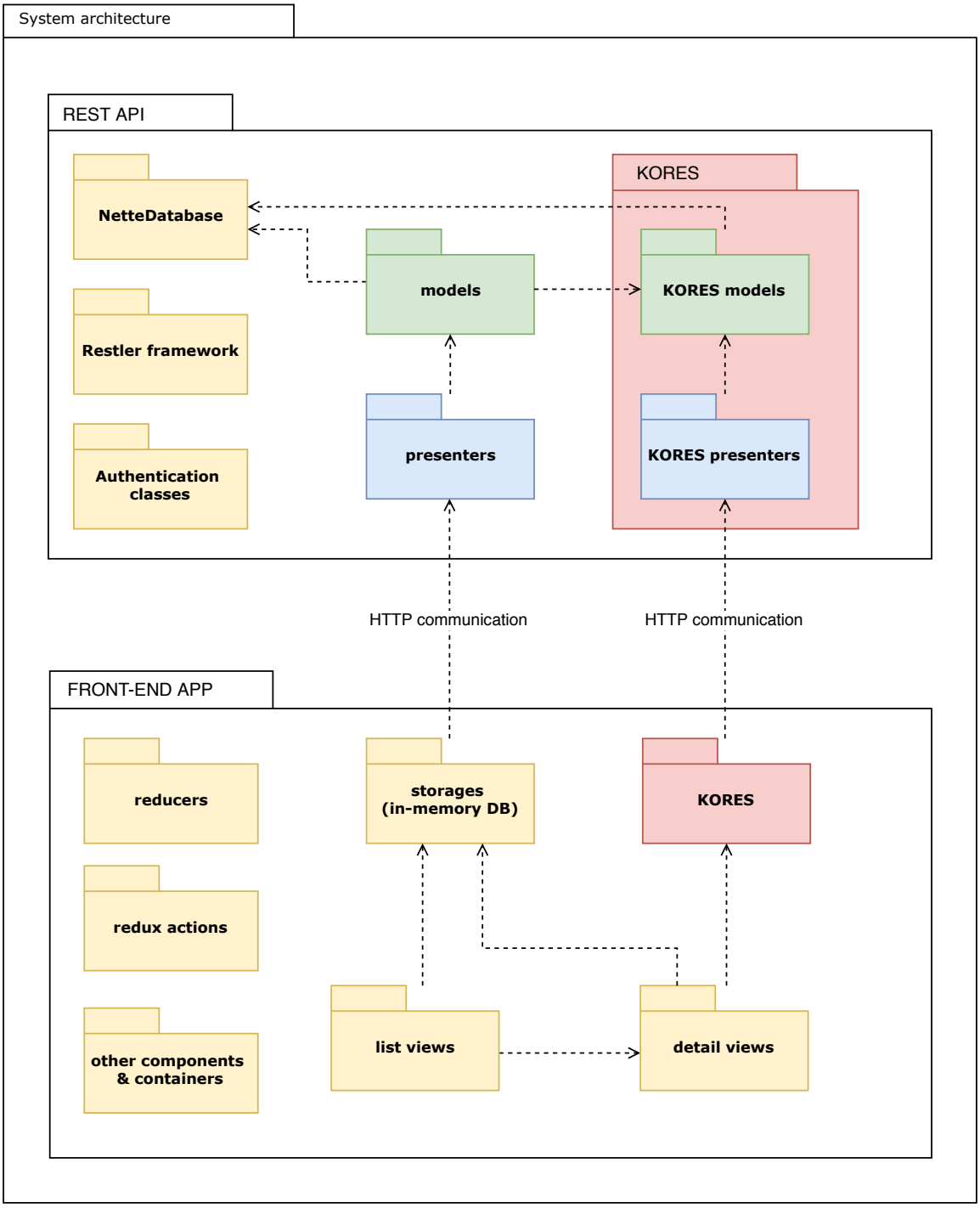

Figure 4.1: Architecture design diagram

4.3.1 Authentication and authorization

It is necessary to ensure that all not authorized requests to the API are automatically discarded (an HTTP 401 response), so that a longer valid client session is unable to perform any action or download more data unless authenticating again. Also, the system must be

fully resistant to potential attackers that might forge requests and look for unprotected endpoints. This might seem uncommon, however as these attacks are often carried out by automated bots [4], every public API, even those of small projects, should be protected thoroughly.

Login procedure

Upon submitting a login form and sending username and password to a special authentication endpoint, the server will check whether obtained credentials are valid, and if so, generate a random[9] 30 byte long authorization token, save it in a database table along with expiration date (by default 2 hours — maximum typical session time of the customer — or longer if the user wants to login permanently) and return this token in the response.

Request authorization

Upon authenticating, the client application will save the obtained token in a cookie[10] with the same expiration time. This way even when the user force refreshes the page or restarts his browser, he will stay logged in, until the expiration time runs out or he explicitly logs out.

This token is going to be sent along every API request in a special custom HTTP header `X-Auth-Token`. The reason for using a header is straightforward, because not every request method (e.g. HTTP GET) allows for data in the request body[11], sending it directly in URL is discouraged (as it would stay in browsing history and may get caught be some third-party applications like browser plugins, firewalls, etc.) and using the value saved in the cookie is prone to CSRF[12] attacks (more in section 4.3.7).

Upon receiving a request, the API must check that the supplied token exists and is valid (otherwise return an HTTP 403 response). After reading which user had created this token, the API will determine whether he has the permission to perform this operation (more on that in section 4.4) and then proceed or again, return an HTTP 403 response.

Session re-validation

The operation of logging-in the user again after a page refresh or other reason of restarting the client application is called re-validation. In the beginning, the application will check whether there already is the token cookie, and if does, it will send a `/me` request to the API to get profile information related to the saved token, or an error, if that token is not valid.

Upon successful login or re-validation, the application will also check, whether a redirect query string[13] is present, in which case it would redirect the user accordingly, instead of switching to main page. Such query string will appear, when the application cannot immediately satisfy the request, because the user is not logged in yet.

[9] Cryptographically secure pseudo-random, see php.net/manual/en/function.random-bytes.php
[10] en.wikipedia.org/wiki/HTTP_cookie
[11] en.wikipedia.org/wiki/POST_(HTTP)
[12] en.wikipedia.org/wiki/Cross-site_request_forgery
[13] en.wikipedia.org/wiki/Query_string

This scenario will occur, when one user of the system sends an internal link (e.g. a detail of an event, like **application.com/event/154**) to another user, that is not, at the moment, logged in. Upon visiting that link, the application will take him to an address like application.com/**login?redirect=/event/154** to log in first and then directly to the address he wants to visit.

KORES authorization

The embedded KORES application has a very similar authorization process (fully documented in section 5.4 of its thesis [10]). It has its own database table to save tokens in and to run the embedded client application a token must be supplied (there are 2 types of token, admin and user ones). It is possible to take advantage of this similarity. During authentication, the API should check, whether the user can use KORES (see 4.4) and if so, copy the newly created token to KORES database table as well (no direct write to the database, but correctly by calling a method of one of KORES models).

This way, the client application can just take the same token it uses to authorize requests, and supply it to the KORES client application. If the user lacks permissions to run it, it will not run (all requests to KORES part of the API will fail).

4.3.2 Data storage

It is necessary to plan how to store, retrieve and update data in the application. As there is a clear distinction between the front-end part (the client app) and the back-end part (the API), it must be decided whether all data requests and updates from the client have to be performed immediately, or whether some sort of client-side caching would not do better.

With a good prior experience with using LokiJS[14] (in-browser performance-focused NoSQL database similar to MongoDB[15]), I decided to use this technology for client-side data storage to speed up the application and decrease the number of API calls.

Client-side caching is however not just about the advantages. There are a few downsides:

- The cache needs to be always in-sync with the real database.
- It must be always clear, how many entries are in the cache and how many aren't — tasks like offline sorting or searching cannot be performed, if the cache doesn't contain all entries.
- If the storage always contains all the entries, the operations must be fast (faster than retrieving them online) and initial setup (downloading all entries) must not slow the application start-up time too much.
- When using optimistic updates[16], failure handlers also need to revert the cache state. [6]

[14] lokijs.org
[15] mongodb.com
[16] Immediate update of the UI (and cache) expecting success of the API call. Needs to include a function that would handle a possible failure and fully revert the update.

Nonetheless, each of these problems can be solved, some of them even generally, and it might be worth the fast, responding interface and lower API utilization.

4.3.3 Cache synchronization

The need for an always-fresh state of the cache can be overcome by using repository classes with standard methods for get, post, put and delete (CRUD) actions that abstract the work with cache and API, so that the programmer does not need to worry about cache synchronization while casually retrieving or updating data. These methods will automatically do all the necessary work (i.e. update the cache only after a successful callback, etc.).

4.3.4 Measuring cache fullness

This problem occurs, when the cache is filled on a per-page basis — only downloading what the user currently needs. For a typical REST API, however, this download schema is not very practical. For example, consider downloading 10 out of 200 news entries. Each entry contains an ID of its author, so for displaying the author's name on each of the 10 pages, it would still be necessary to download all authors. Or to include the author's name in the original query, but that would mimic a GraphQL[17] kind of approach.

For this system, which is unlikely to contain tens of thousands of entries (highest recorded entries number for all entities is less than 1000 for its 20 year history of existence), I have decided to always download and cache all entries of all entities.

4.3.5 Cache performance and data flow

LokiJS is very performance-oriented and claims up to 1.2 M operations per second. Searches in this database are extremely fast even with a higher number of entries, so this is unlikely to be a problem. In contrast, a problem may arise with the initial setup time. As there are entities that contain full text HTML data (pages, news, composers with their biographies, etc.) localized in more mutations, this might require a lot of data to be transmitted. For the web presentation it would mean downloading all pages that exist, and even with compression that is still a lot of unnecessary data.

To cope with that, I have decided to distinguish between *list data* and *detail data*. On the first demand for entity data, the client would download all list data entries, which would contain only basic information like name, title, IDs of other entities (foreign keys), dates and so on. These would be used to render data tables, lists, selection boxes, etc. Detail data would constitute a full representation of the entry with all its attributes and would be only available after specifying a primary key (integer ID in all cases). After downloading, the detail data information would rewrite the former list data entry in the cache and add a binary flag indicating that it is a full entry. Subsequent attempts to retrieve detail data of the same entry would be satisfied from the cache.

This data flow scheme is demonstrated in a sequence diagram [3] on figure 4.2 below.

[17] graphql.org

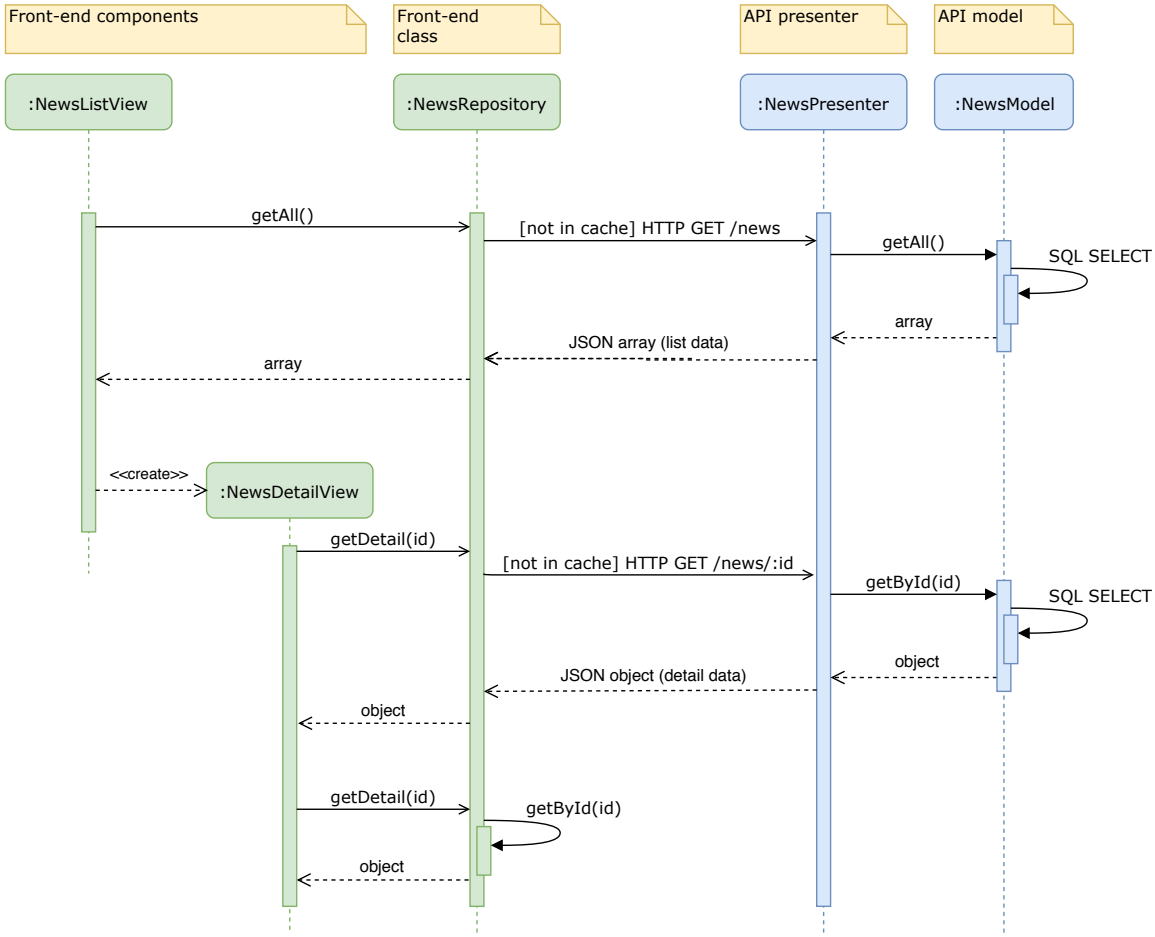

Figure 4.2: Example of data flow between client view components and repository and the API. Client classes in green, server in blue. A typical scenario in case of listing the news and then choosing to edit one. The subsequent call for detail data (with the same ID) is satisfied from cache. The same would happen for list data repeated retrieval.

4.3.6 Optimistic updates

As these can be problematic in this scenario (e.g. multiple cache writes and only one fails on the API), they will generally be avoided. The exception might be delete operations, as they can be carried out by adding a special flag that only the optimistically updated UI would recognize — and removing it if the background operation failed.

Additionally, it is almost impossible to use optimistic updates for standard entity data updates, since there is often some back-end validation and the user needs to know the validation result if it failed. Also, it is more trustworthy-looking for the user if such an important operation takes a short while with a proper indication (see fig. 4.3), than if it is performed instantly (Did it really update? Or just failed and exited silently?).

Figure 4.3: Example of an action indication without optimistic update — animation after clicking and updating the UI only upon receiving HTTP response

4.3.7 Security

One of the requirements on the application is to be secure against all types of common attacks (see section 3.4.2). That is a sensible requirement and in my opinion, one that all publicly available web application should consider. To determine what attack types are common and present some form of a threat, I have decided to rely on a Top 10 list, created by the OWASP Foundation[18] [7], which is widely respected across the API community [4].

Analysis of every point from the list was conducted and brief summaries follow.

A1:2017 - Injection

This vulnerability occurs, when attackers can inject hostile code anywhere in the application, though most often this means SQL injection[19]. This kind of attack can be prevented by a full validation of every single user input. While sounding difficult, it is actually not that hard. As all user input is collected in standard methods of API presenter classes, this validation can be widely automated. For all presenters a list of acceptable parameters must be defined, and anything that is not on the list will simply not be accepted. All database queries will make use of prepared statements[20] with placeholders, therefore user input will never be inserted directly to the SQL query. A database layer (e.g. Nette Database[21]) will be used to ensure that every single input is sanitized. This way, SQL injections are prevented on a system-wide level.

A2:2017 - Broken Authentication

Authentication process can be broken or vulnerable multiple ways. The list specifies especially:

- **Credential stuffing**[22] — automated tests of credential combinations found elsewhere on the internet will be prevented by counting unsuccessful authentication attempts. Upon reaching the limit, the API will discard all future requests from the same IP address.
- **Weak passwords** — password length will be restricted to 8 characters or more and users will be encouraged to use strong and unique passwords. I do not, however, like

[18] Open Web Application Security Project - https://www.owasp.org/index.php/Main_Page
[19] en.wikipedia.org/wiki/SQL_injection
[20] en.wikipedia.org/wiki/Prepared_statement
[21] doc.nette.org/en/2.4/database
[22] owasp.org/index.php/Credential_stuffing

restrictions based on combining both cases, numbers and special symbols, because I think that passphrases[23] might be an even better approach (for some people).

- **Weak or ineffective credential recovery** — there is not going to be any password recovery. Any user losing his credentials will have to contact the administrator.
- **Plain text, encrypted, or weakly hashed passwords** — passwords will be hashed using the Blowfish algorithm with random salt[24] and static system-wide pepper[25]. This hashing algorithm has a high time cost, therefore brute-force cracking of those hashes would have a speed of several passwords per second[26]. The presence of a random salt in each password hash means it is not possible to create a rainbow table[27] and the presence of a pepper provides a protection from dictionary-based[28] hash cracking in case the attacker gained access only to the database and not application code (so that he does not posses the pepper value).
- **Missing or ineffective multi-factor authentication** — this point will not be satisfied, multi-factor authentication is now out of the scope of the project. The customer considers it unnecessary and potentially annoying.
- **Exposing Session IDs in the URL** — will not happen (see section 4.3.1).
- **Session ID rotation and invalidation** — every successful login will create a new token and all tokens will have an expiration date.

So except for the multi-factor authentication (which really might be an overkill), this vulnerability is covered.

A3:2017 - Sensitive Data Exposure

This refers to MITM[29] attacks and other ways of clear text thefts. Data theft during transmission will be prevented by using TLS (HTTPS protocol) with a free certificate from Let's Encrypt[30]. Passwords are protected from breaches by hashing (see point above), and by eliminating SQL injection attacks (first point) no easy database breach should be possible.

A4:2017 - XML External Entities (XXE)

This is not applicable, the application never accepts or processes any XML files.

[23] see xkcd.com/936
[24] see php.net/manual/en/function.password-hash.php
[25] en.wikipedia.org/wiki/Pepper_(cryptography)
[26] Compared to several millions in case of MD5 hashes. Or tens of billions if a GPU is used, see owasp.org/images/e/e0/OWASPBristol-2018-02-19-practical-password-cracking.pdf
[27] en.wikipedia.org/wiki/Rainbow_table
[28] There are lists of most common passwords
[29] en.wikipedia.org/wiki/Man-in-the-middle_attack
[30] letsencrypt.org

A5:2017 - Broken Access Control

As with breaking authentication, access control can be broken various ways:

- **Bypass by URL modification or request forgery** — such attacks will be prevented by token validation and marking every method of every endpoint with the required permission. This will be done using annotations, no method will be left behind.
- **Allowing the item authorship to be changed to another users record** — will be made impossible by attribute white-lists that each model will have. None will contain the author attribute, which is going to be automatically supplied.
- **Elevation of privilege or token tampering** — impossible with the above mentioned (section 4.3.1) token authorization and proper method annotations.
- **CORS**[31] **misconfiguration** — CORS is configured properly, allowing only one domain the client application runs at.

A6:2017 - Security Misconfiguration

This vulnerability covers general security mistakes, such as default users with weak passwords, leaking stack traces with pieces of code, outdated packages and so on. These are the very basics that should be really avoided everywhere, though some of the suggestions might get sometimes forgotten and it is worthwhile to check it all. All of the explicitly mentioned (in the paper) will be taken care of.

A7:2017 - Cross-Site Scripting (XSS)

One of the most common and very frequently exploited vulnerability [9]. Allows an attacker to run arbitrary JavaScript code within victim's application[32]. Exploiting XSS can defeat CSRF and other protections [7], so special attention should be raised to combat this one properly, and at best, with a system-wide solution. The React library, that is used to render all application content, does that automatically[33]. Not even WYSIWYG[34] editors, for which it is essential to get unescaped HTML content, will be susceptible to this attack, because they will obtain the data safely through React props and nothing will be directly rendered (see 4.5.6).

Additionally, the application will not read any query strings or parts of the URL, apart from mentioned redirect (see section 4.3.1) which will be properly parsed.

[31] Cross-origin resource sharing, see en.wikipedia.org/wiki/Cross-origin_resource_sharing
[32] en.wikipedia.org/wiki/Cross-site_scripting
[33] for details, see reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks
[34] „what you see is what you get“, seeen.wikipedia.org/wiki/WYSIWYG

A8:2017 - Insecure Deserialization

This application will never serialize or deserialize any data, except for JSON encoding request and response data. All of the potential vulnerabilities the paper mentions are not related to this project.

A9:2017 - Using Components with Known Vulnerabilities

All components and parts of the technology stack (including DBMS) must be regularly updated. The package manager for client side (npm) does audit all packages for security vulnerabilities by default[35]. With the aid of SensioLabs Security Checker[36] even the back-end package manager (composer) can be scanned for troublesome packages[37].

A10:2017 - Insufficient Logging & Monitoring

This is a place where the system will have room for improvement. Only login attempts are going to be monitored (per IP address; see above). With the customer's approval more logging will be added. Some suggestions are mentioned in section 7.3.

A8:2013 - Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery[38], known from previous OWASP lists, did not make it to the 2017 list, however it is still a threat worth preventing all-together. Using an authorization token, that is sent as an HTTP header, together with correct CORS configuration, will serve as a good CSRF protection.

Potential attackers can lure a logged-in user to visit their malicious website from which they can send cross-site requests by:

- Direct JavaScript XHR[39] — mitigated very simply by CORS configuration (see above) that forces the browser to discard such request before even sending it (using preflight requests[40]).
- An iframe directly requesting an API endpoint — only effective if the authorization token would be saved as a cookie for the same domain. As it is required in headers, iframe cross-requests will never work.
- Ghost form submission[41] — mitigated by a header token as well.

[35] docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities
[36] github.com/sensiolabs/security-checker
[37] logicalmoon.com/2018/05/checking-for-vulnerabilities-with-composer
[38] en.wikipedia.org/wiki/Cross-site_request_forgery
[39] developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[40] developer.mozilla.org/en-US/docs/Glossary/Preflight_request
[41] An HTML form with an API endpoint as a target and prefilled with malicious data, that is submitted without user's intention

4.3.8 Environments

The application will run in 3 environments (branches):

- **Development** — For active development. Represented by the **master** git branch. Unstable. Not deployed anywhere, the client application in this environment is available only on localhost, and the back-end part is deployed using FTP.
- **Staging** — Deployed on a web server on a special URL, uses a development database, only for feature tests and demonstrations for the customer.
- **Production** — Deployed on the main domain on the web server, uses production database. Only well-tested and feature-complete versions will be pushed to this branch.

More environments can be added any time.

4.4 Roles & permissions

Both parts of the system must properly handle all requests and display or allow only what the requesting user has a permission to (see requirement analysis in section 3.1.11). Permission will be enforced on the API (in detail described in section 4.7.6), where it is critical, but also on in the client application (hiding or disabling action button, menu items and whole pages), in order to avoid confusion and errors (see section 4.5.11).

Each user will be bound to a role, and a role will consist of permissions. Admin role is an exception, as it will always have all possible permissions. A role cannot be deleted if there is at least one user who has it assigned.

There are going to be 2 types of permissions — for entities, and for features. Feature permissions will have only 2 abilities — whether the user can view the feature and use the feature. A feature permission will be, for example, KORES usage. Entity permissions, on the other hand, will have more abilities:

- **canDisplay** — whether the user can see the entity in the menu and know, that it exists.
- **canRead** — gives the permission to download and read all list data (see section 4.3.5) in a list view.
- **canReadDetail** — gives the permission the download detail data (might have more information than list data, see section 4.3.5) for single entry and view it in a detail view.
- **canInsert** — permission to create new entries.
- **canEdit** — permission to edit any entry.
- **canEditOwn** — permission to edit entries that user himself created.
- **canDelete** — permission to delete any entry.

- **canDeleteOwn** — permission to delete entries that user himself created.

The **canEditOwn** and **canDeleteOwn** abilites allows for creating a role, that can freely add new content, but also fix and delete any mistakes that could occur.

Entity permissions will be always available for all existing entities, they will be generated automatically.

4.5 Front-end application

4.5.1 Layout

The client application layout is going to be simple and concise. The reasoning is straightforward, it is essential to create a usable and easy to navigate system in order to satisfy user experience requirements (see 3.4.1) and to have a responsive layout that works on every screen resolution without over-complicated layout changes on breakpoints. In the end, the application is going to be tested on end-users, and if one fails to navigate to a required section or loses perception of where he is, it is a very bad result.

Following figures show the planned top-level layout on desktop and handheld device aspect ratios.

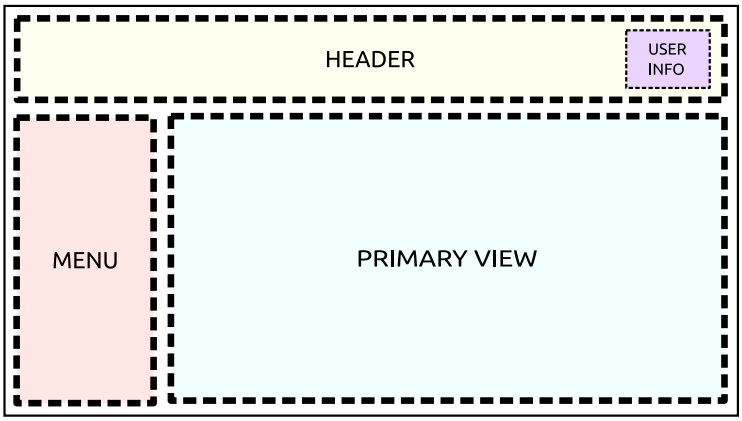

Figure 4.4: Application layout on desktop resolution.

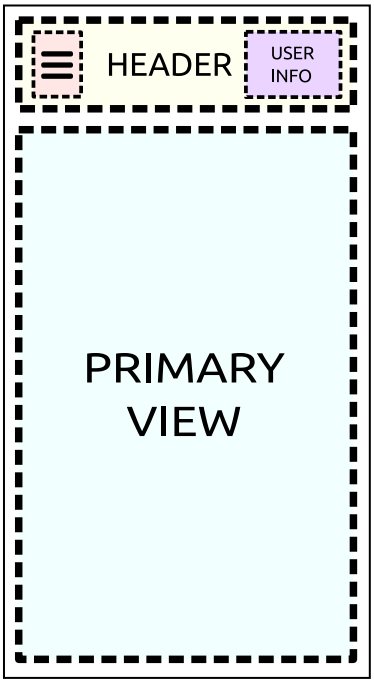

Figure 4.5: Application layout on mobile resolution.

The top region of the screen is going to be occupied by an always-visible header, that will hold important information, such as the logged in user and a simple user menu with an option to visit/edit profile data and a button for logging out.

The left region will contain a drawer[42] with a navigation menu that will include all entities and pages the user has the permission to view or read. This drawer will be retractable and implicitly hidden on smaller (typically under 640 px of viewport width) screens to make more space for the content.

The remaining surface of the screen will house the current primary view. That will be either:

- **List view** — containing a data table of entity entries.
- **Detail view** — a view of the current entry data (can have tabs with additional sub-views, typically for management of entity relations or other special operations).
- **Page view** — custom pages like the dashboard or KORES view.
- **Special view** — currently only login screen (no menu).

Detailed design of each view will be determined later. Changes to the layout are possible if the user tests show some design flaws.

4.5.2 Design framework

Due to the high number of various interface elements, it is highly beneficial to utilize a design framework (or a library) that provides a large set of implementation-ready components that have a unified and quality design. In section 4.2 it was decided that this library will be Material UI[43], which is a collection of React components that implement Google's Material Design[44].

This framework is particularly highly themeable and is convenient to adapt it to project specific needs, mainly because:

- All colors are defined in a provided palette. Additional colors for show drops and focus highlight are then derived from these palette colors.
- It is easy and straightforward to augment basic component functionality. For example, to create a button that shows a loader animation while waiting for completing its action, a simple wrapping component with few lines of code will do all the work.
- Clean component design — a combination of (customizable) low-level components rather than one large ambiguously behaving (black-box like[45]) component.

The simplicity and clean component design well comply with the good maintainability requirements mentioned in section 3.4.3.

The framework, however, does not provide just interaction elements and inputs. There are also large container components like Paper[46], Modal, Drawer and Appbar, Typography

[42] Side sheets containing supplementary content that are anchored to the left or right edge of the screen. Example here material-ui.com/demos/drawers.
[43] https://material-ui.com
[44] https://material.io
[45] en.wikipedia.org/wiki/Black_box
[46] Paper and other component demos can be found at material-ui.com/demos/paper

component for textual elements, universal Table component, and so on. Therefore, it is rather simple to create a full Material Design experience and unify the design of the whole application.

4.5.3 Compilation

As was mentioned in the technology selection (see 4.2), the tools Webpack and Babel will be used to compile the application. The result of the compilation will be a single JavaScript file, accompanied by an assets folder with all used styles (also pre-processed), fonts and images.

Webpack is a module bundler, it is used to process all used assets, from JavaScript and LESS files, to images and even text files. Based on the configuration, it can merge, split, optimize, minify[47] or lazy-load any asset. It has to be accompanied by so-called loaders, that specialize on a given file type. This why Webpack is so powerful — it can process literally anything if it has a loader for it. One of these loaders is a babel-loader[48]. It works together with the Babel library to transpile[49] files written in the most recent version of ECMAScript[50] with even proposals to this specification, to the version widely support by current browsers (currently ECMAScript 5.1[51]). Webpack can then merge and further optimize and minify these files.

The same will be applied to styles written in LESS. They will get processed to CSS files, autoprefixed[52] and then merged and optimized.

Using these features, it is possible to use the latest technologies and in the same time have the application compatible with outdated browsers (which is generic application requirement that the customer expects, see section 2.3). The current browser target for the Babel transpiler and CSS autoprefixer (a browserslist[53] string) is **last 2 versions and > 0.25%, > 1%**[54], which, at the time of writing, covers **87.23%** of the global browser market[55].

4.5.4 Entity management

Operations over multiple items (i.e. choosing, deleting) will be carried out using a list view implementing a Material UI Table[56] that will be sortable and with a fulltext search. Both these operations will be done offline, operating only on list data already downloaded inside in-memory database, as this is extremely fast and does not put any load on the server (explained in detail in section 4.3.5).

Every row of the list view will trigger navigation on the particular detail view with tabs and forms. Upon this action, detail data for a given entity will be re-downloaded with more

[47] en.wikipedia.org/wiki/Minification_(programming)
[48] github.com/babel/babel-loader
[49] en.wikipedia.org/wiki/Source-to-source_compiler
[50] en.wikipedia.org/wiki/ECMAScript
[51] kangax.github.io/compat-table/es6
[52] css-tricks.com/autoprefixer
[53] github.com/browserslist/browserslist
[54] This means: Support last 2 versions of all browsers that have more than 0.25 % of the global market share, or those individual browser versions, that have more than 1 %.
[55] see https://browserl.ist/?q=last+2+versions+and+%3E+0.25%25%2C+%3E+1%25
[56] material-ui.com/api/table

information. Forms will be dynamically generated from their definitions in the detail view container. Detail view will be also triggered by a button for creating a new entry.

Following figures show how these list and detail views might look like (they are drafts).

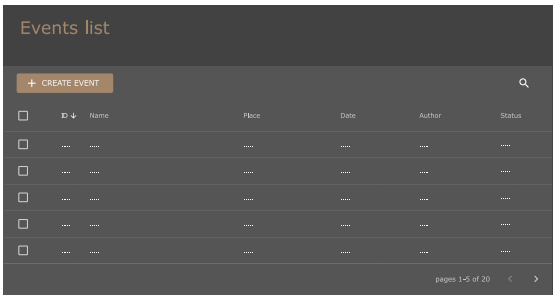

Figure 4.6: Example of event list view (draft)

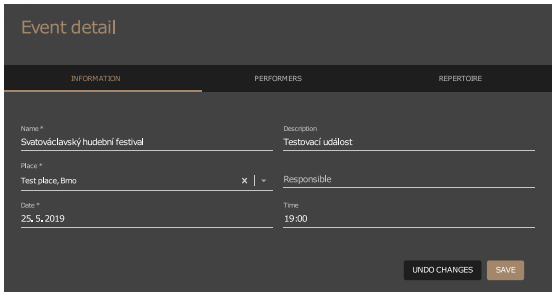

Figure 4.7: Example of event detail view (draft)

Relations to other entities (e.g. performers performing on a given event) and advanced editing procedures (like gallery editing) will be managed in special tabs on the detail pages (handled by special components).

Both list and detail views must have a core implementation in BaseList and BaseDetailPage classes, so that it is very convenient to maintain their functionality and keep a high level of system integrity and uniformity of implementations even in spite of a high number of such views. Each custom list and detail view must then extend this base class and provide only methods which differ from the base functionality. The base classes should be designed with this in mind and provide a number of short and concise methods (see section 3.4.3).

Form components

Detail view forms will have to cope with various data types. Entity attributes are not just texts and numbers, but also files and images, dates and times, code-list and entity selections, HTML data or on/off switches. Each of those types will be handled by a special form component. Files and images will use a file upload component, that will support drag&drop and show the uploaded file or image preview. For date and time types there will be some date-picker plugin and for selections a selectbox (optionally with a full-text search to ease selecting from a large data set). An HTML editor component is described in a dedicated section 4.5.6.

4.5.5 Entity relationship management

Detail view must possess a feature to manage entity relations, such as, for example, linking compositions, that are going to be performed (in a specified order) on an event. These relations will sometimes be accompanied by additional information, for example, event performers must have a specified instrument they are going to use for the event.

To summarize, it is necessary to create a way to describe a set of related entries of one entity with defined order and sometimes additional data. A concept to showcase the feasibility

was created already in Material UI components, so it might look close to the final result of implementation. See the figures below.

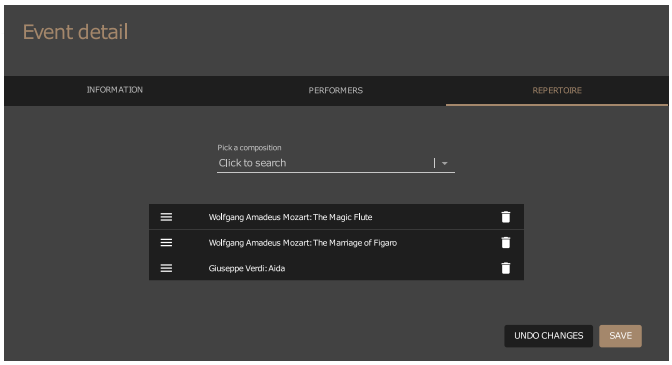

Figure 4.8: Draft of relationship management component

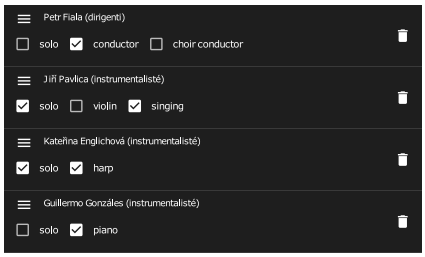

Figure 4.9: Draft of entity relations with additional data

The figure 4.8 shows a concept of a relationship management component that consists of an item picker at the top, that must be able to look up entries using full-text search and then add them to the list below. Every list item has handle icon on the left that must allow sorting by mouse dragging and a trash icon on the right to remove the item from the list.

Figure 4.9 displays how relations that need additional information might look like. It should be unrestricted to add even more form items on the relation item component.

A generic and re-usable component should be created, and custom implementations should only provide custom sub-components to alter the looks and behavior of individual parts (see section 3.4.3).

4.5.6 Rich text editor

As it is necessary to edit rich HTML content, an appropriate editor must be chosen for this task. Based on previous experience, I decided to integrate a well-known and established editor TinyMCE[57]. The key advantages are wide plugin support (even for simple custom plugins), Czech language support, ready-made file-manager plugin with PHP counter-parts (more in section 4.7.12), support for custom skins and zero cost. Official React package was also recently released[58], so it is not anymore necessary to dangerously inject direct HTML code to a textarea tag, which would be susceptible to an XSS attack (see XSS protection in section 4.3.7).

4.5.7 Routing

Routing (URL scheme) will be taken care of by a React Router[59] package configured in a way so that each list and detail view has a custom and unique URL address. The actual

[57] tiny.cloud
[58] npmjs.com/package/tinymce/tinymce-react
[59] reacttraining.com/react-router

navigation will be performed using browser standardized History API[60] so that it is done completely without any page reloads but keeps full history and the possibility for the user to jump back and forward. This is the currently best approach to routing in fully JavaScript web applications.

4.5.8 Reservations

A reservation list view must include a button for exporting all reservations in a predefined Excel spreadsheet (more in section 4.7.7).

Reservation detail page must provide an option to download all reservation tickets. The administrator must always have this option in order to print them if necessary or just allocate a number of tickets for some special guests using a custom reservation (see section 3.1.5). The detail page must also allow altering the seat selection of the reservation, which must trigger ticket re-generation on the API (more in section 4.7.8).

A reservation can be always set manually as paid (with a date of payment) or not paid.

4.5.9 Customer relationship management

Customers as an entity will be treated no different than other entities (list + detail view). Additional third-party service integrations will be handled in custom components.

To fulfill the requirements imposed by GDPR, there will be a special button for deleting (or pseudonymizing[61] if applicable) all data related to a given customer (for a situation where the customer places a request to be deleted). Upon creating any kind of reservation or order, all customers will have to agree with the contents of Protection of Personal Data document[62].

4.5.10 Document generation

As the analysis states, application will generate template-based documents (see 3.1.9). These templates will be user-editable in a rich text (HTML) editing plugin (see section 4.5.6). Based on the chosen template processor, some plugin should be provided, that would highlight template tags, so that simple edits would be feasible for simple users with no knowledge of template processor syntax (more in the API part in section 4.7.10).

4.5.11 User role permissions

Definition

Role will be a standard entity that has a list and detail view. On the detail view, a special tab will be present, allowing for permissions definition. This will be done by component consisting of a matrix of checkboxes, where permission names (entities and features) will

[60] developer.mozilla.org/en-US/docs/Web/API/History
[61] en.wikipedia.org/wiki/Pseudonymization
[62] czechvirtuosi.cz/stranka/ochrana-osobnich-udaju

be in the first left column and abilities (see 4.4) in the top header row. The user can then check those abilities of those entities he wants to assign for the role. Some checks will implicitly trigger others, for example, setting a **canInsert** ability will automatically check **canDisplay**, **canRead** and **canReadDetail** of the same entity and vice versa.

Enforcement

A good approach to handle user role restrictions in a front-end application would be a one that is consistent and universal (same approach for all constrained components). Using some well-known and proven library directly for this purpose seems reasonable. For this project I will use the one recommended by my supervisor — CASL [11]. The features described in the article seem very convincing, so it is likely way better than using some hand-made micro functions or decorators.

The navigation menu in the left menu is going to be rendered based on a route configuration file. Each entry will also specify the name of the related permission. If this permission is missing or lacks the **canDisplay** ability (see section 4.4), the menu item will simply not be rendered and the route will be unknown to React Router (see section 4.5.7). Visiting an unknown address must trigger a user-friendly error page. This behavior will be shared among both entity and feature permissions. The **canUse** ability, which is exclusive to feature permissions will generally be enforced only on the API and other cases will be handled individually (current features are handled solely on the API).

The rest of the entity permission abilities are going to be enforced in the client application[63] in the following ways (if they are missing):

- **canRead** — instead of a list view, the user will simply see an error message, stating that he should ask the administrator for additional permissions.
- **canReadDetail** — list view items are not clickable and detail route is unknown to the router.
- **canInsert** — no button to add a new entry is present on the list view, and if the user manipulates the URL to visit an entry-adding interface, the save button would be missing, form items disabled, and a warning displayed at the top.
- **canEdit** — no save button, form items are disabled and a warning is displayed at the top, when displaying a detail view of an existing entry. If **canEditOwn** is present, this will not happen on entries originally created by the same user and the warning will mention it.
- **canEditOwn** — same as **canEdit**, but even for authored ones.
- **canDelete** — list view checkboxes are disabled, so it is not possible to check entries to remove (currently, the selection does not have any other function). If **canDeleteOwn** is present, entries originally created by the same user will have a selection checkbox enabled.
- **canDeleteOwn** — same as **canDelete**, but even for authored ones.

[63] This section mentions only client application restrictions. Logically, these abilities are primarily enforced on the API (see section 4.7.6

4.5.12 Content localization

The global Redux[64] store will keep an information about currently selected input language. This will be the language all text fields and inputs will use. There will be a language switch component on every list and detail view. These fields (on both views) will always indicate, which language version they are currently displaying, e.g. by appending the language name in parenthesis after description, or showing a small flag. This way it will be always easy to tell localized fields from the shared.

In the detail page form definitions, there will be a flag to set a field as localized. For example, news title is likely to be, however a field for entering a title picture or an external URL address not (but might be). These localized fields are going to be downloaded and uploaded back to the API in all language versions at once, so it would be possible to fill in all language versions at once (no repeated edits, just by toggling the language switcher). This solution completely abstracts the separate handling that will be necessary for such fields, leading to a great time savings when defining a lot of form fields, which is important (there are going to be tens or even hundreds of fields).

4.5.13 Prototype

During design, a prototype was gradually built to test and validate design decisions. Some concepts, like that of a list and detail view, or entity relations components were created in the prototype. Following figures show the last version, that will be used as a springboard for the implementation phase.

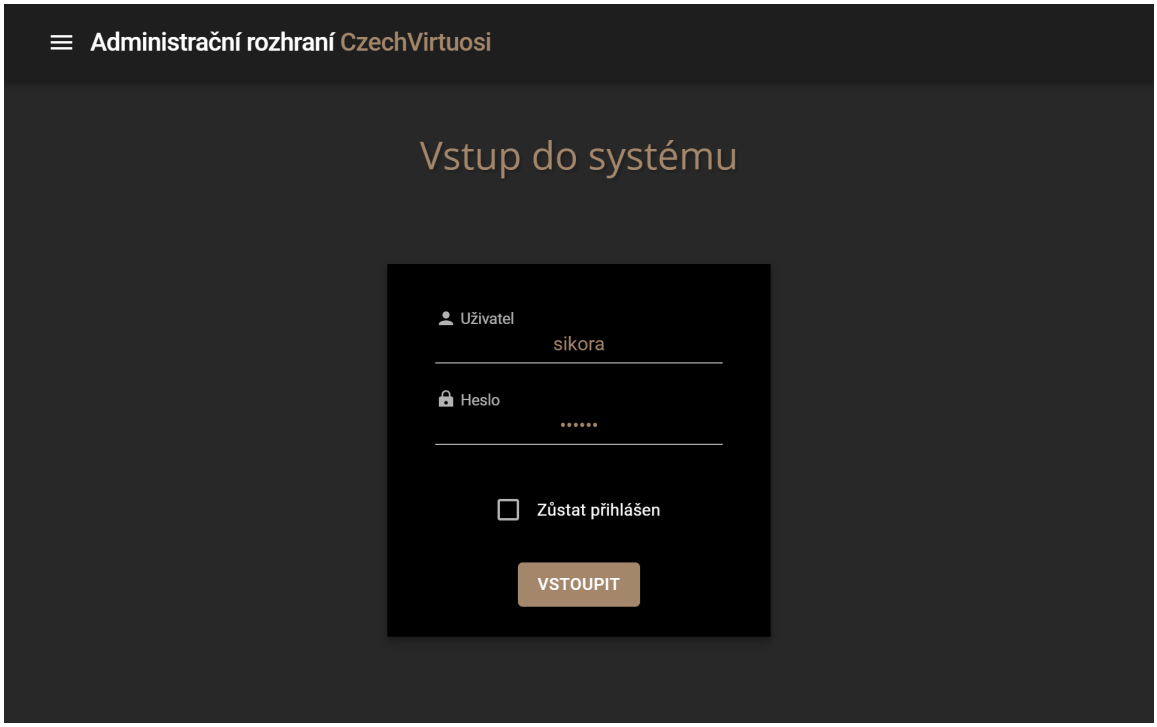

Figure 4.10: Prototype login page

[64] see technologies in section 4.2

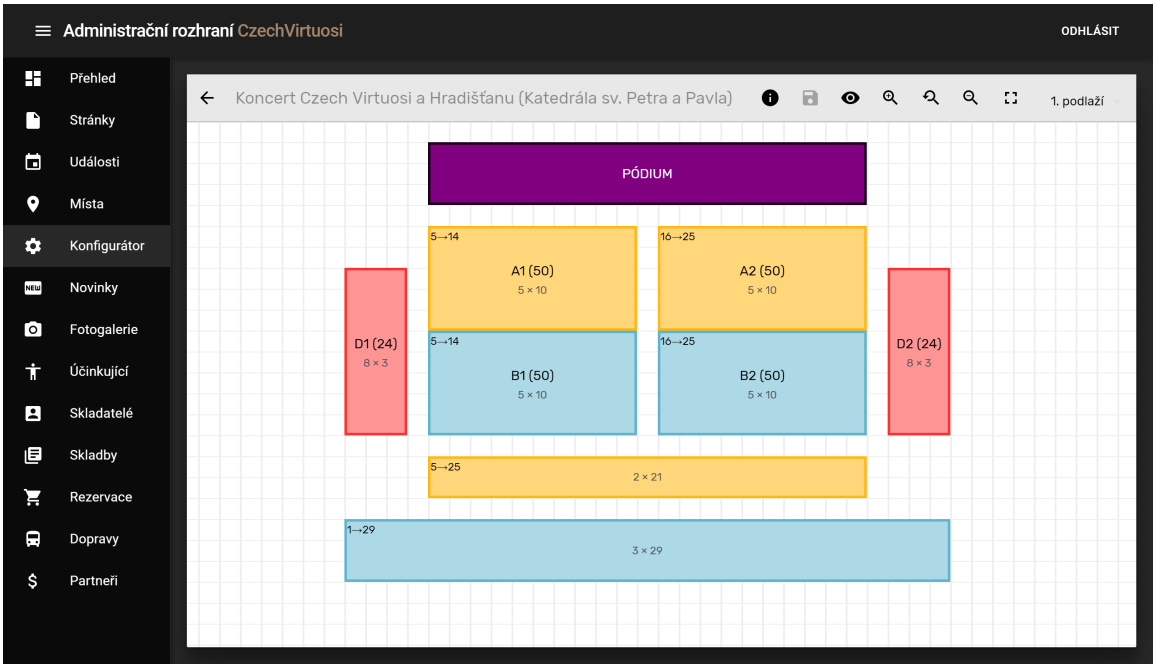

Figure 4.11: Prototype — editing a hall using integrated KORES

4.6 Database

This section describes the design of the database. As was mentioned in technology selection (section 4.2), it will be a MariaDB relational database[65].

Because it was decided to fully integrate KORES API (see section 4.1), KORES tables will have to share the same database (there will be relations between them). For clarity, KORES tables will be prefixed `kores_` and application tables will be prefixed `czechvirtuosi_`.

4.6.1 Versions

In order to have a fully independent development environment, it is necessary to have a development database. Only in that case it is perfectly safe to develop new features and not interfere with the version in production and at the same time have the possibility to stress test the app (massive deletes, data imports, resets or just unit tests).

It is also important to have a script that will sync or reset the development database when necessary (in order to save time whenever it is necessary to reproduce a bug that appeared in production). And even more important to have a standard way of upgrading the production database together with new application version (apply changes in the schema), e.g. an ORM migration or just a simple SQL file.

If the customer wants, it might be good to create a so-called „staging“ database, that is used to showcase new features which are yet to be discussed or tested, but are not ready for production. The application must be built so it is convenient to create even more database

[65] mariadb.org

versions (there are more reasons to add them, e.g. features developed for a long time, locked version for use as a showcase, pre-production, etc.).

The solution is to have a configuration file for every environment. Or better, have a base configuration file and small differential files for other environments. These would specify and override database location, name and credentials, so it will be straightforward to just add more databases.

In the beginning, the application will have 3 environments (see section 4.3.8 above) and two databases (dev and production) with a staging environment linked to the development database. Future changes are then possible and the system will be ready, when having 2 database versions will not be sufficient.

4.6.2 Object-relational mapping

From the beginning I was unsure whether to incorporate an ORM[66] layer, e.g. Doctrine[67]. As usual, there are some upsides and some downsides, and generally it is not suitable for every project. To decide, I considered these key arguments.

Advantages

- **Portability** — not depending on a specific database technology.
- **Zero manual maintenance of DB schema** — schema is generated from entity classes, so no need of manual operations, especially with multiple versions of the database (development, staging, production, etc.), thats what migrations[68] are for.
- **Automated nested query resolving** — possibility to pull additional nested data at any time when needed (though it is not used as much in an API — additional pull often occur during template processing).

Disadvantages

- **Performance** — the curse of all ORM layers, they need to translate all calls to raw SQL, but particularly some subsequent nested calls (that were not predicted) repeated in a loop can have disastrous effects on performance.
- **Learning curve** — as I have never used an ORM layer on a larger-scale project before (only small projects; or have been assigned to optimize[69] some critical queries on tremendously slow company websites), it would take a considerable amount of time to adopt the technology and especially to use it in ways that do not hurt performance.
- **Complex queries and optimization** — ORM layers are typically not suited for big, complex queries with multiple joins and groups, therefore, in such cases, I would

[66] en.wikipedia.org/wiki/Object-relational_mapping
[67] doctrine-project.org
[68] doctrine-project.org/projects/doctrine-migrations/en/2.0/reference/introduction.html
[69] rewrite from an ORM call to pure, carefully hand-crafted SQL query, that was faster by a magnitude of 2-3

still have to resort to raw SQL. The same applies to optimization — if I want to, say, optimize a query using UNIONs[70], I would have to remove the ORM call and again use plain SQL. And since I have decided to output all entity entries in one request (see section 4.3.4), some optimizations might be necessary.

- **KORES** — KORES is not using ORM and since there are several relations between tables of both systems, adopting ORM would ask for rewriting KORES entities to ORM as well.

Verdict

Even though the advantages of using ORM are looking good and especially using migrations would greatly ease updates and synchronization between development and production databases, the disadvantages are far more outweighing for this project. Since the application is rather large and the implementation will be very time consuming on its own, I am a little worried about incorporating a technology that will require probably a lot of learning.

The performance will be very important, and my existing experience with ORMs (optimizing extremely slow Doctrine queries and using Hibernate[71] in projects) leaves me with a perception of a very bad real-world performance (unless it is, of course, very well written by someone who understands well the inner-workings of the given layer; I do not).

Considering the above mentioned, I have decided **not to use object-relational mapping**.

4.6.3 Content localization

In order to persist localized versions of some entity attributes, extra tables are necessary. They will have a uniform naming convention, so saving and retrieving joint entity data can be abstracted and automated (described later in section 4.7.5). When, for example, an event table is named **czechvirtuosi_events** (the entity table, from now on), the table with localized attributes will be named **czechvirtuosi_event_texts** (the text table, from now on).

All entity tables will have an integer-type, automatically incremented **id**, that will always be the primary key[72], and they will lack all localized attribute columns. These will be in the text tables, along with **itemId** [73] and **language** [74] columns, that will form the primary key there. For illustration, see the figure 4.12 below.

[70] iheavy.com/2013/06/13/how-to-optimize-mysql-union-for-high-speed
[71] hibernate.org/orm
[72] en.wikipedia.org/wiki/Primary_key
[73] a foreign key to entity table's **id**
[74] **char(2)** type foreign key to a languages table

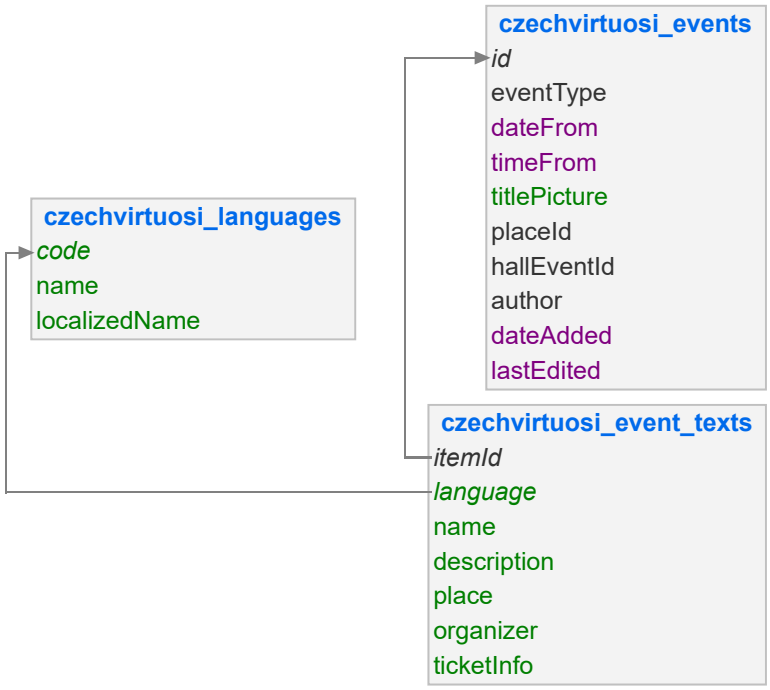

Figure 4.12: Part of the database schema showing the relations of an example entity and text tables. (created with Adminer)

4.6.4 Schema

Figure on the following page shows the final database schema. Some changes are noticeable from the entity-relationship diagram (see fig. 3.2), as the requirements were constantly changing during the design stage (see, for example, section 4.8). For legibility purposes, the schema is missing a few tables (see the caption) and all attributes are hidden.

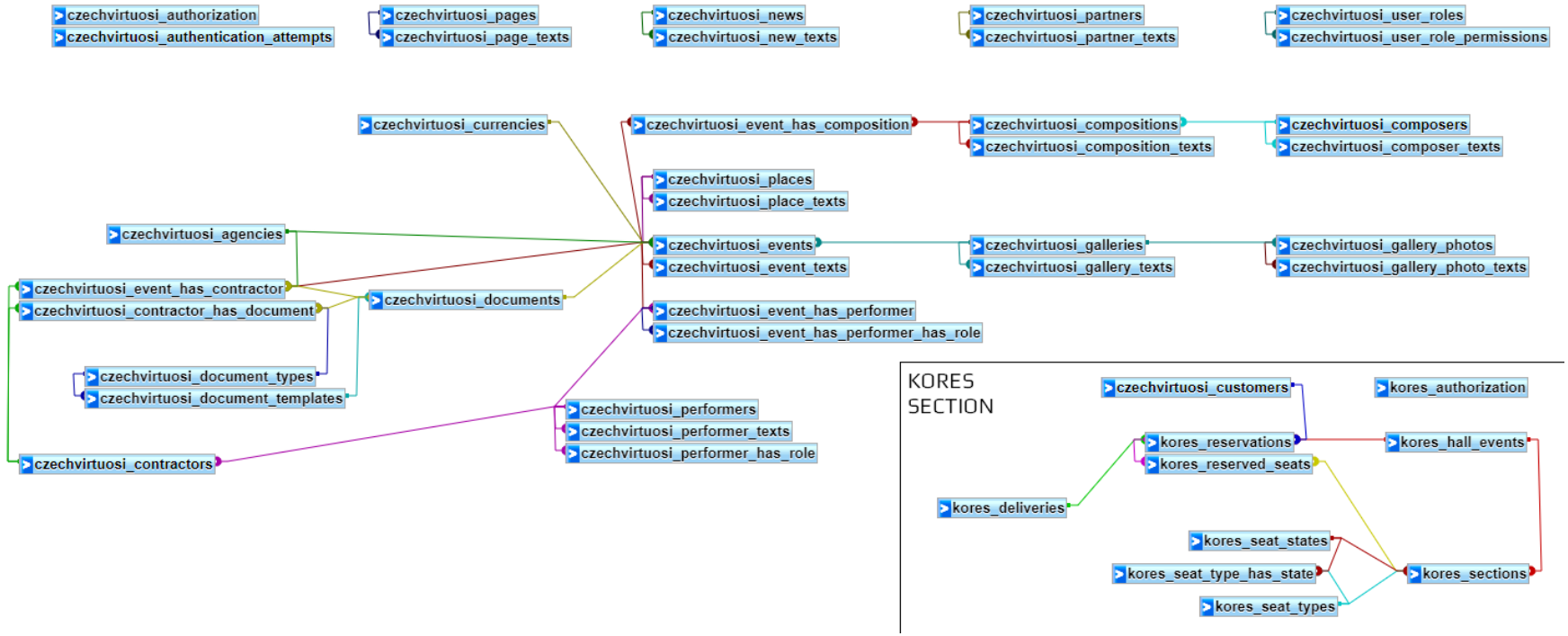

Figure 4.13: Final database schema. Tables czechvirtuosi_languages, czechvirtuosi_users and code-list tables are missing, because they all have a large number of relationships.

4.7 Back-end REST API

4.7.1 API framework

KORES API, which will be fully integrated into this API, is using the Restler framework[75] [10]. It will be effective to use it again, as KORES models and presenters would not need to be rewritten. However, the KORES API is thin consists of just several classes, so I was still looking for potential better alternatives.

To pick a new API framework, I have considered simply using Symfony[76] or Nette[77] frameworks with some third-party plugin, to keep it lightweight. As it turned out after some experiments, it is clear that the Restler framework is already very lightweight and also very purpose-built, as it offers many features for API design, such as custom method annotations, that can be parsed and evaluated before the request is processed. This feature is going to be very useful in implementing user permission enforcement, so I have decided to use this framework again.

4.7.2 Architecture

The API structure will be consisting of:

- **Presenters** — the classes that are modelling API endpoints, that provide CRUD[78] operations. Every endpoint will be matched to exactly one presenter. Every presenter will have standard methods **index**, **get**, **post**, **put** and **delete**. These methods will be mapped to respective HTTP methods of that endpoint, though not every one has to be implemented (in that case, it must return **501 Not Implemented**). The **index** method is for HTTP GET requests, where there is no ID specified (e.g. `GET /event /3` is processed by **get** method, but `GET /event` by **index** method). All presenters will extend a BasePresenter class, that will provide default method implementations (e.g. data validation), and most importantly, will always authorize the user, prior to fulfilling the request.
- **Models** — the classes representing the entities and providing operations on their data. Every entity will have a model. Presenters will depend on models and call their public methods, no model will depend on any presenter. All models will extend a BaseModel class that will provide default implementation of methods like **getList**, **getDetail**, **insert**, **update**, **delete** and so on.
- **ApiException** — a special exception class that expects an HTTP error code and a message, and upon throwing, returns this status code and message as a response and exits script execution (if not caught by an upper try-catch block).
- **Templates** — template files for use as tickets, complex documents or just e-mail notifications.

[75] luracast.com/products/restler
[76] symfony.com
[77] nette.org
[78] en.wikipedia.org/wiki/Create,_read,_update_and_delete

- Configuration and bootstrap files.

See fig. 4.1 for a simple visual representation of presenters and models in the context of the whole system.

The API will also make use of dependency injection[79] provided by Nette DI[80] service. This will enable all presenter and model classes to explicitly state their dependencies as arguments in their constructors. The DI service will automatically determine an order in which all classes must be instantiated so that every constructor can be called with already instantiated objects. All presenter and model classes will always have only one instance, though it is not the common Singleton[81] design pattern [5].

4.7.3 KORES integration

All KORES models and presenters will be included unchanged, but their names will be prefixed with **Kores** (e.g. original **Reservation** class will become **KoresReservation**). Some changes will then be performed, so that KORES models can also extend the new BaseModel and presenters the BasePresenter. Also, KORES models must be edited to reflect database changes (see section 4.6).

In the future, some dependencies will be formed between system models and KORES models (namely the reservation model), but again, nothing will depend on KORES presenters. Dependencies of KORES presenters on system models are possible, but should be avoided where possible, as this would greatly decrease KORES independence (future deployments of KORES in other systems should be always possible, it must stay an embedded module).

See fig. 4.1 for a simple visual representation of relations between system and KORES presenters and models.

4.7.4 Simple item listings

In order to provide listings of values such as countries, concert types, musical instruments, etc. (so-called code-lists) for all entities, the application will need to provide a way to persist listing types and their items (different values for each language). They will be (for now) statically defined in the database. It will be possible to make them user-editable (in CRUD manner, just like normal entity entries) anytime in the future (see section 7.3).

4.7.5 Content localization

As mentioned in the front-end localization design (section 4.5.12), all data requests and responses will always send localized fields in all languages at once. This means that, for example, when an event is downloaded, standard entity data like the event date and time will be listed once, localized fields, such as event name and description, will be listed in all language variants. The same applies for data inserts and updates.

[79] en.wikipedia.org/wiki/Dependency_injection
[80] Dependency injectiondoc.nette.org/en/3.0/dependency-injection
[81] https://en.wikipedia.org/wiki/Singleton_pattern

In order to implement this feature in the most automated way (to reduce costs and follow the DRY principle[82]), every model should simply have an array of localized fields as a property, and all methods should take that into account. A new class **BaseInternationalizedModel** should be created, extending the basic **BaseModel**, to provide all method implementations with localization in mind. All models, whose entities have at least one localized attribute (and therefore a text table in the database; see section 4.6.3), will then extend **BaseInternationalizedModel**.

Example

To describe how this abstraction will work, follow this example. **Event** is an entity class extending **BaseInternationalizedModel** and has a property **translationAttributes** with a value of `["name", "description"]`. **Role** is a standard entity class extending the **BaseModel**. When the method **getList** gets called on both models, the following happens. Role model simply translates the call to an SQL query like `SELECT * FROM czechvirtuosi_user_roles` and returns the result. However, the method in the **BaseInternationalizedModel** first reads the **translationAttributes** property, derives the text database table name, then loads all languages, and (in case of a Czech and English languages present) forms the following SQL query with a JOIN clause for every language:

```
SELECT
    t.*,
    cs.name AS name_cs, cs.description AS description_cs,
    en.name AS name_en, en.description AS description_en
FROM
    czechvirtuosi_events AS t
LEFT JOIN czechvirtuosi_event_texts as cs
    ON cs.itemId = t.id AND cs.language = "cs"
LEFT JOIN czechvirtuosi_event_texts as en
    ON en.itemId = t.id AND en.language = "en";
```

Code sample 4.14: Final SQL query for localized entity data extraction (identifier escaping back-ticks removed for better legibility).

4.7.6 Enforcement of user permissions

User permissions must be validated and enforced with every method of every single endpoint. If a single one is left out, it opens a door for attackers (see more in section 4.3.7). It is therefore necessary to use a way of permission description, that is straightforward, explicit and clearly visible when inspecting presenter methods. As the Restler framework supports custom annotation (see section 4.7.1), this looks like a perfect solution.

After trying out the possibilities, it can be concluded, that it is feasible to use method annotations to enforce permissions and it is the best solution. As most presenters will be

[82] en.wikipedia.org/wiki/Don%27t_repeat_yourself

dependent on one main model, many of the annotation can be auto-wired to use permissions of the model's entity. However, it must always be possible to customize the permission.

A special **AccessControl** class will parse the annotations[83], validate, if the user has all required permissions and abilities and if not, exit with a **403 Forbidden** response.

Following figures show the designed way of annotation usage.

```
/**
 * @access protected
 * @class AccessControl {@requires canInsert}
 */
function post($request_data = NULL) {
    $data = $this->model->extractModelAttributes($request_data);
    $data["author"] = $this->authUser->id;
    return $this->model->insertAndGet($data);
}
```

Code sample 4.15: Example **post** method that requires the **canInsert** ability of the inherited (taken from $this->module) permission.

Although this automated way will be the most common one, it can also be explicitly specified for which permission the ability is necessary. Also, more abilities can be chained.

```
/**
 * @access protected
 * @class AccessControl {@requires
       canRead performers,
       canRead events,
       canRead reservations,
       canUse someExportFeature
  }
 */
function index() {
    // export some data from 3 models
    $performers = $this->performerModel->getList();
    $events = $this->eventModel->getList();
    $reservations = $this->reservationModel->getList();
    return $this->createExport($performers, $events, $reservations);
}
```

Code sample 4.16: Example chaining required permission abilities in a presenter annotation.

Using this designed way of permissions description, it will be easy and straightforward to protect all endpoints (missing annotation would be clearly visible and when the permission name is inherited, it is impossible to mistakenly supply a bad one).

[83] will get called by Restler

4.7.7 Reservations

Reservations will be created from the KORES embedded application, that will run inside the orchestra website and will directly communicate with this API. Customers will select seats in a KORES-configured hall that will be linked to an event (see fig. 3.2), supply their personal data, consent with TOS[84] and GDPR data processing and place a reservation. The reservation process is fully documented and described in the original KORES thesis [10].

Reservations will contain all personal data and will always be linked to the customer that created the reservation. Upon placing a reservations, the system will check whether a customer with that e-mail already exists and eventually create a new one.

Setting a reservation as paid will trigger sending a successful payment notification to the customer's e-mail.

There are several types of delivery the customer can choose from:

- **Czech post (pre-paid)** — The customer pays the amount via bank transfer and then has printed tickets delivered to his address. Choosing this delivery adds 50 CZK[85] to order price.
- **Czech post (COD[86])** — Printed tickets are sent immediately and paid on delivery, but the customer is charged extra 100 CZK.
- **Box office** — Paid via bank transfer and the customer gets the tickets in the box-office directly at the venue. He just needs to arrive ahead of time to ensure picking tickets up before the performance starts, however, this option is free.
- **E-tickets** — Right when the payment is credited, the customer automatically gets an e-mail with e-tickets in a PDF, that can be either printed at home, or just shown on a display at the entry. Also free of charge. Certainly the best option for customers with a printer or a smart device.

Possible payment methods other than bank transfers, that are credited manually, are discussed in section 4.7.14. E-tickets are described in detail in the following section 4.7.8. No integration with Czech post is planned, this option of delivery is to be gradually eliminated in favor of e-tickets.

The specification further states (see section 3.1.3) that the system must be able to export reservations in a predefined Excel spreadsheet. For this task, I have decided to use a comprehensive and feature-rich library PhpSpreadsheet[87] that I have a prior good experience with. A special export endpoint will be devoted for this task and also a special feature permission (see section 4.7.6), so that it would be possible to give an external user reservation reading permissions, but prevent him from exporting them all in an XLSX file.

[84] Terms of service
[85] This is the current price, it can be changed in the administration anytime.
[86] Cash on delivery
[87] phpspreadsheet.readthedocs.io/en/latest

4.7.8 E-tickets

Generation

Tickets should be generated from a customizable template, stored in HTML (or some other templating language, like Smarty, Latte or Twig) for easy modifications. Final generation should always be to a PDF format because of its portability.

For the PDF generation itself, I have decided to use the PHP library mPDF[88], since it is very convenient to use, but mainly due to its good support of HTML and CSS [12] and that it is still actively maintained and improved (recently a new major rewrite, version 8.0).

Final ticket before being transformed to PDF will be always in HTML format (with simple CSS styles). Naturally, the first idea of templates was to use plain HTML with special placeholder variables like `#event_name#` or `#ticket_price#` that would get replaced by their values before further processing. Soon, it become apparent, that tickets will need at least some conditional logic (e.g. no row and seat number for halls that do not have seat labels), so these plain placeholder templates would not be feasible. It was eventual that a template processor[89] had to be used.

Template processor

To choose the optimal PHP template engine[90], I have considered 3 main candidates:

- **Smarty** (smarty.net) — due to prior knowledge and the fact that it is one of the first template engines around (developed from 2000).
- **Latte** (latte.nette.org) — due to prior knowledge and personal preference.
- **Twig** (twig.symfony.com) — due to its high popularity and usage (it is a part of the Symfony framework).

Since the template engine is going to be used rather at a small scale for a few templates (tickets and possibly some other documents), it is not so important to choose the absolutely best in the world, however it should be fast and convenient to use.

After finding out, that Smarty is not actively maintained anymore (just two simple commits in 2019) and it is also rather slow[91], I decided to remove this candidate.

In speed concerns, Twig and Latte are going to be similar. I did not run any benchmarks and also failed to find any, but both vendors claim, that they compile the templates down to a pure optimized PHP code with minimum to zero overhead. So the performance difference is likely going to be minimal, if there even is any.

In the matter of convenience, I have to say, that I personally find the Latte syntax more natural. It is generally more concise (e.g. `{/if}` vs `{% endif %}`) and uses the same

[88] mpdf.github.io
[89] en.wikipedia.org/wiki/Template_processor
[90] Template processors for PHP are often called „template engines“ by their vendors, so I will also use this naming from now on.
[91] reddit.com/r/PHP/comments/6nqord/twig_is_much_faster_than_smarty_in_2017

notations that are common in PHP code, such as dollar signs before variable names, keywords like `foreach` with unchanged syntax and the use of some native internal functions like `isset()` or `empty()`. Considering that, for someone already skilled in PHP, it would be easier to adopt Latte over Twig.

As I have already created some templates in Latte, it is a clear winner for template engine in this project.

Validation

There are a number of ways how to perform ticket validation. Although, not every one of them is reliable nor easy. Let's compare the options and choose the right one.

Bar codes can contain only numbers, so they are condemned to be used only with a specialized application.

QR codes can, on the other hand, contain a URL and with plenty of QR code readers for various device types currently on the market, a simple solution is showing up — a special URL address for each ticket that would just display the true owner. Some QR code readers immediately render the website title upon the recognition. If this title contains the information about validity and ticket holder, it might even be not necessary to visit the actual URL (in which case it would slow down the validation process).

A custom mobile application will always be ideal (offline validation, ticket counter, etc.), but this is for now out of scope of this project, however it might be a possible enhancement implemented in the future.

Until then, tickets will contain a QR code with a URL with a special endpoint that will provide the most important information in the page title.

4.7.9 VCF export

The specification states (see section 3.1.6) that for some entities, the system must be able to export virtual contact files (also known as vCards). For this task, I have decided to use an already-made package `jeroendesloovere/vcard`[92]. Detail-reading permissions will be sufficient to export vCards. Customers and reservations are currently the only entities where vCard export is feasible, but the application should be made so it is straightforward to add more.

4.7.10 Document generation

As decided in the specification analysis (see section 3.1.9), the system will support universal document generation from user-defined templates. As a template processor was already decided during the e-tickets design (Latte template engine; see section 4.7.8 above), the same will be used to process document templates. As with e-tickets, documents are also going to be processed to a raw HTML (but in three versions — page header, content and a page footer) and then converted to PDF using the mPDF library.

[92] github.com/jeroendesloovere/vcard

Some document templates are going to be very complex, for example, royalty liquidations will consist of long tables with a lot of conditional logic and altering headers and footers (separated lists of performers by agency). This document type will have an internal content template, that will not be editable by the user. However, headers and footers will still be editable. On the contrary, performer contracts will be rather simple — large portions of text the user can freely edit, complemented by occasional template tags.

4.7.11 File uploads

Many entities have files as their attribute. This might be either a document or more often an image (title or profile picture). In the client application, a special form component (see section 4.5.4) will take care of the upload prior to submitting the full form. These individual file uploads will be performed on a special file-upload endpoint. This endpoint will accept binary data (accepting base64 would waste 25% of the bandwidth) and return the URL the file was saved at. The standard entity update will be always done with the final file URL.

Files will be also saved in a special location, different than that of the application code. The reason is to enable for future load-balancing and eventual migration to a CDN[93]. All exported documents and tickets will also be saved at that same location.

4.7.12 File manager plugin

To enable file and image embedding into user-managed HTML content (see section 4.5.6), a TinyMCE-compatible file manager plugin has to be on the back-end of the application. Responsive Filemanager[94] was chosen for this task. It is completely free and has all necessary features. Integration is straightforward, because the vendor provides a ready-to-deploy back-end PHP package, that will handle all file operations. It will be however necessary to ensure proper request authorization, so that no attacker can misuse this file manager API. Authorization will be done using the same token as for the application API. A new database user **czvirtuosi_fm** will be created with read-only permissions for the token table. In the config.php file of the file manager, all valid tokens will be read and access granted only to requests with a valid token. No special file upload permission will be necessary.

4.7.13 GDPR compliance

To comply with GDPR a standard system function must exist, that would erase or pseudonymize[95] (in case that erasing might have adverse effects on system function) all sensitive data of a given person. This function will be supplied as a special red button in the detail view of each customer. It will pseudonymize the customer data and sensitive data in his reservations. In case of a future SmartEmailing integration (see section 3.1.10), it will also blacklist him in this service.

[93] en.wikipedia.org/wiki/Content_delivery_network
[94] responsivefilemanager.com
[95] en.wikipedia.org/wiki/Pseudonymization

4.7.14 Payment gate integration

As a nice-to-have function, the customer would like to have automated payments via a payment gate. The ideal one would offer several payment types, including bank transfer, and will make the end-user carry all extra charges (such as for a PayPal or credit-card payment). Payment confirmation would be automatic by a background endpoint callback. Such services are offered for example by the gate Pays[96].

However, the customer is not yet decided which gate to use, so this feature will probably get implemented sometime in the future.

4.8 Financial data

During the last stages of design and preparations for implementation, it started to become apparent, that event and performer entities will have to be split. This is due to the fact, that both entities will have to hold also financial and sensitive information. There are two main reasons, why it would not be appropriate to keep it all within one entity:

- **Role separation** — It is in the original specification, that some roles will be dedicated to maintain website content and some others to specifically read only the financial information (see original specification in section 2.1). This would be impossible if both types of information would be combined on one screen.
- **Clarity** — To manage all of the event or performer attributes in detail views, a large number of fields and tabs would be necessary. Also it could be unclear, which data are publicly available on the website and which are internal and sensitive information (e.g. contact information).

Because of the above mentioned, I decided to split both entities. But split how far? There is a number of ways how to split an entity:

- **Front-end-only split** — Only create two list and detail views, and distribute the attributes between them. There will still be one API endpoint providing and accepting all data at once.
- **Partial split** — A new presenter and endpoint will be created on the API part, however still one model (or two, when the new is extending the original one) and one database table. Both presenters would restrict access based on permissions for their sub-entity. For the front-end application, it would act as two separate entities.
- **Full separation** — A completely separated entity with own custom table. They are linked together with a `1:0..1` foreign key.

After careful consideration of implementation feasibility and consultation with the customer to better understand the real-world use cases and implications, I have decided to split both entities differently. Performers will be fully separated, and events partially.

[96] pays.cz

Performers will be kept as an entity used solely for the purposes of website content (they will be listed as performing, have their custom pages with their photos and biography, etc.). A new entity, contractors, will be created. All previously mentioned (see section 4.5.10) contracts, royalty liquidations and overviews will be linked to contractors. Any contractor can be linked to the performer he represents, yet there can also be contractors, that are not performers (any crew members, that might appear in contracts and liquidations, however, they are not performing directly). Contractor attributes will consist of private contact information and other sensitive data (such as passport ID) for the purpose of contract generation, so it will very useful, that it is possible to completely restrict access to them for any role and that they are being downloaded from a non-public endpoint (the standard performer endpoint will not be protected, as it must be read from the public website).

Events, on the other hand, are entities that are modelling a specific real-world event, that simply has both a public and a private part. Creating two entries and linking them together in this context does not seem reasonable. Therefore, events are going to be split partially, each with a custom endpoint (`/event` and `/event-financial`). The permissions will also be split, so it will be possible to control access to event financial data. The front-end-only split is not feasible, it would be impractical to handle permission access and it would break many of the detail view auto-wiring.

Both event endpoints are going to provide the same data. They will both load entries from the event table, and if there is a missing permission for the **eventFinancial** entity, they will drop all financial attributes. It has to be this way, because the data storage on the front-end must hold full entity entries, not only partial, otherwise it would have to merge **event** and **eventFinancial** entries together, when they are sequentially downloaded.

Chapter 5

Implementation

This chapter describes the implementation of whole application from the prototype created in the previous design stage. It is divided to describe the gradual development of the front-end and the back-end part individually. They are described after each other, however, they were developed concurrently. The end of the chapter describes the deployment of the final application. Testing is covered in the following chapter. Some parts of the implementation, that are unimportant or otherwise not noteworthy (e.g. implementations of every individual view, parts that are already covered well in the design stage, helper functions and components, etc.), are omitted.

5.1 Front-end application

To implement the front-end application, I started by using the prototype from previous chapter as a springboard. As all critical technology decisions were already made and the prototype consisted of concepts of feasible solutions, the remaining work was to just complete the implementation. This section is a walk through of that process.

5.1.1 Application skeleton

After cleaning up the prototype from unused drafts and packages, I prepared it for implementation. During that, I had to decide the final directory structure, that was to be followed. It is an important task, because a poor design would lead to subsequent refactoring and wasting time. Luckily, it was not the first front-end JavaScript application with these technologies that I designed so I settled with some patterns that I know are working well.

Following figure 5.1 describes the most important parts of the final directory structure.

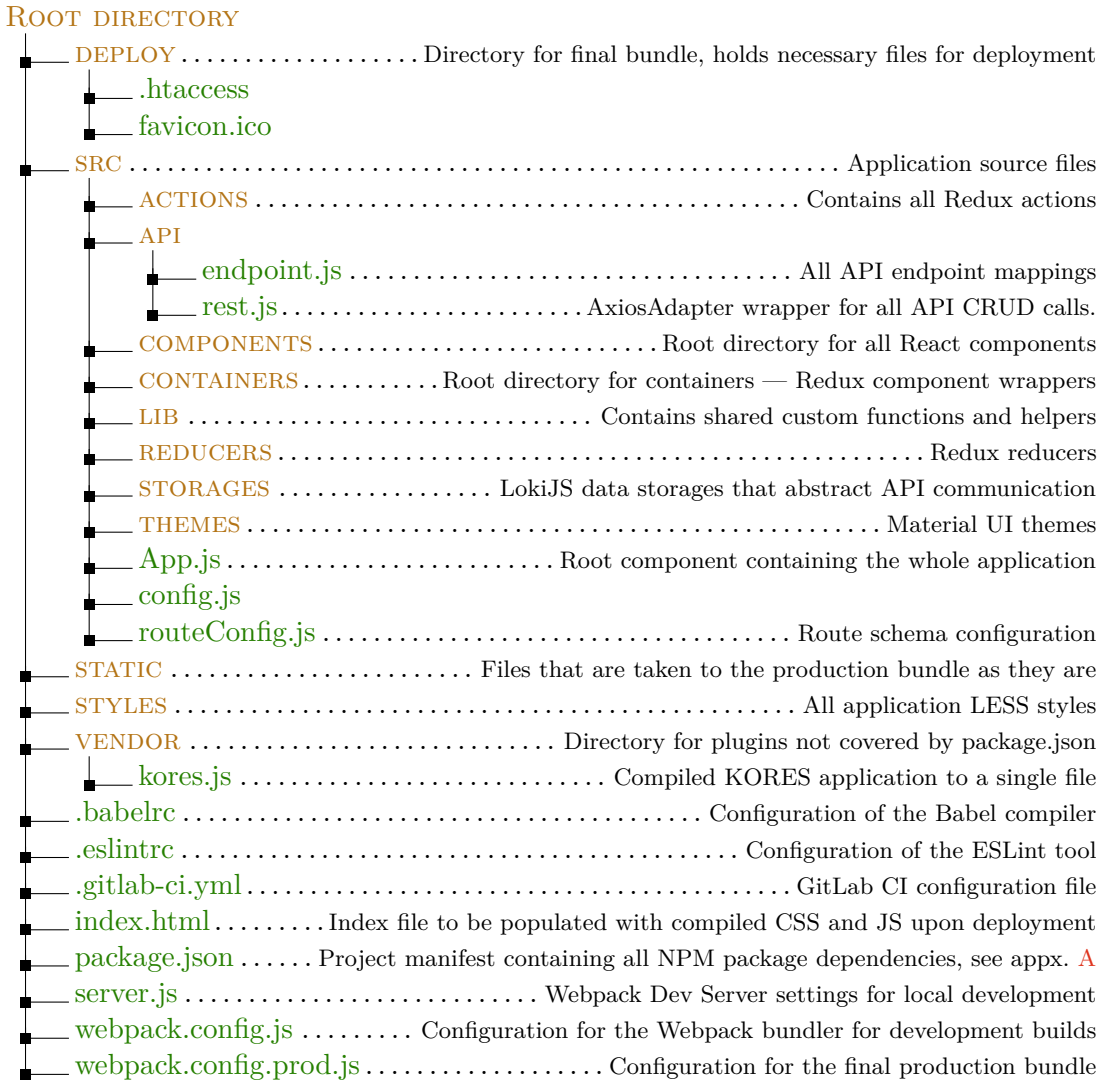

Figure 5.1: Front-end application directory tree.

5.1.2 Theme

It can be seen in the design prototype already, that the theme of the application is dark, with an accent of a light brown color for some important elements. This color scheme is derived from customer's new brand identity and will be applied across all new applications. Changing the look of Material UI components was simple, because they all use styles defined by a central color palette. The rest of the system, though, had to by styled manually (with LESS styles).

5.1.3 Data local storage

To start, I have decided to create data stores first. They are going to be the gateway to obtaining any data, so that should be implemented first, even if the data was to be mocked.

So that other components can be developed with at least a simulation of later usage with production data.

In order to name the classes to best describe their real function, I have decided to call them repositories, as this seems to be an established name for this task[1]. Every entity has its own repository (e.g. EventRepository), and all of them together form a data storage. Each repository is automatically mapped to an endpoint (derived from the class name) and extends a base Repository class with default implementation of the following methods (some minor methods omitted):

- **constructor** — base constructor that uses the class name to derive an endpoint name and saves an Axios[2] instance to a class property **restAction**, so all API calls from that instance can be performed like `this.restAction.post(data).then(response => {})`.
- **getOne** — retrieves a single entry from the store based on given condition, otherwise the default value provided as second argument. Triggers data download if not yet happened (explained below).
- **get** — reads all data satisfying a given condition, sorted in given order. Also triggers download.
- **getDetail** — requests detail data for a given ID. If not yet downloaded, will trigger download of this one entry.
- **massDelete** — deletes multiple entries defined by an array of IDs on the API and after a successful response also locally (so-called pessimistic update). Accepts a callback to execute after the local update. Dispatches the **dataUpdated** action.
- **update** — updates a single entry on the API and then locally. Also accepts a callback to execute after the local update. Dispatches the **dataUpdated** action.
- **localUpdate** — only updates local data in the store and also dispatches the **dataUpdated** action.
- **insert** — inserts a new entry on the API and then locally. Also accepts a callback to execute after the local insert. Dispatches the **dataUpdated** action as well.
- **downloadData** — directly download all list data. Cannot be called multiple times when the download is not complete. After persisting the data dispatches the **dataUpdated** action.

These are the base implementations of all basic repository methods. Any repository can override them. All data operations are well abstracted with this implementation, so it is really straightforward to work with application data anywhere in the app.

A special **FileStorageRepository** is used to upload files. It uses FileReader API[3] to decode the file to an array buffer and then uploads it to a special file upload endpoint using a binary octet-stream (see back-end design in section 4.7.11).

[1] en.wikipedia.org/wiki/Information_repository
[2] github.com/axios/axios
[3] developer.mozilla.org/en-US/docs/Web/API/FileReader

Data download triggering

The first component in the application, that requires data from each Repository instance, will trigger an actual data download from the API. That will take a while, so the Repository will return null or a provided default value (like as if nothing was found), because right as the download finishes, all components will receive a Redux update (**dataUpdated** action) and ask for the data again. This time, and anytime later for the same entity, the call will be satisfied from the cache. Detail data items are however loaded individually, because due to their size (HTML content, etc.), they are not downloaded in bulks. See fig. 4.2 for illustration.

5.1.4 List views

List views were implemented more or less just as designed in the prototype (see section 4.5.4). As they load data directly from the above described repositories, all operations are (after the initial short download) instant. Especially full-text searches, sorting and paging might seem like are using some kind of desktop application, that has all data off-line. This feeling is actually justified, as the application is running completely in-memory, so it nearly is a desktop application (if just bundled with Electron[4], it would truly be). A demonstration in below figure.

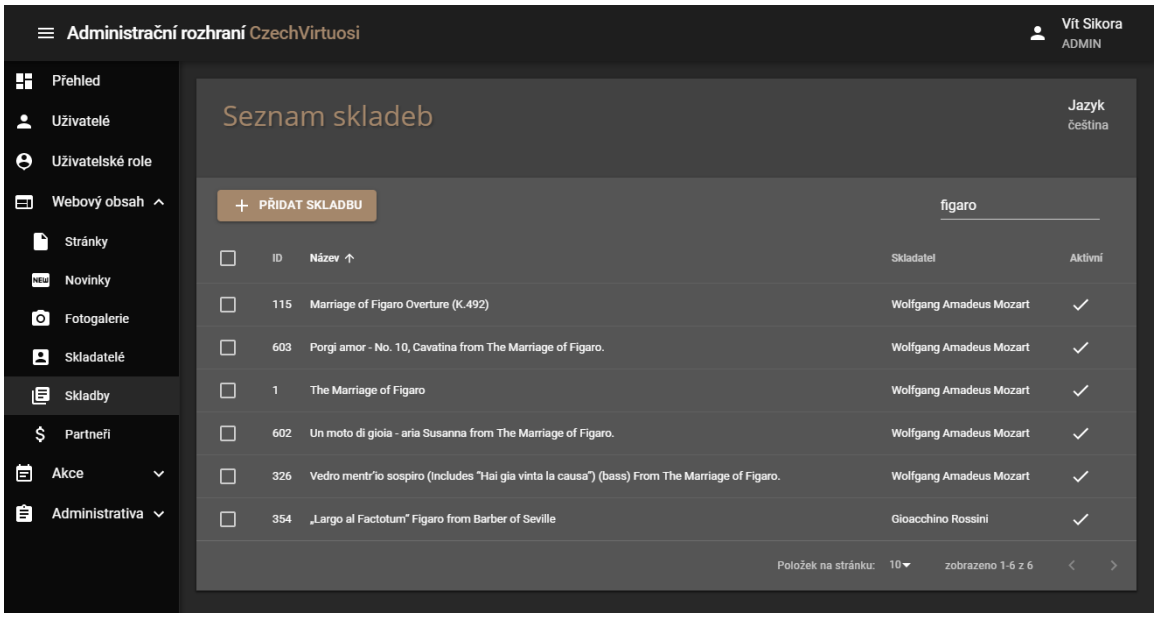

Figure 5.2: A sorted list view of compositions with search for „figaro“

The list view in the figure (CompositionList) is extending BaseList and overriding these two methods:

- **getColumnData** — standard method that returns list column definitions. Every List must supply its own.

[4] electronjs.org

- **getDataset** — the method that reads data from a corresponding storage. In this case an override was necessary in order to supply the name of the composer of each composition (otherwise there would be just IDs).

5.1.5 Detail views

Similar to list views, detail views were also implemented not much differently from the designed form. The implementation of all form and tab components was rather a long and exhaustive task. The following screenshot shows the full-featured version of event detail view.

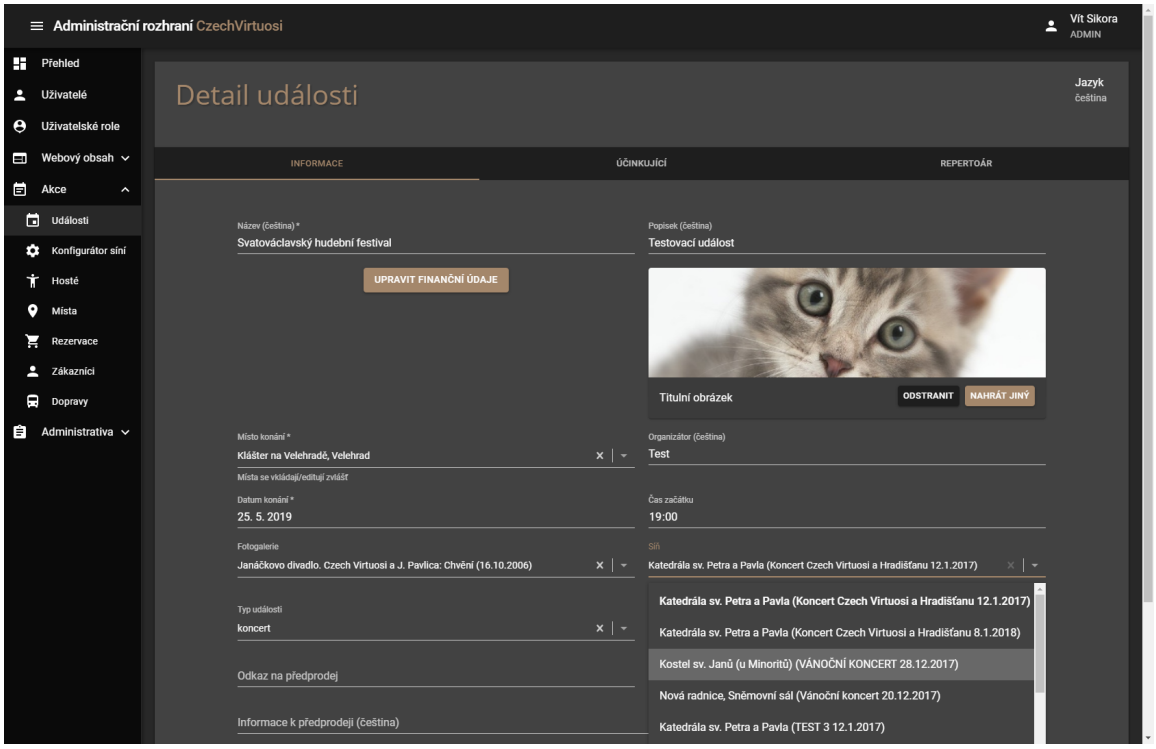

Figure 5.3: Event detail view example

In the bottom right corner, a selection of KORES-created halls to link to this event can be seen. The image component on the right supports drag&drop and internally immediately uploads images using the above mentioned FileStorageRepository. The form value for the image is always the final URL. Date and time pickers are used from the package Material-UI pickers[5].

Localization

It can also be noticed (in figure 5.3 above), that some of the form fields have an input language in parenthesis appended to their descriptions. This input language is controlled globally by a switch component in the top right corner. This way, the fields that this is

[5] material-ui-pickers.dev

applicable to, can be filled with different values in different languages. All other fields have their value shared across all languages.

Content editing

Rich text content editing is done by the TinyMCE plugin accompanied by a filemanager plugin for file uploads and embedding. A custom skin was created for TinyMCE to better fit in with the rest of the system (the default skin is pale white) as well as custom content styles to match the theme of the target environment. Following screenshots show the TinyMCE editors look with different content styles, depending on whether their target is the website (which also has a dark theme) or paper.

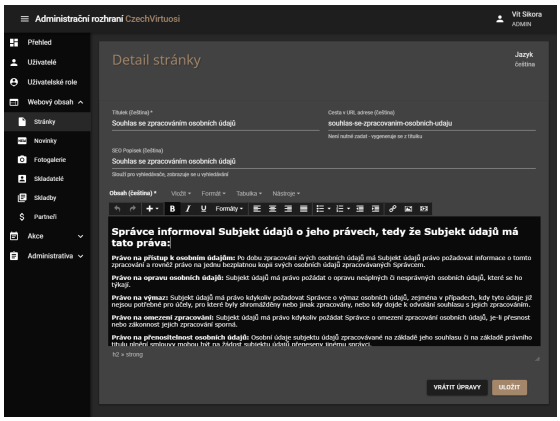

Figure 5.4: Page detail view editing the GDPR data processing page.

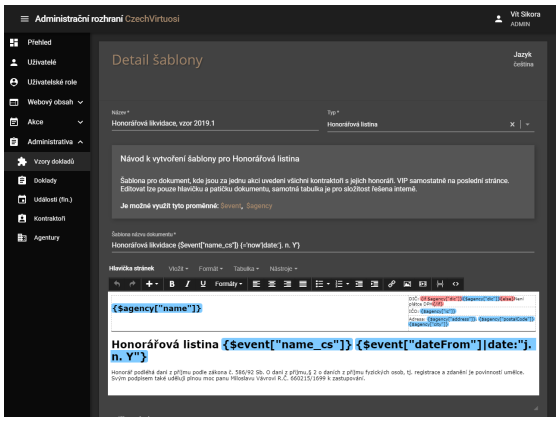

Figure 5.5: Document template detail view editing a royalty liquidation template.

On the right, in figure 5.5, it can be noticed that the content has a highlighted Latte syntax. This feature is described later in a dedicated section 5.1.9).

In order to allow users to embed any image or video, or link any file for downloading, a Responsive Filemanager plugin is fully integrated with TinyMCE and the back-end API (see section 4.7.12). On the front-end, the plugin runs in a pop-up window, displaying PHP-generated content from the back-end part in an iframe. It allows to upload files in bulk, organize them into directories, or even edit photos in a built-in photo editor. See the screenshots below.

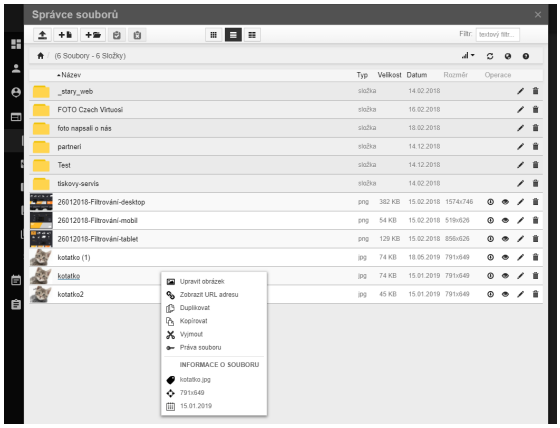

Figure 5.6: Responsive filemanager — directory listing.

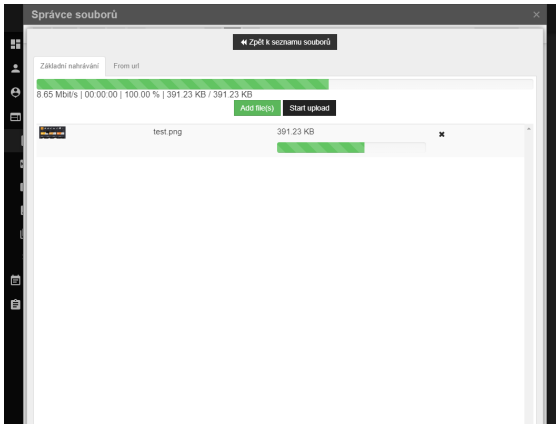

Figure 5.7: Responsive filemanager — bulk drag&drop uploads.

5.1.6 Routing

In order to keep a high level of integrity and maintainability, everything related to routing (including access restrictions mentioned later) is generated from a configuration file **routeConfig.js**. Here is a commented excerpt from that configuration to illustrate how it works.

```
const routeConfig = [
  {
    route: '/', // the URL this is effective for
    exactMatch: true, // otherwise would match all subroutes of "/"
    menuName: 'Přehled', // name in the left drawer menu
    menuIcon: (<DashboardIcon />), // icon for the menu
    feature: 'dashboard', // requires canDisplay of a 'dashboard' feature
        permission
    component: Dashboard // component that is rendered in the primary view
  }, {
    route: '/users',
    menuName: 'Uživatelé',
    menuIcon: (<UserIcon />),
    entity: 'user', // requires an entity permission
    component: UserList, // this is a list view component
    detailRoute: '/user/:id?', // parametric route, the menu item is in
        active state even on the detail view, this is also used in the list
        view
    detailComponent: UserDetailPage // detail view component
  }
  // ... more entries
  ]
```

Code sample 5.8: Commented excerpt from routeConfig.js.

5.1.7 Photo galleries

To edit photo galleries for events, the application has custom detail view tab component called **GalleryEditor**. This consists of a several subcomponents. Although not covered in the design stage, the implementation was rather straightforward, as it was possible to re-use some already created functionalities. For example, the photo upload placeholder (see the figure 5.9 below, bottom right corner), can be used to add new photos (even more at once) in a drag&drop fashion. In the background, it is also using the already-created **FileStorageRepository**. Individual photos in the editor are represented by Cards[6], that can be sorted with mouse drag, deleted by clicking on that trash icon, or can have their caption edited. Upon clicking the pencil icon, the card expands to full width and enables editing. These captions are also localized, so they can be filled out for each language separately.

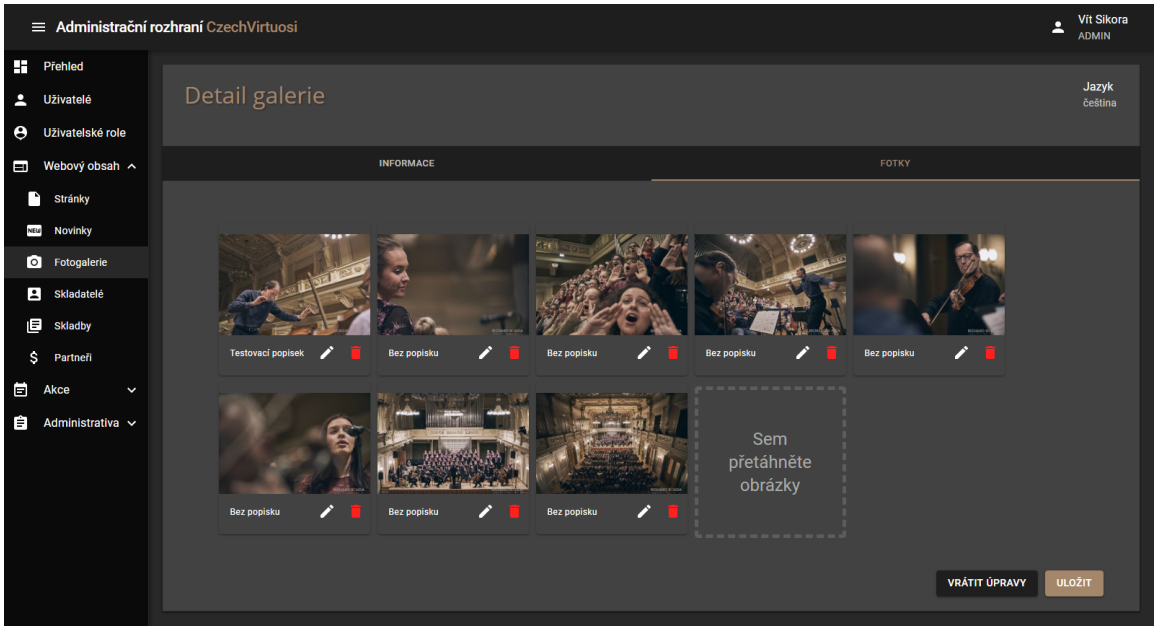

Figure 5.9: GalleryEditor detail view tab component editing a sample gallery

5.1.8 Role permissions enforcement

Role permissions were implemented according to the specification and design. A special detail view tab component is used to configure role permissions on a single view (see figure 5.11 below). These permissions are uploaded back to the API as an array of permissions where each one consists of boolean flags (one flag per ability).

To enforce the permissions, the recommended (see section 4.5.11) CASL library [11] was used. A CASL instance is created during the application startup, and upon downloading profile data (calling the **/me** endpoint; the first request when re-validating[7]) this instance is

[6] Material UI components, see material-ui.com/api/card
[7] see section 4.3.1, or the subsequent when authenticating

provided with the user's permissions[8]. Anywhere in the application it is possible to import the CASL instance (called **ability**), and check for any kind of permission simply by calling, for example: `ability.can("edit", "composer")` or `ability.can("use", "kores")`. It is also possible to use it as a standard React component, which is very elegant:

```
<Can I="delete" a="reservation" >
    <button> Delete this reservation </button>
</Can>
```

Code sample 5.10: Example of using CASL as React component.

The enforcement itself works precisely the same as was suggested in the design stage (see section 4.5.11), mostly because the library was already tested in the prototype on some test entities. An example can be seen on the following figure 5.12 with a detail view of a composer, where an explaining warning is present, inputs are disabled and the submit button is not displayed. The user's role in this case allows only **canEditOwn** ability but not the **canEdit** ability and the user is currently visiting a detail of a composer he did not create.

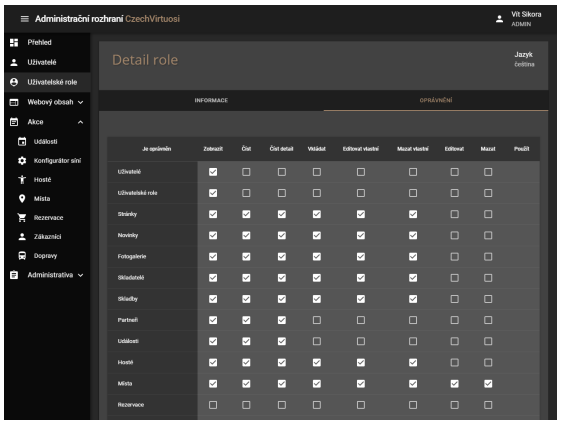

Figure 5.11: RolePermissions component — a matrix of checkboxes to form role permissions with allowed abilities.

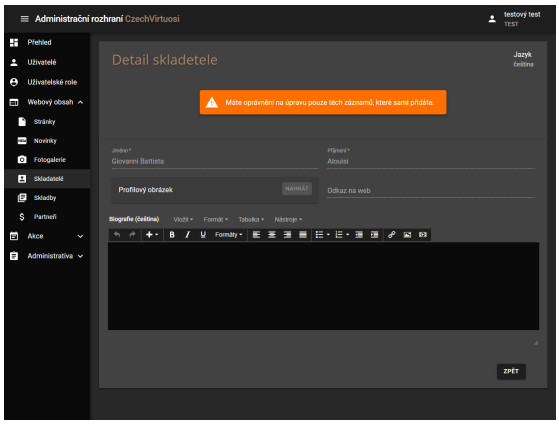

Figure 5.12: Example of permissions enforcement — not having a permission to edit this entry.

5.1.9 Documents

On the front-end part, the implementation of document generation was rather easy. The rich text editor needed custom content styles (so that the edited content looked similar to the final version — a PDF file). And also, as mentioned in the design, it was necessary to create a custom TinyMCE plugin to highlight document template tags (Latte syntax, see section 4.5.10). Without this, the templates looked disorganized, their legibility was severely hampered, and they were prone to syntax errors from user not skilled with using

[8]NOTE: Theoretically, it is possible to hack this procedure by, for example, intercepting the response inside the browser and obtaining all possible permissions. They are, however, primarily enforced on the API, so the attack would not help himself in any way.

Latte template tags. The following screenshot show a contract template with highlighted syntax.

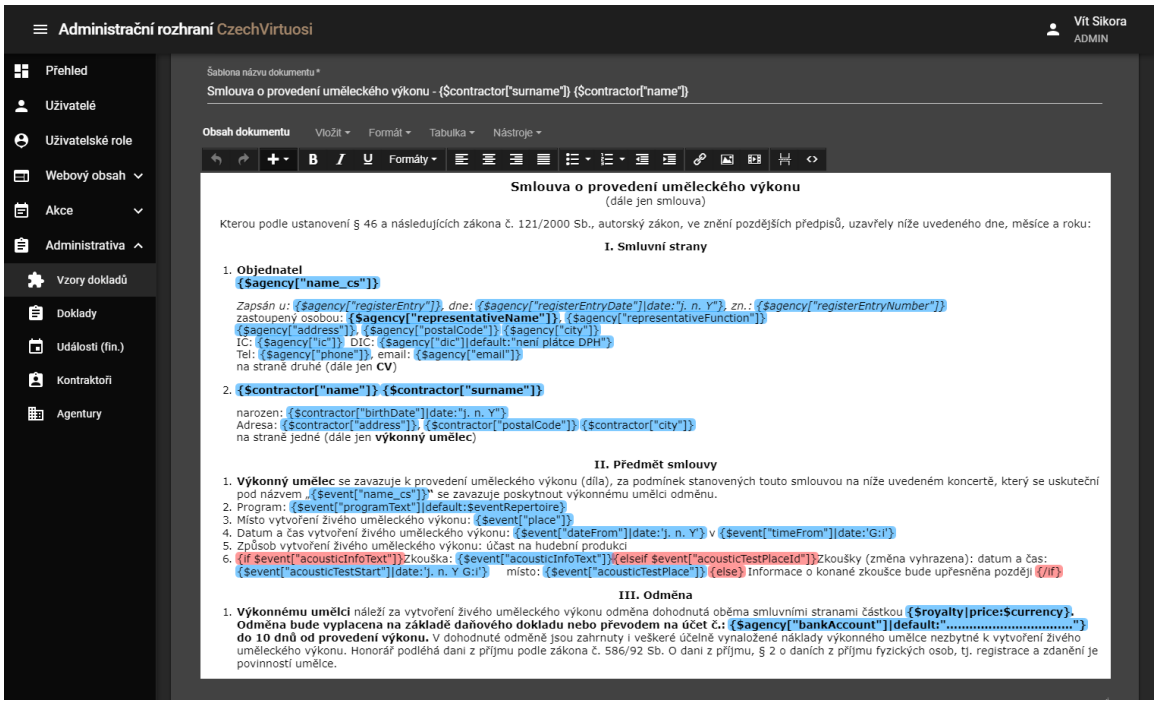

Figure 5.13: Example document template for a performer contract.

As can be noticed, standard variable outputs are highlighted in blue color and conditional statements in red. Some system users can be taught how to use these tags, while for others, it can be as simple as ordering them to not edit anything colorful.

Final documents are generated on special detail view tab components. For example, financial events (see section 4.8) have a tab for royalty liquidations (see figure 5.14 below), where it is possible to pick one of the user-created templates for royalty liquidations and generate the document. It is possible to view (opens in a new tab) or directly download the previously generated one (if such exists). Contractor royalty overviews are similar (see figure 5.15 below), apart from being able to set a specific year (by default the current year) of the overview and modify the date of document generation (additional customer requirement). Overviews are saved separately for each year.

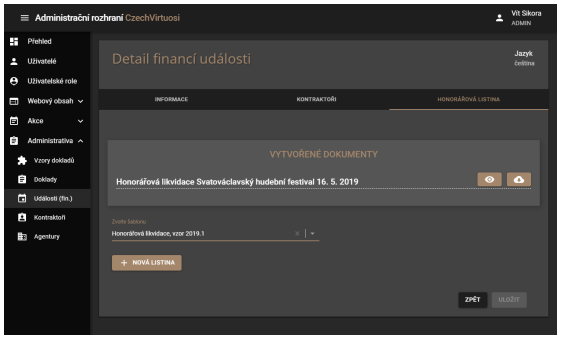

Figure 5.14: Example of generating a royalty liquidation for an event

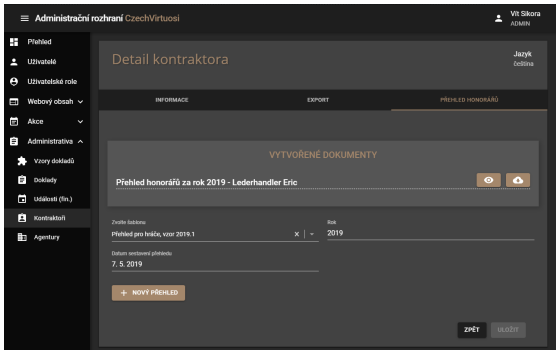

Figure 5.15: Example of generating royalty overviews for a contractor

5.1.10 Financial data

The split of event and performer entities was implemented as designed (see section 4.8). On top of that, special switching buttons were introduced to performer, contractor and both event detail views, so that it is possible to quickly jump between entering public website data and private financial information. If the user lacks the permission for the other view, he will not see the button. Also, if visiting a contractor detail page, that has no connection to any performer, no switching button will be available as well.

The following figure 5.16 shows a complex detail view tab component for assigning contractors, that will perform on an event. It is based on a regular entity relationship management component (see section 4.5.5), but extended by far to be able to manage the extensive relationship data between events and contractors. Each relation is complemented by financial attributes, such as the royalty, travel fee, an extra fee and their due dates. Every contractor is supplied by some agency, though it is often the same one, it must be possible to set it individually. Also, as can be noticed from the figure, every contractor performing at one event must have a generated contract. In the top section of the component, there is a selectbox for picking a template to use when generating the contracts. It must then be generated for each performer manually, however, it does allow to use a different template for each performer.

As the contractor card can get rather large in size and so with a higher number of contractors the view loses clarity, a checkbox filter at the top is there to improve this by condensing to view to only those inputs the user wants to work with at the moment. The top view also includes subtotals of the financial data.

If a royalty or an agency is not set, it is taken from the event, where default values can be specified. Currency and its exchange rate is also specified within the event attributes and are shared across all performers. Therefore, it is impossible to pay two contractors at the same event in two different currencies. This may be get implemented in the future (see section 7.3)

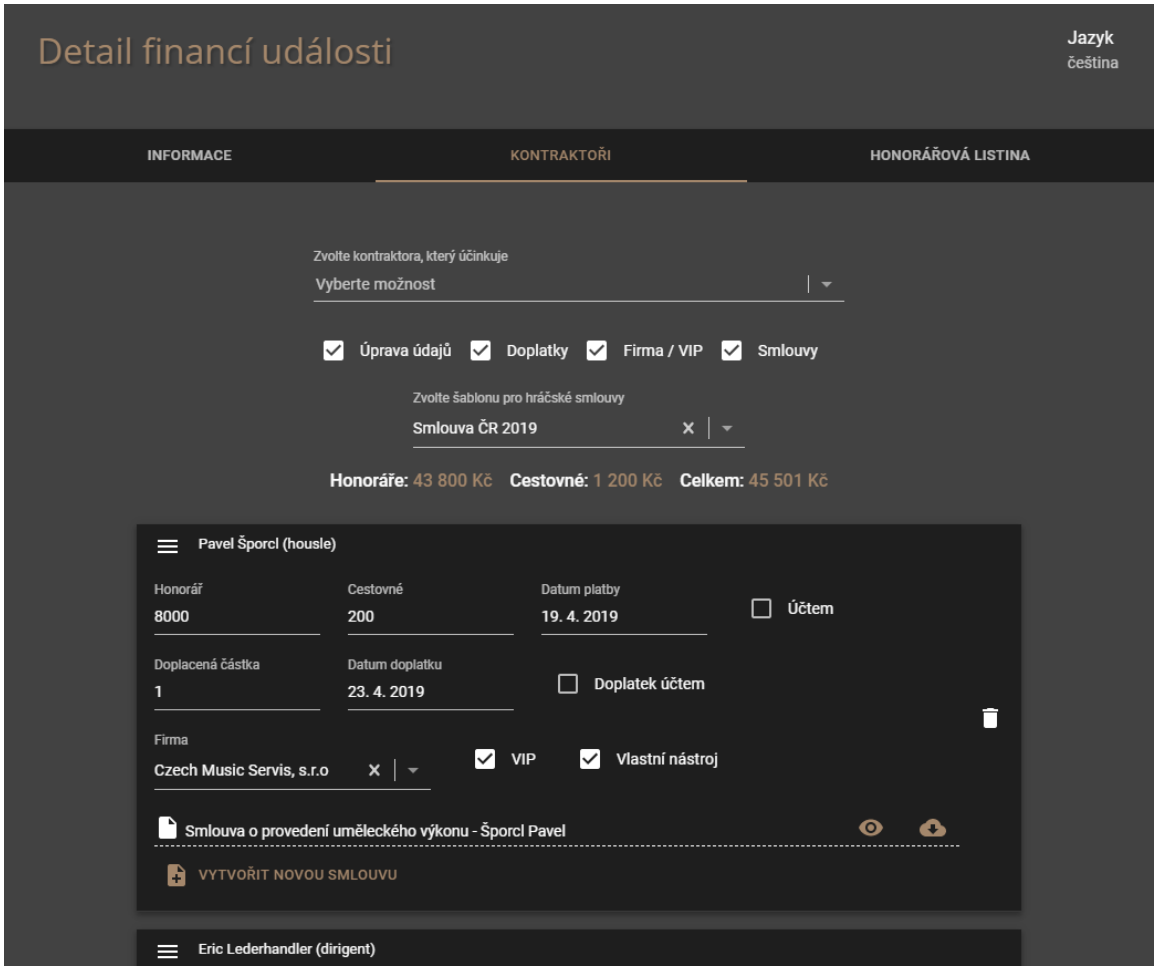

Figure 5.16: Example of assigning contractors that will perform on an event and managing their financial information and contracts. Note, that all data in the picture is for demonstration only.

5.2 Back-end REST API

To implement the back-end part of the application, I was also able to begin with building up on what was already created in the design stage. As technologies and frameworks were already decided, the development of this part consisted mostly of adding new and updating existing presenters and models, but sometimes also of implementing interesting and complex functionalities, such as, for example, e-ticket and document generation. These will be described in detail in the following sections.

5.2.1 Application skeleton

Similarly to the front-end application, it was vital to decide in the beginning, how will the project directory structure look like. As the technology selection was similar to the one used in KORES development [10], the structure was going to be similar, only with some

small improvements and additions, based on previous experiences. The final structure with notable files and comments follows in the figure below.

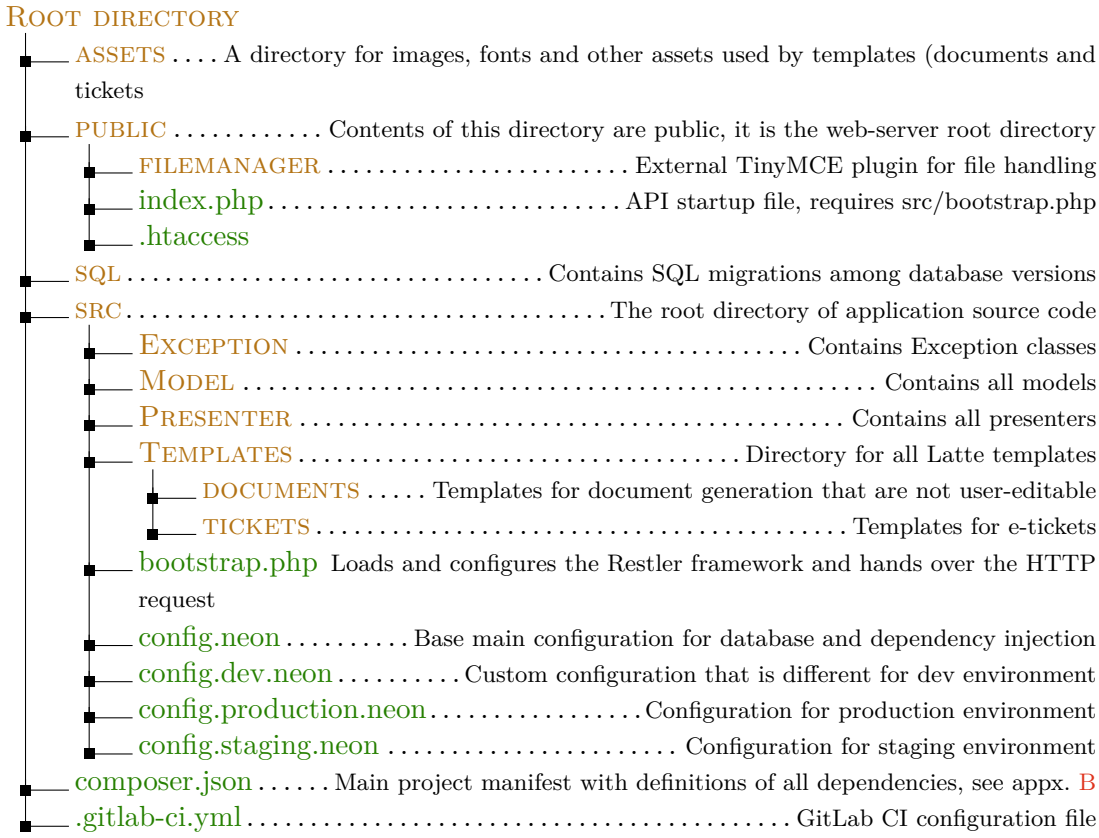

Figure 5.17: Back-end REST API directory tree.

5.2.2 Authentication and authorization

The authentication process is implemented exactly as planned in the design stage (see section 4.3.1) with the exception of KORES token generation. It was planned that these would be generated upon authentication, however, this would not cover token re-validations for long term tokens. As the system allows creating a 14-day token (if the remember login checkbox in the login screen is checked) and KORES only allows 24-hour tokens, this might induce some trouble. Therefore, it was implemented so that the KORES token is refreshed with every `/me` endpoint call (always called after login or in re-validation).

5.2.3 Architecture

The architecture of models and presenters was implemented according to the design (see section 4.7.2). Each entity model has several attribute enumerations in its properties:

- **detailOnlyAttributes** — a list of attributes that can only appear when obtaining detail data (see section 4.3.5) of a given entity. They are all discarded when loading a list view. Typically contains rich text or other data-intensive attributes.

- **userRequiredAttributes** — a list of attributes that are required when inserting or updating an entity. If one of them is omitted, the API refuses to fulfill the request.
- **userSuppliedAttributes** — complementing set of attributes to **userRequiredAttributes**, which together form a list of accepted attributes. During inserts and updates, any other supplied attribute is discarded.
- **translationAttributes** — for models extending **BaseInternationalizedModel** (see section 4.7.5). A list of attributes that are localized to all languages.
- **searchableAttributes** — for entities with public access, such as events, performers, composers, compositions, galleries, news or pages. These attributes are used in full-text data queries when conducting a search.

The API currently consists of 33 models, 42 presenters and 12 template files.

5.2.4 E-tickets

Upon receiving a new reservation or updating an existing one with a seat selection change and a delivery selection of „e-tickets“ (see section 4.7.7), the application generates a new set of e-tickets.

These are generated from predefined Latte templates. A blank PDF file is generated first, with a notice for customers, that they must either print the tickets or show them on a display at the entry. Then, for each seat, a ticket is added to the file. The dimensions of the ticket are 180 mm by 60 mm, in order to contain four tickets on an A4 page.

The following figure shows an automatically generated e-ticket. The design of this ticket is created by me, however future changes are possible if the customer does not like it. It could also be possible to have multiple ticket templates and choose one for each event. The QR code should still work and provide information about the ticket holder.

Figure 5.18: A generated e-ticket during reservation.

The QR code is generated using the library `mpdf/qrcode`[9] which has a good integration with mPDF. The code contains a verification URL, which is a call to a special API endpoint

[9] packagist.org/packages/mpdf/qrcode

(just `/t` to keep the QR code small) with a verification URL (see fig. 5.19 below). For each reservation and also each ticket, a unique random[10] verification code is generated. The ticket verification URL contains both codes, so upon validation, the validating person will see details of the reserved seat (row and seat number, section and floor) and also details of the whole reservation (overview of all reserved seats, details of the person that created the reservation, time of last validation of that ticket and reservation, etc.). This might be really useful, when a large group of people from a single reservation appears. The staff member could simply validate one ticket, check the total number of reserved seats and let the same number of people go.

Because the QR code represents a simple URL, any QR code reading capable application on any device type can be used to validate the tickets. The next figure is a screenshot from a mobile device, that validated the above ticket.

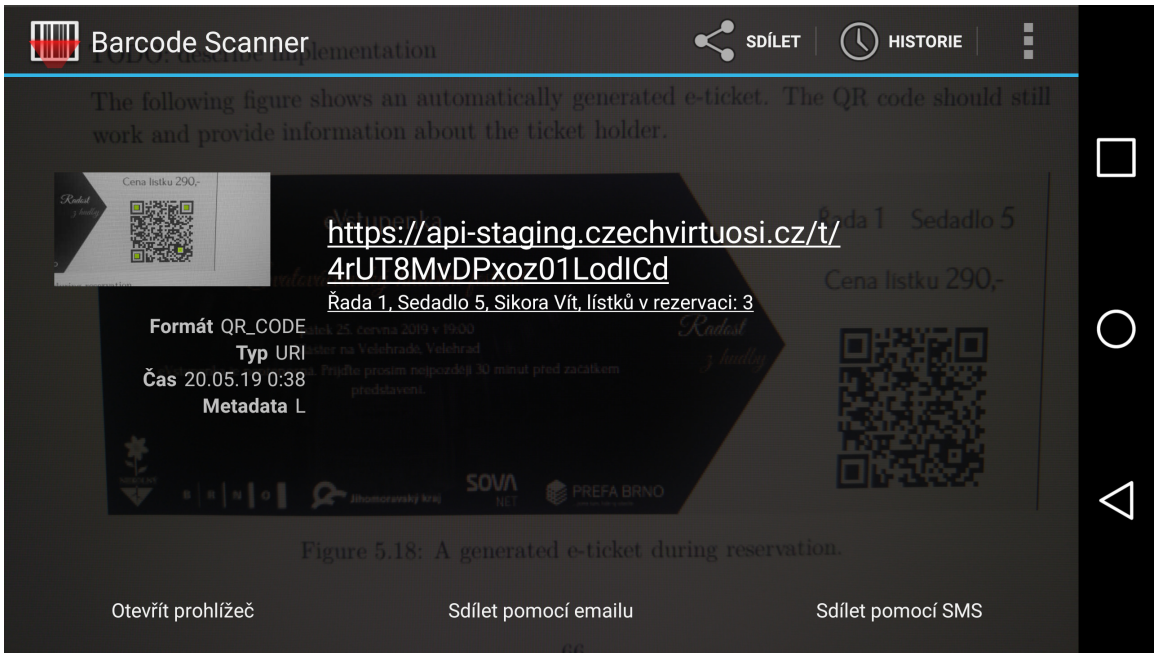

Figure 5.19: Example validation of an e-ticket QR code.

Some QR code reading applications, including the Barcode Scanner[11] for Android that was used in this case, display the title of the linked page. The application takes advantage of this and displays the most important information (seat info, customer's name and sum of seats in the reservation) in the title (can be seen in the screenshot just below the URL). If the validating person is still in doubts or wants to see details of the full reservation, he can still tap the URL and see it all.

In the future, a custom mobile application could be created for this task. This might bring off-line and quicker validations, while still being compatible with basic QR code readers.

[10] Cryptographically secure pseudo-random, see php.net/manual/en/function.random-bytes.php
[11] play.google.com/store/apps/details?id=com.google.zxing.client.android

5.2.5 Documents

As mentioned in the design (see section 4.7.10), some document templates are not fully user-editable (only headers and footers are). This is currently the situation for royalty liquidations and overviews, as both of these documents contain tables with lots of conditional logic and calculations (namely exchange rates). These templates are complex, and an average user would be unable meaningfully edit the template. For any modification, a developer will be necessary.

A special endpoint `POST /generate-document` is used to create new documents. A **templateId** parameter is mandatory to indicate the template and the document type, that will be used. Additional data, such as a year for a royalty overview, are supplied in the request body. The presenter also checks whether the user has enough permissions to read all the data that go into the document. Otherwise, returns **403 Forbidden** just like in other cases of missing permissions. Upon successful generation, a document entity including the URL of the created PDF is returned.

The generation process itself is carried out in the Document model, with a special method for every document. These methods typically extract a lot of data from other models, that is necessary for the template and then proceed with processing the templates to raw HTML. This is then converted to PDF. The Document model is dependent on 10 other models as a result.

Each document generating method has a unique implementation, as some documents may have alternating headers or some other specialty, that prevents from using a shared method that would just take the templates and generate the PDF.

An example of a generated royalty liquidation follows on the next page. It is only the first page, as it is a 4-page one. VIP contractors are always separated from others on a new page, and it is also split by agency (the page header with agency is different on pages 3 and 4).

Czech Music Servis, s.r.o

DIČ: Není plátce DPH
IČO: 27736784
Adresa: Lelekovická 256/16, 62100 Brno

Honorářová listina Svatováclavský hudební festival 25. 5. 2019

Honorář podléhá dani z příjmu podle zákona č. 586/92 Sb. O dani z příjmu,§ 2 o daních z příjmu fyzických osob, tj. registrace a zdanění je povinností umělce. Svým podpisem také uděluji plnou moc panu Miloslavu Vávrovi R.Č. 660215/1699 k zastupování.

Jméno, adresa	Datum narození	Honorář	Cestovné	Podpis
1. Amaryllis Grégoire Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
2. Ardašev Igor		1 000 Kč	0 Kč	
3. Ardaševová Renata Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
4. Barová Anna Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
5. Benoit Philippe Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
6. Bláha Vladislav Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
7. Bockstal Piet van Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
8. Bogza Anda-Louise Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
9. Broda Jan Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
10. Brožková Jana Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	
11. Bárta Jiří Testová 47, 12345 Testov	17. 4. 1963	1 100 Kč	0 Kč	
12. Englichová Kateřina Testová 47, 12345 Testov	17. 4. 1963	1 000 Kč	0 Kč	

Figure 5.20: An example generated royalty liquidation (all data in this document are example only).

5.2.6 Data import

All data were imported from the old system (see section 3.1.13) to the production version of the database. Because the new system was created without any backward-compatibility restrictions, it was not possible to use a plain SQL script to transfer the data. Also, many physical files, mostly pictures, had to be moved together (to keep all links to the files working). A special temporary presenter was created to perform this operation, so that the import could be controlled and debugged by calling an endpoint from Postman (see section 6.1.1). The import method consisted of almost 200 lines of code.

5.3 Deployment

The application is ready to be deployed using GitLab Continuous Integration[12]. CI configuration files are created both for the front-end and the back-end parts. During this deployment, both applications will use a Docker[13] image (to have a uniform and controlled environment), inside which they will have their dependencies installed, built and then deployed on web server. More deployment stages can be easily added in the future, for example, running unit tests (see section 7.3).

[12] about.gitlab.com/product/continuous-integration
[13] docker.com

Chapter 6

Testing

To ensure that the system was properly developed and is functioning correctly (and will be in the future), it had to be thoroughly tested. For now, only manual testing was used, however there are plans for extending the test procedures to include automated tests.

The second part of the chapter is devoted to user testing. It is not only important that the application works properly, but also, whether users can utilize this functionality and use the system as a whole effectively [8]. This was tested directly with guided and measured user scenarios.

6.1 Function testing

The application was tested for functionality on a set of real-world data (imported from the former system, see section 5.2.6), so the results should be credible. This section discusses test procedures used to test both parts of the system.

6.1.1 REST API

The Postman[1] tool is used for testing individual API endpoints, and verification of the underlying functionality. This is a specialized tool for this task, as it has a built-in support for endpoint collections, environment switching and using variables in URLs and request headers (so managing authorization is very easy).

It is still, however, manual testing. No automated tests (e.g. unit tests) are currently used, but they are planned in the future (see section 7.3), in order to secure a stable run of all critical procedures, such as the reservation process. They would run within continuous integration, so no participation from developers would be necessary to run them with every code change. They would stop the system from deploying upon failure.

The figure below shows an example of endpoint testing with Postman.

[1] getpostman.com

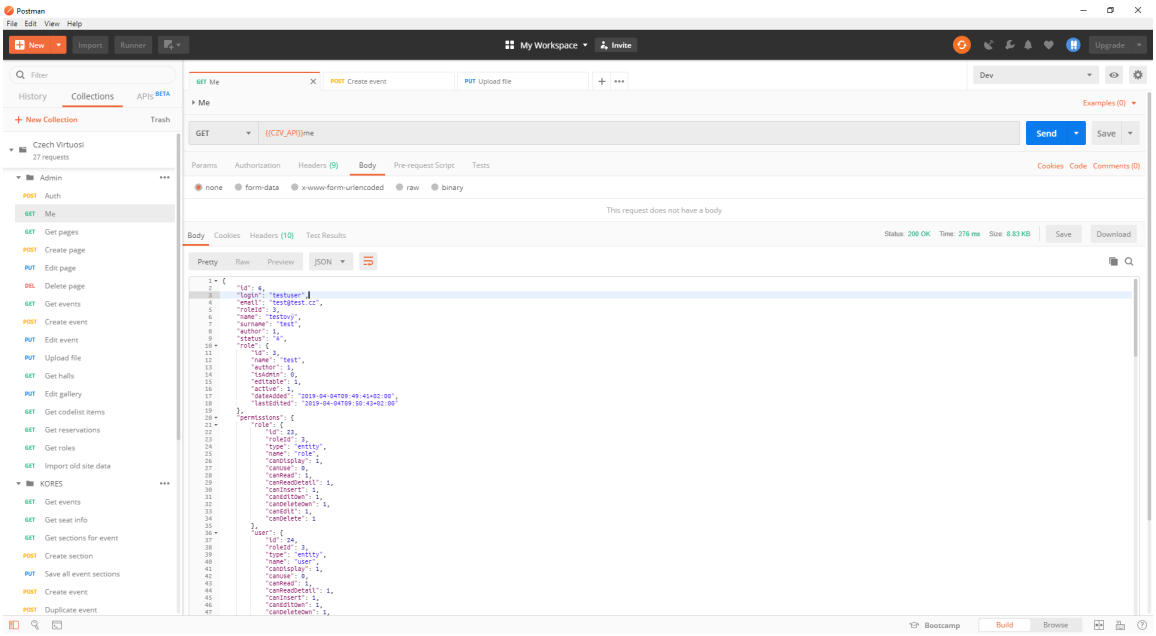

Figure 6.1: The Postman tool for API testing.

6.1.2 Front-end application

No automated front-end tests are used or planned, due to higher complexity that is necessary to set them up. They are also notably slower (they need to run in a virtual DOM), and might not be that useful without visual regression tests. Also, there are no critical front-end functionalities now. It is necessary to test the reservation and ticket generation procedures, as these are critical, but they run on the API only. Other problems can often wait, as it is a closed administration system.

However, to ensure that the front-end application works properly, every part of it was examined manually and all discovered problem were fixed.

6.2 User testing

It is necessary to test the application directly on end-users, because the developer, knowing the system well, will never be able simulate the situation of an unknowing user and experience the same obstacles [8].

The testing had a form of individual user interviews. In total, 14 users were interviewed. As the testing was conducted at the company I work for, it was a variation of a Hallway testing[2]. Even though I picked only people not knowing anything about the application, they were not a good population-representing sample, as most of them were of a young age and working in the IT sector. However, it was still more than helpful in the means of spotting gaps in the design.

[2] en.wikipedia.org/wiki/Usability_testing#Hallway_testing

6.2.1 User scenarios

In order for the tests to have quantifiable results, the first part of the interview was a sequence of measured actions, the user had to perform. For each task, I sat with the user, told him what to do, and then watched all his actions and attempts in silence, providing no further help. This way, I discovered some potential problems, when more than user got stuck in the same section.

1. Deleting a photo from a specific gallery

In this scenario, the instruction was to delete a single photo (any photo) from a specific photo gallery, that was taken in the city of Pozořice in the year 2006. This specific gallery was chosen, because due to its age it was buried down by more than 100 newer entries, and furthermore because the name of the city is unique among other galleries. This was to test, whether the users will notice the possibility of using a full-text search. Upon opening the detail of the gallery, the users had to switch to the tab, where they could edit the photos (this can be seen in the fig. 5.9) and then save the changes.

The average time to complete this task was **30.6 seconds**. Best time was **19 seconds**, and the worst **41**. The search feature was usually quickly discovered, by most users it was used as a first option, only some had discovered it after browsing through several pages of results. No user browsed all the the results manually.

However, as it turned out, the pain point was the tab selection on the detail view. Almost every single user had a hard time looking for the photos, probably thinking of the tab panel as of some kind of a design feature, separating the heading from the form. Eventually, everyone had successfully discovered it, but it was obvious, that it could have been designed better. Probably an indication similar to the one used by web browsers (active tab in the same color as the content) would help.

2. Granting a specific permission to a specific role

In this scenario, the users were asked to look up a specific role, and grant it a specific permission (performer editing ability). The reason was to test, how intuitive is the large checkbox matrix used to define permissions (can be seen in fig. 5.11), and how will the users feel comfortable using it.

The average time to complete this task was **29.4 seconds**. Best time was **18 seconds**, and the worst **51**. It is notable, that even if this task also required changing a tab on a detail view, now it was not an obstacle at all. Every user understood the layout of the detail view and got used to it after a single use.

Regardless, looking up a specific role was a slow task for all users. Though everyone quickly understood how the matrix works (5-15 seconds), approximately half of the users used a browser built-in search feature to find the correct row, and then they struggled finding the right column as the table was too big and the top row was sometimes not visible. In this situation, a floating sticky table header would provide aid. A search feature would also help in this task, but the matrix will be primarily used to have an overview of what one role can

do. One user had a problem understanding the difference between a user and a user role, but that would be difficult to improve with the design.

3. Resetting own password

In the final scenario, the task was to reset the password of the currently logged-in user to a specific value. This was to test, if it is well indicated who is logged in (in the top right corner), and whether the users will click on that indication. Moreover, it was also to find out, whether everyone will understand, that to reset the password, it necessary to just fill in the input and save the whole profile.

The average time to complete the last task was **25.1 seconds**. Best time was **12 seconds**, and the worst **52**. The disparity between the best and the worst time was the largest in this task. Most of the users (approximately 70 %) quickly peeked to the top right corner and immediately navigated to their profiles, but the rest struggled a lot. Some of them navigated to the list view of all users, but then did not know, which of them is the currently logged-in one. Eventually everyone fulfilled the task without help, but it was indicative of that the user zone in the rightmost part of the header might have been more contrast. After discovering the zone, no user hesitated to click it, probably because they are used to trust the meaning of the „pointing hand“ mouse cursor[3]. Also, no user had troubles with the final changing of the password.

6.2.2 Feedback

In the second part of the interview, the users responded to general statements about the application and gave feedback about what they thought might improve the usability. The answers to the statements were meant to summarize how the users felt when using the application. All 14 users participated in the survey.

For each of the statements, the users could answer that they either:

- strongly agree,
- agree,
- do not mind,
- disagree,
- or strongly disagree with the statement.

[3] en.wikipedia.org/wiki/Pointer_(user_interface)

Intuitive interface

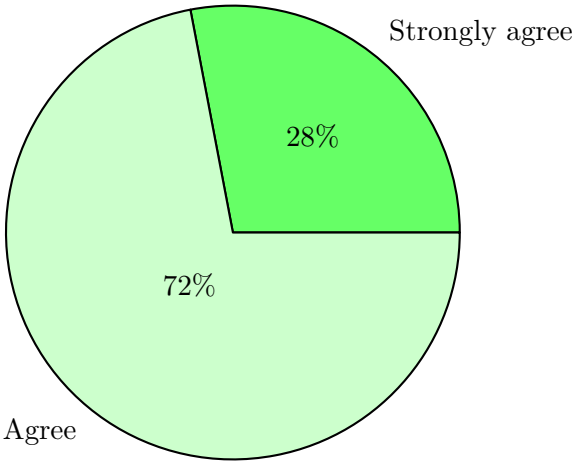

Figure 6.2: Answers to the statement „The application is intuitive to use“

While every participant agreed with the statement, that the application is intuitive to use, they often mentioned some of the problems that occurred in the scenarios. The most mentioned one was the tab panel, that was overlooked by everyone for a notable amount of time.

Organization of the application

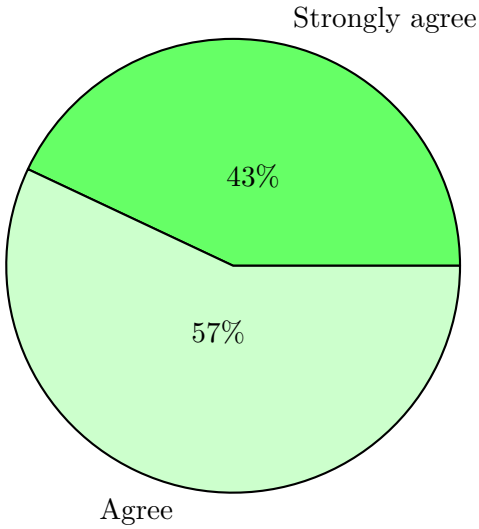

Figure 6.3: Answers to the statement „The application is well organized and structured“

A slightly better score received the application organization, about which the users spoke very well. They appreciated the easy and straightforward navigation using the left drawer, however more than half of the users criticized the role permissions matrix for being too big

and messy. Without the matrix in the scenario, the application would likely receive an even better score.

Attractive design

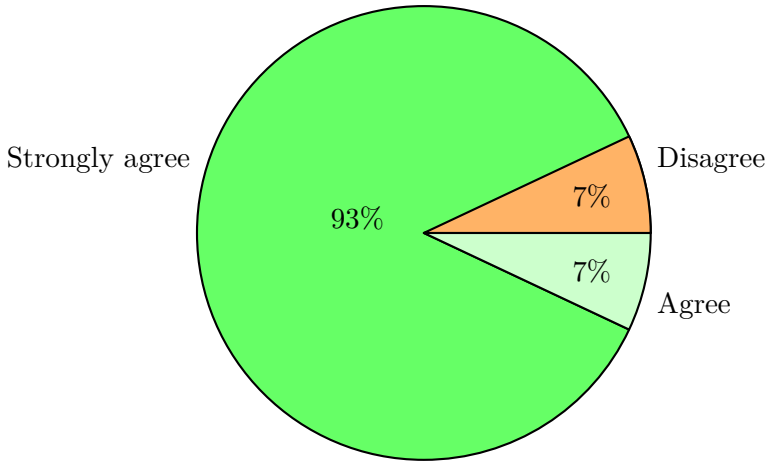

Figure 6.4: Answers to the statement „The application has an attractive design“

With the exception of one user, the participants generally liked the design of the application a lot. However, it is a subjective matter.

6.2.3 Summary

The user testing brought some valuable experience and had shed light upon several gaps in the design and user experience. It was however conducted in the final stage and it will not be possible to make any changes before submitting the project. Nevertheless, all discovered rooms for improvement will be discussed with the customer and will eventually get implemented in the future (see section 7.4 in the following chapter).

Chapter 7

Further development

This chapter introduces features, suggestions and bugs, that are likely to be taken for consideration after the application is released and developed further on.

7.1 Unfinished features

Features that were a part of the specification but were not implemented.

- Payment gate integration (see section 4.7.14) — not implemented, because the customer failed to negotiate a deal with any such service. Hence, it was not possible to implement in this timeframe. It was, however, considered a nice-to-have feature from the beginning.

7.2 Known problems

Some problems and bugs, that were discovered too late in the development and were not fixed before submitting (but will get fixed in the near future).

- Datepickers are not localized to Czech language — happened after an upgrade of a package, will probably wait for a patch, it is not a critical problem.
- Opening a datepicker causes the window to scroll to the top — also a bug in the library.

7.3 Possible enhancements

Additional feature suggestions, that arose during development and deployment:

- Download currency rates automatically (e.g. from CNB[1]).

[1] cnb.cz/en

- A button to generate all performer contracts for a given event and send them as e-mail attachments to each performer individually.
- Enable setting different currencies for contractors on the same event (currently, the currency is set globally for the event).
- A custom mobile application to validate e-tickets.
- User-editable listings like countries, instruments, concert types, etc. (currently statically defined).
- An activity log that keeps track of all committed operations (inserts, updates and deletes) and their author.
- Unit tests to ensure stable function of critical procedures, such as user reservations.

Many of the mentioned feature would improve the real-word usability tremendously while not being costly, therefore it is very likely, that they are going to be implemented in the application sooner or later.

7.4 Usability improvements

During user testing (see section 6.2), several places for improving the usability of the application were discovered. They were all noted, and might get eventually implemented in the future:

- Improve the look of the tab panel on detail views, to better indicate the actual function and avoid it being overlooked as some kind of a design feature (a separator).
- In the checkbox matrix for permissions configuration, make the table header floating (to be always visible), and differentiate the background of even and odd rows, to aid the user with keeping track of the rows.
- Highlight the user box (top right corner of the screen) so it is more noticeable.

Another user testing should be conducted after the eventual implementation of these features to verify, whether the solutions did actually help.

Chapter 8

Conclusion

This project was about creating an application that is tailored to the real-world needs of an actual chamber orchestra, that will use it to manage nearly all aspects of its functioning. The potential of the application was inducing both motivation to create a comprehensive and feature-rich software with added value, but also responsibility, to deliver high quality, reliability and practical usefulness.

The final product integrates an application created two years ago as a bachelor's thesis [10] and then developed further on, to form a final, complex system, that meets all requirements, and is ready to be publicly launched. Before that happens, it is likely that the customer would like to implement some of the features suggested in chapter 7. The release is planned after the state examinations. It is assumed, that after releasing, the co-operation and further development will continue. There are even suggestions for modifications and releases of the system for another customers.

The design and implementation of the system was an extensive task, that was a source of valuable experience. New and previously unmatched challenges that required creative solutions arose during the development. It was also the close co-operation with the customer and understanding the requirements, that taught me a lot. I am very glad for the opportunity to work on such a project.

Bibliography

[1] Elliott, E.: *Programming JavaScript applications*. Sebastopol: O'Reilly. 2014. ISBN 9781491950296.

[2] GDPR Alliance: *The General Data Protection Regulation (GDPR) In A Nutshell*. May 2017. [Online; cited 16. 1. 2019].
Retrieved from: https://medium.com/@GDPR_alliance/the-general-data-protection-regulation-gdpr-in-a-nutshell-2f02290dbd0d

[3] Grässle, P.: *UML 2.0 in action*. Birmingham: Packt Publishing. 2005. ISBN 1-904811-55-8.

[4] Harguindeguy, B.: *API Security: The Past, Present, and Future*. January 2018. [Online; cited 5. 2. 2019].
Retrieved from: https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1510001675.pdf

[5] Khorikov, V.: *Singleton vs Dependency Injection*. May 2016. [Online; cited 7. 2. 2019].
Retrieved from: https://enterprisecraftsmanship.com/2016/05/03/singleton-vs-dependency-injection/

[6] Mishunov, D.: *True Lies Of Optimistic User Interfaces*. Smashing Magazine. November 2016. [Online; cited 16. 1. 2019].
Retrieved from: https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/

[7] OWASP Foundation: *OWASP Top 10 - 2017, The Ten Most Critical Web Application Security Risks*. Technical report. Open Web Application Security Project. 2017. [Online; cited 6. 2. 2019].
Retrieved from:
https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[8] Pacholczyk, D.: *Web UI Design Best Practices*. Uxpin Publishing. 2014.
Retrieved from:
https://www.uxpin.com/studio/ebooks/web-ui-design-best-practices/

[9] Priya James: *Top 5 Most Common Web Application Attacks That Affecting Websites*. October 2018. [Online; cited 6. 2. 2019].
Retrieved from: https://gbhackers.com/web-application-attacks/

[10] Sikora, V.: *Configuration and Reservation System for Concert Halls*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2017. Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=20232

[11] Stotskyi, S.: *Managing user permissions in your React app*. DailyJS — Medium. February 2018. [Online; cited 16. 1. 2019].
Retrieved from: https://medium.com/dailyjs/managing-user-permissions-in-your-react-app-a93a94ff9b40

[12] Vencl, J.: *PHP a tvorba PDF dokumentů za pomocí mPDF*. August 2013. [Online; cited 8. 2. 2019].
Retrieved from:
https://www.itnetwork.cz/php/soubory/php-export-pdf-tvorba-dokumentu

Appendix A

package.json contents

```
{
  "name": "czvirtuosi-admin",
  "version": "1.0.0",
  "description": "CzechVirtuosi systems administration",
  "sideEffects": false,
  "scripts": {
    "start": "node server.js",
    "build": "NODE_ENV=production webpack -p --config=webpack.config.prod.js",
    "stats": "webpack -p --config=webpack.config.prod.js --profile --json > stats.json",
    "analyze": "webpack-bundle-analyzer stats.json",
    "add-locale": "lingui add-locale",
    "extract": "lingui extract",
    "compile": "lingui compile",
    "lint:fix": "eslint --fix src/**"
  },
  "license": "UNLICENSED",
  "repository": {
    "type": "git",
    "url": "git@gitlab.sovanet.cz:czech-virtuosi/admin.git"
  },
  "author": "Vit Sikora",
  "optionalDependencies": {
    "fsevents": "*",
    "google-closure-compiler-linux": "*",
    "google-closure-compiler-osx": "*"
  },
```

Listing A.1: package.json file contents, part 1 of 3

```
"devDependencies": {
  "@hot-loader/react-dom": "^16.8.6",
  "eslint": "^5.16.0",
  "eslint-plugin-react": "^7.12.4",
  "eslint-plugin-standard": "^4.0.0",
  "react-hot-loader": "^4.8.3",
  "webpack-dev-server": "3.2.1"
},
"dependencies": {
  "@babel/cli": "^7.4.3",
  "@babel/core": "^7.4.3",
  "@babel/plugin-proposal-class-properties": "^7.4.0",
  "@babel/plugin-proposal-decorators": "^7.4.0",
  "@babel/plugin-proposal-export-default-from": "^7.2.0",
  "@babel/plugin-proposal-export-namespace-from": "^7.2.0",
  "@babel/plugin-proposal-function-bind": "^7.2.0",
  "@babel/plugin-proposal-logical-assignment-operators": "^7.2.0",
  "@babel/plugin-proposal-nullish-coalescing-operator": "^7.4.3",
  "@babel/plugin-proposal-object-rest-spread": "^7.4.3",
  "@babel/plugin-proposal-optional-chaining": "^7.2.0",
  "@babel/plugin-proposal-throw-expressions": "^7.2.0",
  "@babel/plugin-syntax-dynamic-import": "^7.2.0",
  "@babel/plugin-syntax-import-meta": "^7.2.0",
  "@babel/plugin-transform-modules-commonjs": "^7.4.3",
  "@babel/plugin-transform-runtime": "^7.4.3",
  "@babel/polyfill": "^7.4.3",
  "@babel/preset-env": "^7.4.3",
  "@babel/preset-react": "^7.0.0",
  "@babel/runtime": "^7.4.3",
  "@casl/ability": "^3.0.2",
  "@casl/react": "^1.0.3",
  "@date-io/date-fns": "^1.1.0",
  "@material-ui/core": "^3.9.3",
  "@material-ui/icons": "^3.0.2",
  "@tinymce/tinymce-react": "^3.0.1",
  "autoprefixer": "^9.5.1",
  "axios": "^0.18.0",
  "babel-eslint": "^10.0.1",
  "babel-loader": "^8.0.5",
  "babel-plugin-lodash": "^3.3.4",
  "babel-preset-es2015-ie": "^6.6.2",
  "closure-webpack-plugin": "^2.0.1",
  "connected-react-router": "^6.3.2",
  "css-loader": "^2.1.1",
```

Listing A.2: package.json file contents, part 2 of 3

```
    "date-fns": "^2.0.0-alpha.27",
    "file-loader": "^3.0.1",
    "google-closure-compiler": "^20190301.0.0",
    "history": "^4.9.0",
    "html-webpack-plugin": "^3.2.0",
    "less": "^3.9.0",
    "less-loader": "^4.1.0",
    "lodash": "^4.17.11",
    "lodash-webpack-plugin": "^0.11.5",
    "lokijs": "^1.5.6",
    "material-ui-pickers": "^2.2.4",
    "mini-css-extract-plugin": "^0.5.0",
    "number-format.js": "^2.0.8",
    "postcss-import": "^12.0.1",
    "postcss-loader": "^3.0.0",
    "prop-types": "^15.7.2",
    "react": "^16.8.6",
    "react-custom-scrollbars": "^4.2.1",
    "react-dimensions": "^1.3.1",
    "react-dom": "^16.8.6",
    "react-dropzone": "^4.3.0",
    "react-redux": "^6.0.1",
    "react-router": "^4.3.1",
    "react-router-dom": "^4.3.1",
    "react-s-alert": "^1.4.1",
    "react-select": "^2.4.2",
    "react-sortable-hoc": "^1.8.3",
    "redux": "^4.0.1",
    "redux-thunk": "^2.3.0",
    "style-loader": "^0.23.1",
    "url-loader": "^1.1.2",
    "webpack": "^4.29.6",
    "webpack-cli": "^3.3.0"
  },
  "moduleRoots": [
    "src",
    "styles"
  ],
  "browserslist": [
    "last 2 versions and > 0.25%",
    ">1%"
  ]
}
```

Listing A.3: package.json file contents, part 3 of 3

Appendix B

composer.json contents

```
{
    "name":"czechvirtuosi-api",
    "description":"CzechVirtuosi administration system - back-end part",
    "keywords":["server","api","framework","REST"],
    "license":"UNLICENSED",
    "authors":[{"name":"Vit Sikora", "email":"vit@sikora.cz"}],
    "require":{
        "php":">=7.0.0",
        "luracast/restler":"*",
        "nette/database": "^2.4",
        "nette/di": "^2.4",
        "tracy/tracy": "^2.4",
        "phpmailer/phpmailer": "^5.2",
        "latte/latte": "^2.4",
        "phpoffice/phpspreadsheet": "^1.2",
        "jeroendesloovere/vcard": "^1.5",
        "mpdf/mpdf": "^8.0",
        "mpdf/qrcode": "^1.0"
    },
    "autoload":{
        "psr-0":{
            "Luracast\\Restler":"vendor/"
        }
    },
    "conflict": {
        "restler/framework":"3.*"
    }
}
```

Listing B.1: composer.json file contents

Appendix C

Contents of the enclosed storage medium

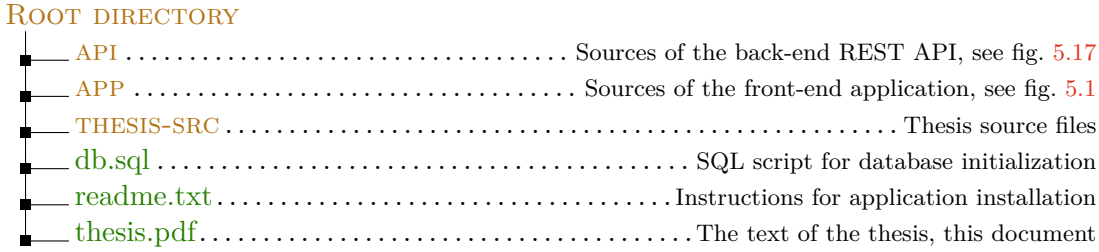

Figure C.1: Contents of the attached CD.