Ministry of Education and Science of Ukraine
NATIONAL UNIVERSITY OF "KYIV-MOHYLA ACADEMY"
Network Technologies department of Faculty of computer sciences

Development of the auditoriums' occupation management system called "KMAuditoriums"

Text part for the coursework on specialization „Software Engineering" 6.050103

Instructed by:
Andrii Hlybovets,
Doctor of science

"25" March 2020 p.

Performed by:
Leskiv Oleksandr
"25" March 2020 p.

Kyiv 2020

Contents

Annotation

In the modern world, most applications are built using software as a service (SaaS), client-server approach. With server-rendering is no longer being a common nor recommended practice for most scenarios, Frontend development became separated from the Backend one. They might be built using different languages and technologies and will usually be hosted on two different servers, one for web's static files serving and another for the application programming interface (also known as API) server. The only thing that ties those two is the HTTP protocol. With that, the whole application development becomes more flexible as frontend and backend teams can work independently of each other and even test their code using mock data (for frontend) and an API development tools like Postman (for backend) to be completely separate.

Another benefit of having application's parts communicating over HTTP protocol is that a website itself (page markups, styles and the code) can be cached on user's device and only constantly updated data is queried from the backend, which increases speed and greatly improves user experience. This can even be taken one step higher and web application can be converted to a progressive web application, which can be installed on user's device to make them even more accessible.

But as with everything in this world, there are some tradeoffs. With this approach, API is exposed to the internet and we have to make sure that if somebody tries to access it (besides our website), no data on the server will be changed in unintended way nor it will be overexposed to users without the correct accesses. Another point of consideration is the frontend application itself: we have to make sure it doesn't contain any sensitive information (private or API keys), or that sensitive information is supposed to be publicly available since its usage is restricted only to the domain where your website is hosted.

If the second security consideration mentioned above is rather simple to fulfill, the API security may be somewhat a challenging task, especially for big multirole applications. In

order to make it easier for developers, the API is usually built using a common practices and recommendations, which imply specific naming for endpoints, HTTP methods that is to be used for each of them and so on. There are quite a few approaches, but the most common and practical are REST and the ones that involve Graphs. Graph-oriented APIs are modern but should not be used everywhere. They mostly shine in applications where database entities have really a lot of nested dependencies, for example, Person -> Post -> Photo Series -> comments -> replies etc. Therefore, this technique is adopted, continuing the previous example, by Instagram & Facebook. On the other hand, in the auditorium's management application, it would be obsolete, so we'll go with a REST API.

REST or Representational state transfer is an architectural style that defines a set of constraints to be used for creating Web services [1]. It's a solid approach, and we are going to break it down and see how we can apply it to our API in the following sections of this coursework.

In order to make it easier for developers to tie frontend and backend, it is considered a good practice to document your API so it is not only exist somewhere is the code, but you can also investigate all of its endpoints with exact paths, HTTP methods, parameter, status codes they produce etc. It used to be done manually back in the older days, but in modern world people can not only generate this type of documentation automatically, but also utilize it to heavily speed up the development process for frontend. Technologies that we are going to use to do that will also be described in detail in the following sections.

Last but not least is the deployment. Besides making an app, you also have to make it available to your target audience. KMAuditoriums is going to be rather simple app that is not going to be constantly updated. Therefore, we will not implement Continuous Integration nor Continuous Deployment techniques for it during this study. However, it will be definitely covered how application's parts should be built and/or compiled and uploaded to the cloud server or the one based in NaUKMA in order to make it work semi-manually.

Introduction

National University of Kyiv-Mohyla Academy offers a lot of possibilities to its students. As students, we can make requests to organize all kinds of events in the university starting from small rehearsals and gathering and ending with lectures and concerts in the assembly hall. The problem with that is rather technical: it is hard to manage the availability of the auditoriums. Each auditorium and hall in the university have one or several people that are in charge of the events happening in those. Therefore, for students it is usually not clear whom they should contact if they want to organize an event there. Furthermore, it is not clear whether auditorium is going to be free at that time in the first place.

In order to address this issue, during this study I am going to create a service, that in conjunction with NaUKMA Schedule Service and NaUKMA Role Service will handle auditoriums' occupation state including schedule, student events and events created manually by managers.

The study will cover functionality of this service along with a block-scheme style design, requirements that have to be met in order to make it work, roles and their responsibilities distribution, technologies that will power that and implementation details.

The objective of the study is to develop an application that will keep track of auditoriums' occupation in National University of Kyiv-Mohyla Academy including schedule events & student requested events.

The task of the study is to analyze the how students can actually make requests to occupy an auditorium and how that request should be handled by administration in order to incorporate that flow in the application being developed.

The topicality is on a high level because, the university currently doesn't have any automated nor centralized solutions to handle auditoriums occupation, that is easily accessible to students and easily managed by people who are in charge of auditoriums occupation.

Research sources consist of online resources like official documentations of technologies being used and related topics about best practices on how to use those technologies as well as written consultations of a supervisor on how the system should be built and managed.

Research methods mostly consist of analyzing auditoriums occupation flow and research on how it can be automated using client-server applications.

The practicality of the results received is high, since the existence of a described service will encourage students to organize more events within the university and will make it easier for people that are in charge of auditorium's events to review and accept those events.

Section 1. Business Requirements of KMAuditoriums

As with every project, before the implementation it's recommended to start with planning and defining the requirements, so I am going to do exactly that.

Generally, auditorium's management service is not going to be a standalone product, but rather a part of NaUKMA's technical solutions eco-system. As for authentication it will use, like other services, Office 365. But besides that, it will also directly communicate to Role Service and Schedule Service. Role Service will help authorize user after the authentication to distinguish platform admins, auditoriums managers and students. Schedule service will query auditorium service and fill auditoriums occupation with periods from schedule, since auditoriums can be occupied both by students and, actually, periods that have already been there.

One of the primal entities that KMAuditoriums will operate with is "Auditorium". An auditorium is characterized by its name, belonging to one of the NaUKMA's buildings, its capacity, an optional description and a photo.

Next important entity that actually auditoriums are occupied with is "Event". An event is a more complex entity, since, as we have already mentioned, it can be one of two types: "schedule" or "application" (a request from a student). Therefore, it can also have several statuses, namely, "pending", "approved" or "declined". And last but not least, an event also defined by its title, an optional description, date, start and end times. It also has a requester and an optional reviewer reference. An event can also belong to a faculty.

As we have already mentioned, there are going to be multiple roles across the platform. KMAuditoriums will have at least one Platform Administrator, who's primal goal will be to assign managers to auditoriums they will be in charge off. Besides that, platform administrators will be able to accept or decline any auditorium event requests. Also, platform administrators will be able to edit auditoriums info in case it is needed, even though it will be mostly managed by a Schedule Service. Since administrator can accept any event, the

ones he creates automatically get “accepted” status. An exclusive feature that administrator will also have is a statistic, which will help investigate the auditoriums occupations though the study weeks. An administrator should be able to select one or more auditoriums in the system, target weeks, weekdays and period times and view the information about how many events are happening under these specifications compared to the potential (how many there might have actually been). There most relevant statistic charts to be displayed are days distribution, periods distribution and occupation information that includes date, period and event’s faculty or requester information.

Once administrator grants a manager an access to an auditorium, that person gains an ability to review (accept or decline) events related to that auditorium. Similarly, any event this manager adds to one of auditoriums that he/she is in charge of, automatically receive “accepted” status as well. One auditorium can have unlimited number of managers.

Least capable role in the system is a student role. A user can still browse and search auditoriums and even see their occupation at different dates and times, but he will not see, nor, obviously, have an ability to accept or decline any events and events that this user submits will always have to be reviewed by corresponding manager or a platform administrator. Also, unlike Platform Administrator or Manager, a student’s event request won’t be able to contain a faculty reference thus it will be treated as non-faculty event, and platform administrator will see this student’s name in his statistic instead of a faculty name.

Important to note that even though administrators and managers are the ones who will be responsible for accepting and declining events, those can also be hard deleted only by their creators (in case you’ve accidentally created an event).

Everybody will be able to browse and search auditoriums and view their occupation for a day of interest. Platform Administrators and managers, who have access to the auditorium, among approved events will also see pending & declined ones. The occupation of auditorium on a specific day consists of approved non-overlapping events that happen on that day.

One specific role that will also exist in KMAuditoriums is a schedule one. With this role, an already mentioned Schedule Service will create auditoriums and events with “schedule” type based on university’s schedule. Unfortunately, by the time is study was conducted, such service hadn’t been introduced to NaUKMA’s eco-system. On the bright side, the creator of this study, as a lead developer of digital schedule manager called “KMASchedulerBOT”, has access to its database. It is not a centralized Schedule Service but rather a schedule parser that converts university’s schedule documents into digital format and makes it available to NaUKMA students in more convenient way. It will be used to replace Schedule Service for now, and the data about auditoriums and schedule will be loaded from there upon the launch of KMAuditoriums.

Section 2. Backend implementation details

2.1. Data storage considerations

The implementation of backend starts from designing a scheme of a database. Considering a pretty straightforward technical task (business requirements), we can do that pretty easily. However, there are still going to be a couple of considerations about the way images and profile information are going to be stored.

The problem with storing images lies in the choice of a storage type. Basically, there are two options – we can either store images in a raw format in corresponding columns in the main service's database or we can store them using a dedicated third-party service like Amazon's AWS S3 storage or Imgur and store only a link to the corresponding image resource in the main database. Generally saying, second option is more attractive from user's experience perspective, since dedicated third-party services are more specialized on storing big files (relative to all other information in database) and, therefore, they have bigger performance and bandwidth comparing to potential performance of our custom database. On the other hand, this creates a weak spot in the whole system's infrastructure. Furthermore, a photo uploading is not going to happen too often (especially for students, since there is no functionality for them that is related to photos uploading, expect for potential profile photo upload, which anyways happens objectively rarely). So, with that being said, in this coursework I decided go with inside-database images storing technique. This will be applicable to profile.

Next point of consideration lays in the profile data. Obviously, some basic profile information in the system must not only be available to those users themselves, but also to other people in the system. This is applicable to events, for example, because requester and reviewer information should be always available whenever administrator, auditorium manager, event creator or whoever else with the appropriate access wants to view the details

about particular event. That brings to the to the Office 365 authentication investigation – we need to see what info we can extract from the information that is exposed to us after we click that “Login with Microsoft” button on our website.

In order to do that, I created a sandbox React project and included external library called React AAD MSAL [2]. In order to make it work, as its documentation suggest, we had to proceed to “How to create Microsoft app” tutorial and create an appropriate app registration on Azure’s portal. Well, that is pretty straightforward process. Since I am a NaUKMA student, I already had a Microsoft account and luckily, it was sufficient enough to pass me through directly to main portal’s page. From “apps registration” page I proceeded with creating a new registration for out KMAuditoriums project. It was enough to give it a name (something like “NaUKMA-Auditoriums”) and the whole creation process was basically done at that point. In order to make it work on my website I proceeded to “Authentication” page and included “http://localhost:3000/” to the list of “Redirect URIs” for local development (since react apps by default are launched on port 3000 during the development). Later during the deployment we’ll also have to add here our final domain where a website will be hosted in order to make authentication work in production. After some testing, a few additional configurations in “Authentication” tab were added under “Implicit grant”, specifically two optional authentication items were select: “Access tokens” & “ID tokens”. The last thing I checked is “API permissions”. By default, it includes “User.Read” permission underneath “Microsoft Graph” permissions group, and that was satisfactory enough for auditoriums’ service needs. At this point setup is done, in “Certificates & secrets” tabs I grabbed an Applications (client) ID, a Directory (tenant) ID and an Object ID values to utilize them later on frontend and backend.

After setting up everything for a test, I finally was able to actually “Login with Microsoft” to my university account. The object I got (see Figure 1) includes several crucial things: “idToken.rawIdToken” and “accessToken”. Everything else in the object returned can actually be either extracted from these two tokens or fetched from Microsoft endpoints using

them. Furthermore, they are Oauth2 bearer tokens, therefore they can be used on backend to authenticate a person. That is exactly what we need for KMAuditoriums.

With that being said, it is time to dig deeper into the technologies we are going to use for out backend. Since most of the services for National University of “Kyiv-Mohyla Academy” either are written in Java or will be written using Java, we basically have no choice here rather than go with Spring Boot solution for the backend.

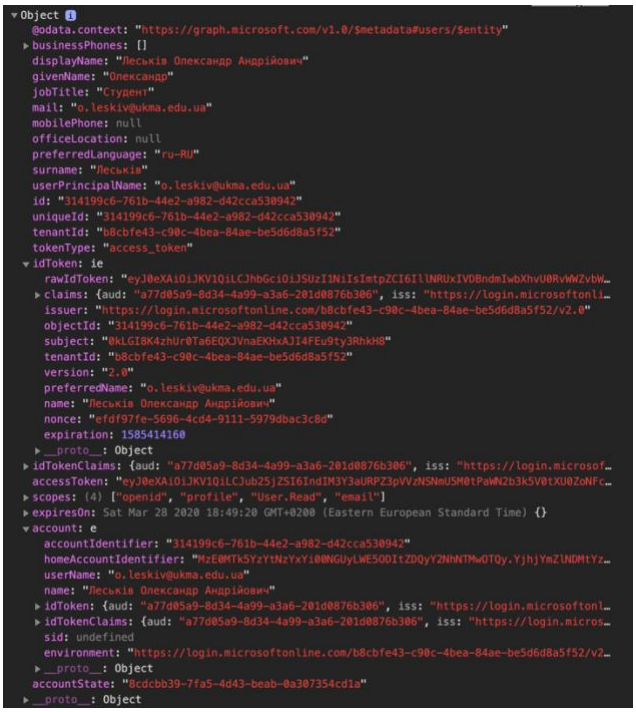

Figure 1. Successful login with Microsoft

After the initial “getting started” project, next thing I though was logical to proceed was is authentication. Having the results of frontend’s “Login with Microsoft” investigation I started to make a research on Spring Boot’s libraries which can provide me a functionality I was looking for. Sadly, after a long day of searching, it was clear that Azure’s extension that can automatically authenticate requests using tokens can actually only work with “IdTokens” but not with “AccessTokens”. Well, one may wonder, what is the difference in those two. And the answer lays in the type of data that can be extracted. With “IdToken” you can only have brief information about a person being authenticated. That basically limits us to the “display name”, “email” and “Microsoft id”. On the other hand, access token would have given us a full name breakdown, like “surname”, “name” and “middle name”. The absence of this breakdown can be mitigated by the fact that we at least have “display name” on the backend, but the most crucial info that would be nice to grasp is the profile’s photo. On the bright side, Microsoft’s photo is actually accessible on frontend (since we have already established, that it actually has an “AccessToken”) and in theory can be uploaded manually to our backend using an id token. The downside of this approach, is, obviously, security. The problem lays in the fact that a

potential hacker can upload an arbitrary photo using his signed id token. But on the other hand, a photo is still uploaded manually to Microsoft's profile as well, and nobody actually verifies that it is your photo. The same can't be said about the full name, since if a hacker would be able to change that info in our KMAuditoriums service, it could cause serious security problems. With that being said, I decided to not go with manual photo nor full name breakdown upload using an "IDToken" for now, since "display name" is sufficient enough on its own and photo can be generated using a person initials using a third-party service even on frontend.

The reason is why we actually need to copy data from these tokens to a database on backend lays in the fact that both "IDToken" and "AccessToken" have short lifespans. The expiration date for a token can be approximately 30 minutes which is far less than a desired infinity. After user's session will expire, we still need to show his name and email to others in order to let them know who, for example, created a specific event. Therefore, our database will have a dedicated profile table, that will map user's Microsoft IDs to their names, emails and photos. Not for those users themselves, but rather to whoever will want to see info about them after their session expires. This info will be updated every time user will make an authenticated request to KMAuditoriums in order to make this "cached" data always up to date.

The last point of consideration is a storage of event's types and statuses. The initial desire was to make two table called "event_type" and "event_status" in order to store names in separate table and avoid names repeating in the "event" table itself. From one perspective, this was a good idea because who wouldn't want their database to be in 4-th normal form (with three other normal forms in addition). But on the other hand, this would require a couple of additional inner joins whenever we would want to load an event, and it would be even more tedious to create such event. Therefore, I decided to go with plain strings for type and status and utilize Java's enum to handle constant strings for me.

With everything said above, it's finally time to create a database scheme. We will skip the ER-model step because of the simplicity of our database and the fact that nobody actually does this anymore, and continue with the actual "R" model, e.g. the one that is directly applicable to an SQL database (see Figure 2).

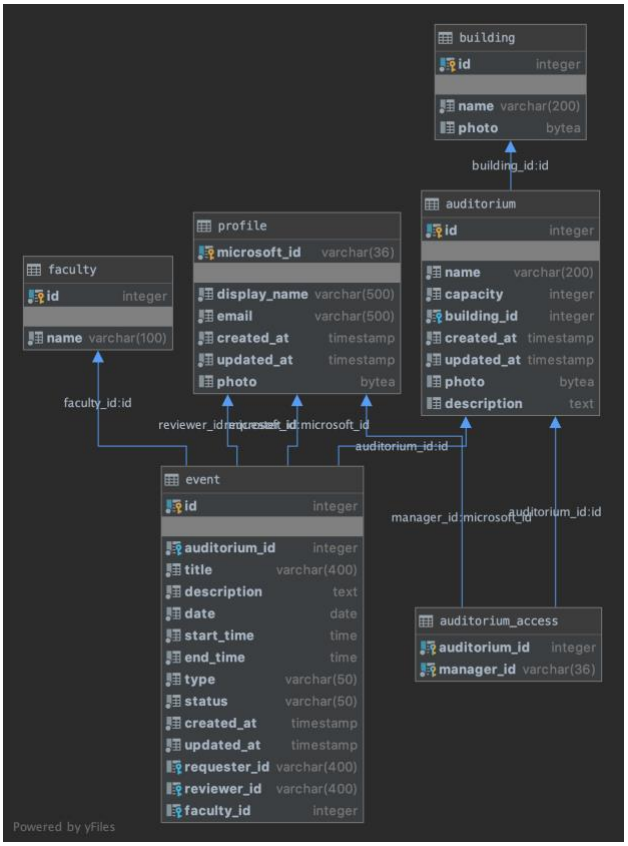

Figure 2. R-model of a database

Resulting model looks logical taking into account business model and all considerations that was described previously in this section. With R-model finally being created, it is time to apply it a database.

There are quite a few SQL databases to choose from nowadays, but since in the KMAuditoriums we won't have high demand on any specific characteristic (like write speed or read performance), in this coursework I am going to use relational database PostgreSQL, just because I have been working with it before and it has some neat functions that will be benefit us during the statistic generation.

We are going to use database migrations. Even though this project probably will not differ a lot in the future, it is still nice to have this in a backend project to help future developers that may come over you project to understand what the database scheme looks like and how it have been changing over time. For database migrations, Spring Boot offers quite a few options, but we'll go with Flyway migrations, since they work just fine. It's enough to place "db.migration" folder inside a "resources" directory, place there an SQL file called "V1__initial.sql", and we're good to go.

2.2. Entities

In Java Spring Boot, most operations with databases happen indirectly, but rather with Java Persistence API (JPA) and Hibernate. It is a high-level abstraction on top of the raw database's tables that frees developers from writing SQL queries themselves, but rather rely on the Entities class definition to automatically instantiate, save them, update and everything else. For every column, key, join and others we just have to put corresponding annotations and Java Virtual Machine will automatically validate entities in the project to the tables that exist in the database. That is pretty convenient since having this mechanism guarantees you the integrity of a database schema. Conveniently, if you have some foreign keys in your database, you can specify in corresponding entities columns, by which they should be joined, and those sub-entities will be instantiated automatically as well.

For KMAuditoriums, there are not too much entities, and only a handful of enums. The exception for this rule is only applicable for Auditoriums Statistic since these types of queries is easier to write in raw SQL rather than describe them though some Query Specifications in those high-level abstractions. In those cases, a Spring Boot developer can write some native queries and provide interfaces that describe those queries return types, and Spring Boot just creates classes that implement those interfaces and instantiate them using the data returned from a database in a runtime.

Among entities in KMAuditoriums we have Auditorium, AuditoriumAccess, Building, Event, Faculty & Profile. Those correlate with tables created in a database. There are a couple of additional classes that help handle the security process (roles distribution) and also helpers for searching controllers (like auditorium search).

2.3. Architecture

Having our entities set up, we have to think about how we can actually access them though our REST API. To get started, Spring's Boot offers a possibility to implement great

programming pattern called “dependency injection”. In a nutshell, this frees developer from instantiating classes inside a project manually, but rather use “Autowired” annotation above variables. Usually, it is recommended to “autowire” some beans inside a project, so when the instances of corresponding classes will be created, they effectively implement a singleton pattern, meaning, that if several classes use variables of the same type annotated with “Autowired”, they will point to the same instance at a runtime.

Having that technique, Spring Boot projects usually evolve into an architecture called “domain-driven design”. This concept is rather complex but can be relatively easily implemented for a small project like KMAuditoriums. A main goal of this approach is to separate an application interface (for our web app – it is an HTTP API) from use cases (create something, approve an event), database operations (query something) and entities themselves. Breaking down each web request into multiple layers may seem redundant, however in a long-term perspective this really saves a lot of development time because it forces developers to think about high-level abstractions, which generally is more productive because it is more natural for humans.

Speaking more specifically, there are common practices how those layers should be defined, named and what they should do. Let’s break an example use case to see how every part of the whole process will map to our layered architecture.

Take for example event approving. Using a KMAuditoriums website, an administrator will come over the details of some pending event and will see two buttons, “Approve” and “Decline”. Suppose this administrator clicks an “approve” button. A corresponding API endpoint will be queried with a POST request, receiving an event id is its request path (administrator’s ID Token will be provided in the header, and the way it is going to be validated will be described later in this section). The first layer that will be involved would be an interface layer. This layer contains does no more than mapping an API request to a specific domain action, a use case. This separation is recommended, because as application gets bigger, a use cases can be triggered from different sources. For example, an event

declining can happen both from an API endpoint (whenever admin click the “decline” button in the web interface) and from a hypothetical cron job that will just automatically decline events that have been in pending state X amount of time. Interface level for web applications the most part consists of API controllers; however, it can be potentially expanded to handle socket connection, RMI or something else among these lines. Interface level also utilizes Data Transfer Objects (also known as DTO), to reduce amount of data transferred between frontend and backend. For example, Event entity has Auditorium instance inside of it. But in order for frontend to specify auditorium in which event is taking place at, it’s enough to pass only an auditorium’s id, and a backend can generate an Event entity from a DTO and an info it can pull locally from a database. Not only this reduces transferred data amount but also simplifies things for frontend.

Use cases or services usually represent business logic actions that can happen to a domain. Usually services have at least CRUD functionality, so you can create your entity, query it, update or delete by id. Thankfully, with Java’s generics and inheritance it is frankly easy to create services from a CRUD template and extend them with entity-specific functionality like “approve event” for events service or “gather statistic for auditorium” for auditoriums service. In order for services to work, they utilize repositories, which belong to the next layer.

Repositories act like an intermediate layer between service and a data storage (in our case, a database). They are mostly responsible for inserting things into and pulling things from a database. Java Persistence API is doing a great job at creating those repositories automatically taking Entity class and an Id type. However, repositories sometimes may be expanded with custom native queries like the auditorium statistic repository.

2.4. Security

As I have already mentioned in previous section, our API must be protected in case somebody will try to access it not from KMAuditoriums frontend. For these purposes, we

are going to use Azure's library to help us with authentication in combination with our custom Requests Filter that will attach role to an authenticated user, update its profile in our database (as we have established in previously in this section) and save the whole thing in the context so it can be reused in the next layers. In one word – authorize.

Thankfully, the whole process is quite straightforward and easy. In order to get Azure's authenticator to work, we have to insert into "application.properties" files secrets from an azure's directory we have retrieved earlier and create a bean that extends "WebSecurityConfigurerAdapter" in order to specify which API endpoints we want to secure and also specify Azure's authentication as security type. In KMAuditoriums there is no "guest" role, therefore, all logic-related endpoints will be marked as "secured". However, as I have already mentioned, in this project we will have an API documentation, that should be accessible to everyone, and those endpoints will be exceptions to the rule. CORS policy is also specified here in order to permit by default restricted POST, PUT & DELETE HTTP methods as well as arbitrary endpoints.

Having that set up, every request that doesn't have Authorization header with valid id token we have been previously talking about will be automatically rejected with "Forbidden" status code. For authenticated requests, a request's context will be will filled with instance of Principle class – a one with email, display name and Microsoft id fields. In order to perform an authorization, it is enough to add a class that implements "javax.servlet.Filter" interface, mark it with "Component" and "Order" annotations and basically replace existing principle with custom class "AuthorizedUser" which is basically identical to the base Principal, but also with a "GrantedAuthorities" list that is populated with Role enum (Admin, Manager or Student) based on the results of querying the Roles Service. Unfortunately, at the moment of implementation this service had not been introduced to the NaUKMA's ecosystem yet, so in my case I've just made a switch that grants a role based on the hardcoded email.

The next logical step is to take advantage AuthorizedUser principle existence. In Spring Boot there are a couple of ways to do that. Firstly, we can filter some of the requests path to only specific granted authorities. Second option would be to define an xml file that will basically do the same. And the third option is to apply “PreAuthorize” annotation on the interface to a specific rest controller, or one of its methods. Also, we can skip the granted authorities check at all and do it manually inside any of the methods in case an endpoint can work for multiple roles but slightly differently for each of them. In case of KMAuditoriums, we will go with a third option, since most of the controllers will be able to work with several roles at a time and a lot of endpoints will slightly change their behavior based on granted authority but will still support all of them. For example, event creation endpoint assigns different status to the event based on the granted authorities – students always create their events with “PENDING” status, admins – with “APPROVED” and for manager it depends on their access level.

2.5. REST API and Swagger

To make things easier for frontend developers, a lot of tools exist that can help create a documentation for an API. Some even them take this process to the next level and generate a such detailed specification of parameters, request and return entities types that it can be used to generate a API client file that will construct the whole request, filling an http method, appropriate parameters, path, query and body variables and even type check the whole thing to catch most of the request format errors on the frontend prior to making a request to a backend. That’s really helpful since effectively whole application’s REST API can be defined only once, on the backend, and a frontend project will be able to utilize http requests without explicitly writing paths or worrying about where to put a variable – into query parameter, path or body and so on.

These kinds of tools can basically work in two modes: you either define the documentation first and then it manages routing and parameters on your backend, either you

first write all of your parametrized interfaces and documentation is generated based on that. In Spring Boot, we do not really get to choose, because it only works with a second approach. So, with that being said, it is time define our API controllers.

As it has already been said in previous sections, the API must be built using REST recommendations. In order to do that, we first have to groups all endpoints by entities. In KMAuditoriums, there are a handful of rest controllers to make: auditorium, event, building, faculty, profile & statistic. Each of them should have unique prefix in the request path that corresponds to their name. Next basic thing to do is to divide endpoints into two groups: id-related and non-id-related. To non-id-related endpoint belong “create” endpoint, for example, but “update” is an “id-related” one. For those endpoints that perform operation on the entity with an id, the “id” should be included in request’s path parameter. Third step of REST API construction is to define https methods. Generally, for entities creation, business logic status changes and heavy calculations, recommended http method is “POST”. For querying (getting by id, searching etc.), “GET” method should be utilized. Finally, for editing and updates, it usually preferable to use “PUT” and any deletion should be performed with an http method “DELETE”.

With that being quite easy to follow and implement, it is time to enable the documentation. In this coursework we are going to use tool called “Swagger”. It is a popular and actively supported tool that has a great integration with both Spring Boot and Typescript (for frontend). In order to enable it, it is enough to add a corresponding dependency into the “pom.xml” file and create a “SwaggerController” class with “Configuration” and “EnableSwagger2” annotations, in which it is enough to provide path to REST controllers we want to document. A final step would be to permit Swagger paths in the “SpringSecurity” bean in order to access documentation files without being authenticated.

Now when we launch our application and go to the "/v2/api-docs" path relative to the domain of where the server is launched, we will see a JSON file similar to the one shown in Figure 3. This is a description file that will later be utilized on a frontend.

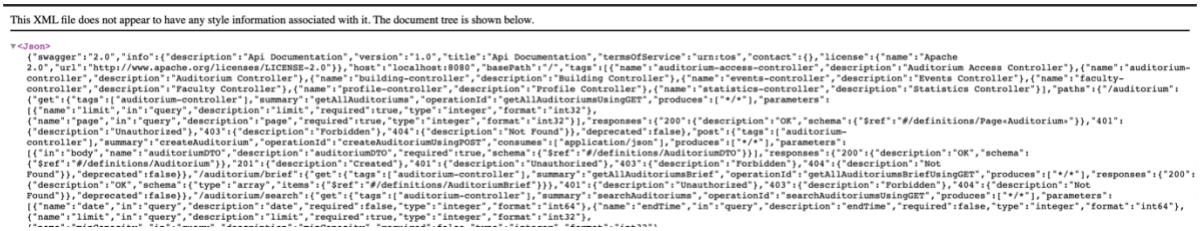

Figure 3. Swagger JSON

To see a human readable version of our API, we have to browse to "/swagger-ui.html#/", again, relatively to the domain where server is hosted, and we'll see somewhat similar documentation to the one that is shown in Figure 4.

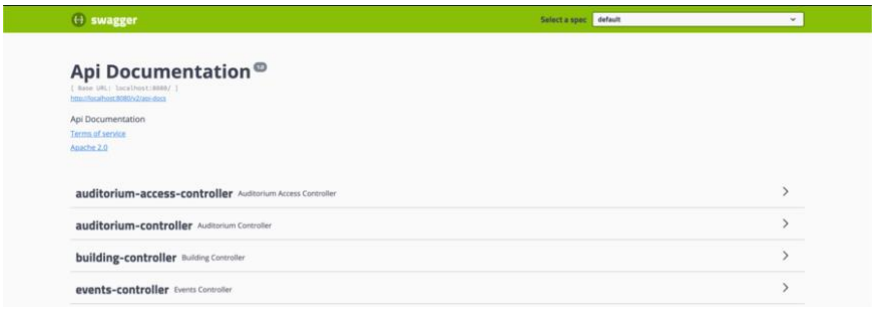

Figure 4. Swagger Documentation

If we will continue browsing this mini automatically generated for website, we will also see the detailed description of the available endpoints (Figure 5), their parameter and return types (Figure 6) and even exceptions and status codes that may occur during the execution (Figure 7).

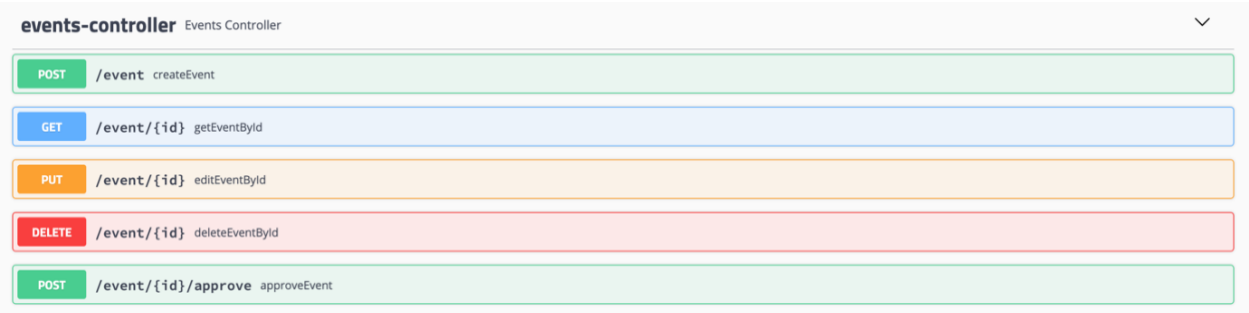

Figure 5. Available Endpoints

Similar to endpoint's paths, methods and parameters, REST recommendations are also applicable to statuses codes being used. For example, creation-related endpoints after successful creation of a resource should return status code 201 (Created), or whenever authentication error is raised, endpoint should return status code 401 (Unauthorized). There are a lot of status codes available and their description can be easily found on the internet.

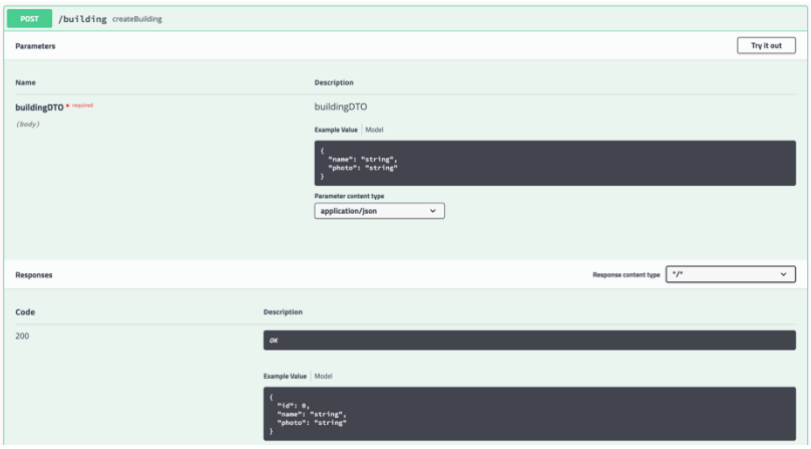

Figure 6. Endpoint Parameter Types

Figure 7. Possible Error

2.6. Auditorium Statistics

One of the requirements of the application, besides managing auditoriums occupation, is for platform administrator to see the statistic of all events happening in the auditoriums. Since this info is primarily used to optimize a schedule, it is also a good idea to compare current occupation to the potential, e.g. how many events are happening for the selected period vs how many could have been happening.

As we have already established, events stored in database have their dates and times. However, schedule in the NaUKMA is mostly oriented not on absolute values, but rather on weeks numbers (since the beginning of the trimester), weekdays and periods numbers. Therefore, statistic will be much more helpful if even make It work in that format. And, actually, this can be pretty easily achieved. In order for backend to support that it needs that date of the trimester beginning. And in order to make backend the most versatile, we can provide this date as another parameter for statistic's rest controller.

With that being said, let's break down 3 charts from business requirements. All of the charts are built using the same input data: week numbers, weekdays, periods numbers and auditoriums ids, and all of them can be arbitrarily combined in one request by a platform administrator.

First chart of the interest is a distribution of periods by weekdays. It's a bar chart with up to seven columns (depends on weekdays being queried). The "potential" for one column equals to the product of selected weeks count and periods count, since in one day you can have only as much events as there are days of interest amount times periods of interest in each single day. Similarly, in order to calculate actual a number of events in comparison to the potential, for each week of interest we have to count events, grouping them by weekdays and filtering by all of the parameters mentioned above. This can be achieved by running a database query which is shown at Figure 8. Note that this query should be run several times once for each week, because weeks may not be sequential (a query can be run for 2-nd, 4-th

```
interface DaysDistributionRaw {
    int getWeekday();
    int getPeriodsCount();
}

@Query(value = """
            SELECT date_part('dow', date)     AS weekday,
                   COUNT(DISTINCT start_time) AS periodscount
            FROM event
            WHERE date ⩾ :startDate
                AND date ⩽ :endDate
                AND auditorium_id = :auditoriumId
                AND date_part('dow', date) IN (:weekdays)
                AND start_time IN (:startTimes)
            GROUP BY weekday
        """, nativeQuery = true)
List<DaysDistributionRaw> genDaysDistribution(
        @Param("startDate") LocalDate startDate,
        @Param("endDate") LocalDate endDate,
        @Param("auditoriumId") int auditoriumId,
        @Param("weekdays") List<Integer> weekdays,
        @Param("startTimes") List<LocalTime> startTimes
);
```

Figure 8. Days Distribution DB Query

and 6-th week numbers). The last step is pretty straightforward – functional reduce method is used to accumulate periods count among different weeks and we get the desired data for a chart. There is also a way this process can be a bit optimized by replacing distinct weeks requests into a combination of OR parameters to minimize database requests count, but since in one of the consequent charts, distribution by weeks is required anyways and the general performance is not so affected (nor it is supposed to be extremely fast) I decided to make the queries work in similar fashion so the whole code is easier for future programmers to read and debug.

Next statistic char is built similarly to the first one. But instead of weekdays, events that satisfy parameters of the interest are grouped by periods. This chart is conveniently shown as a pie chart, where 360 degrees correspond to the potential of the selection (weeks count times weekdays count times periods count), and number of events for each period take up some percentage the whole Pie. A corresponding query is shown in Figure 9.

```
interface PeriodsAccumulationRaw {
    String getPeriod();
    int getPeriodsCount();
}

@Query(value = """
            SELECT to_char(start_time, 'HH24:MI:SS') as period,
                   COUNT(DISTINCT date_part('dow', date)) AS periodsCount
            FROM event
            WHERE date ⩾ :startDate
              AND date ⩽ :endDate
              AND auditorium_id = :auditoriumId
              AND date_part('dow', date) IN (:weekdays)
              AND start_time IN (:startTimes)
            GROUP BY start_time
        """, nativeQuery = true)
List<PeriodsAccumulationRaw> genPeriodsAccumulation(
        @Param("startDate") LocalDate startDate,
        @Param("endDate") LocalDate endDate,
        @Param("auditoriumId") int auditoriumId,
        @Param("weekdays") List<Integer> weekdays,
        @Param("startTimes") List<LocalTime> startTimes
);
```

Figure 9. Periods Distribution DB Query

The most informative and the most complex chart is the one that does not accumulate anything, but rather shows all events of interest on a 2D chart where event's date (broken down into week number and a weekday) goes along X-axis, its period number goes along Y-axis and the color of a corresponding segment indicates corresponding events' faculties or requesters names based on the event type.

All of these charts make up a statistic for one auditorium. Having database parameters mentioned above easily choosable via convenient interface and being able to see those charts dynamically generated make up a great tool that can be used to optimize occupation of auditoriums by rearranging existing events and creating new ones.

Section 3. Frontend Implementation details

3.1. Libraries and Architecture

In the modern world there are plenty of options of how the frontend can be implemented. In previous sections it was already discussed why it is better not to go with server-rendering. Nevertheless, there is still a lot to choose from for REST-based frontend. As technologies evolve, it is becoming clear that the most efficient & versatile language to use on frontend is, well, JavaScript (besides, of course, HTML & CSS).

Writing in pure JavaScript is not so efficient for developers, therefore, a lot of libraries and frameworks exist to make the development faster and easier. At the moment this study is written, there are a few options that standout among all others and considered to be one of the best in their field. That's a library ReactJS and frameworks Vue & Angular.

Obviously, all of them have their pros and cons but detailed comparison of them will not be conducted in terms of this study. All of them are decent, but due to the personal preference, a ReactJS was chosen to be the tool that will drive development of KMAuditoriums. Causes of this choice are mostly consist of the ideologies and core principles that React is offering to the developers. More detailed information on this topic can be found in my previous study "Development and optimization of universal web-interfaces using React.js library" [2].

ReactJS is considered a pure UI library that is excellent at dealing with complicated designs because of its component's reusability. But since it is not a framework, it is not shipped with all tools that a frontend developer might want to utilize during the development of something bigger than a TODO app. And one of those missing bits would be a state management.

We have already established that ReactJS' building blocks are components. A component is a reusable piece of code that usually renders some part of the page based on a given props

and internal state. Therefore, it is convenient to image that a website built using ReactJS behaves pretty functionally, meaning that all resulting markup a website will look exactly the given the same combination of all states and props of all components in the application.

In order to simply things for the developers and make the whole app more predictable, not all components have an internal state. In fact, that divides all components into two subgroups: the ones that are aware of the application's state (also called "containers") and the ones that are not (also called "display components"). This setup works great: containers are for the most part "busy" with querying an API server, altering local storage and so on, while display components, given the props by containers (like API call results) take care of rendering the data to the user's screen. The problem arises at the moments when there is too much nested components and it becomes tedious to pass props all the way down from the top container down to a display component or when you have to share some state between several containers or in order to change the state of the container a lot of code is required and is just gets messy.

In order to address those issues, it is a common practice for modern complex apps to use Flux architecture. Basically, what it suggests is to unite all containers states into one "store" that exists apart of containers and components and make all of the state changes happen in a centralized way. That way, business logic of the application moves from containers to a different place, but still providing access for containers to the same information.

Having that, more components can actually become containers without making the app unpredictable, which solves the nesting problem. Also, it becomes clear that a shared unified state can be accessible from multiple places at once, making it reusable across containers. And lastly, Flux suggests a concept of reducers which basically moves every subpart of the business logic to its individual place which makes it much more manageable. Reducers are pure functions that change application's state based on dispatched actions with a type and some payload. Today's standard implementation of Flux architecture is Redux library which is extremely popular and reliable and also has great integration with ReactJS.

Since not so long time ago, ReactJS introduced Hooks concept, which makes Flux architecture implementation even easier thanks to the shared logic possibilities. Prior to that, every business logic part should have been written manually. For example, every asynchronous endpoint request would be implemented with 3 different actions to the store like “make a request”, “finish a request with success” and “finish a request with failure”. Additionally, a middleware would have to be used in order to start an asynchronous request itself after the first dispatched action and then dispatch 2-nd or 3-rd when it resolves or rejects. With hooks, we can write custom hooks that easily can be made generic and configurable enough to be used in a lot of different contexts with minimal effort.

Last but least, since React by default does use code compilers and transformers in order to support all latest ECMAScript standards and JSX (markup inside JS), it is also pretty easy to bring the development a whole new level by introducing Typescript support. During the project build, source code written with Typescript is transformed to widely supported by browsers vanilla JavaScript, just like any other source code written with ECMAScript would be. However, the presence of this technology really bootstraps the development process since developers obtain possibility to introduce interfaces and types into the project that not only helps avoiding incorrect types inside components and API requests, but also greatly improves hinting inside Integrated Development Environments which is a big deal in a modern world with fast development cycles.

3.2. Swagger Client

In previous section we discussed why using Swagger would be beneficial for KMAuditoriums frontend and how it can be generated on backend. Having that, it is time to utilize that JSON file in order to generate an API Requester class.

Luckily for us, quite a few libraries exist for that purpose. One of the most configurable ones and designed for Typescript is called “swagger-js-codegen”. Using NodeJS we can fetch a Swagger file from our backend, provide codegen library with it along with a couple

of template files and we get an output file which we can use later in out containers. With a default Typescript template shipped with a library we can achieve a decent result, but it still would not be perfect considering specific needs of our application, like Authorization. It would inefficient to load authorization token in every container and attach it manually to every request we make. In order to avoid that, I decided to introduce common headers inside an API class that is generated by swagger-codegen. In order to do so, it is enough to copy 2 template files that are shipped with a library into a local project's folder and alter them to satisfy our needs. These templates are written in popular Mustache format and are easy-modifiable. One of the templates files is responsible for generating a class file and second one – for each API method is this class. A few simple lines of codes as well as addition of "auto-bind" library and local-storage clearance whenever we receive responses with status codes 401 (Unauthorized) and 403 (Forbidden) and we are done.

Resulting API class can be used to create singleton instances pointing to an arbitrary domain and with custom dynamic header (like Authorization one) attached to every request. That gives us a great tool that automatically checks arguments types, distributes them properly between body, query and path parameters, specifies HTTP method and request URL. For example, on Figure 10 we can see a fragment of code that makes an HTTP request

```
private request(method: string, url: string, body: any, headers: any,
                queryParameters: any, form: any, reject: CallbackHandler, resolve: CallbackHandler) {
    let req = (request as any as SuperAgentStatic)(method, url).query(queryParameters);

    Object.keys(headers).forEach(key => {
        req.set(key, headers[key]);
    });

    if (body) {
        req.send(body);
    }

    if (typeof(body) === 'object' && !(body.constructor.name === 'Buffer')) {
        req.set('Content-Type', 'application/json');
    }

    if (Object.keys(form).length > 0) {
        req.type( val: 'form');
        req.send(form);
    }

    req
        .then((res : Response ) => {
            if (res.noContent) return Promise.resolve();

            return res.body || JSON.parse(res.text);
        }) Promise<Response>
        .then((body : Response ) => {
            resolve(body);
        }) Promise<void>
        .catch((err) => {
            // force logout
            if ((err.status) === 401 || (err.status) === 403) {
                purgeStoredState(persistConfig)
                window.location.href = '/'
                window.location.reload()
            }
            reject(err);
        });
}
```

Figure 10. Request method of API Client

```
/**
 * getAuditoriumById
 * @method
 * @name KMAuditoriumApi#getAuditoriumByIdUsingGET
 */
getAuditoriumByIdUsingGET(parameters: {
    'id': number,
    $queryParameters ? : any,
    $domain ? : string
}): Promise < any > {
    const domain = parameters.$domain ? parameters.$domain : this.domain;
    let path = '/auditorium/{id}';
    let body: any;
    let queryParameters: any = {};
    let headers: any = {};
    let form: any = {};
    return new Promise( executor: (resolve, reject) => {
        headers['Accept'] = '*/*';
        this.commonHeaders.forEach(function(header) {
            headers[header.name] = [header.value];
        });

        path = path.replace( searchValue: '{id}', replaceValue: `${parameters['id']}`);

        if (parameters['id'] === undefined) {
            reject(new Error('Missing required  parameter: id'));
            return;
        }

        if (parameters.$queryParameters) {
            Object.keys(parameters.$queryParameters).forEach(function(parameterName : string ) {
                queryParameters[parameterName] = parameters.$queryParameters[parameterName];
            });
        }

        this.request( method: 'GET',  url: domain + path, body, headers, queryParameters, form, reject, resolve);
    });
}
```

Figure 11. Get Auditoriums Request

using superagent library that is used by specific requests as a foundation. And on Figure 11 we can see an example request “getAuditoriumByIdUsingGET”. It returns an asynchronous function that checks the presence and type of required number argument “id”, adds appropriate headers from a “commonHeaders” array and makes request to “/auditorium/{id}” URL, adding domain at the beginning and replacing {id} with a passed value. Thanks to “auto-bind” library all of KMAuditoriumApi’s methods can be used as functions without losing their context.

3.3. API Hooks

Given tools that make API requests perfectly clear and easy, it is time to introduce next level abstraction that will make the development of the frontend application even more satisfactory. As we have discussed earlier in the study, each asynchronous request should be accompanied by state changes, specifically, an application’s state must be aware of whether request is being in progress and whether it has succeeded or failed and with what result or error correspondingly. In ordered to do that I introduced a set of custom hooks that work with Redux store.

Assuming, that state management in KMAuditoriums would be rather simple & straightforward, first hook is called “usePersistedState”. It behaves identically to React’s built-in hook called “useState” but instead of using component’s state, it saves everything to the store, using Redux’s dispatch and additional argument called “entityName”.

Having that, we can go ahead and build “useApi” hook, which loads authorization token from Redux state using a hook above and based on that return an instance of the API with appropriate Authorization headers. A code fragment is shown in Figure 12.

```
export default function useApi() {

  const authorizationToken = usePersistedState({
    entityName: entitiesConstants.authorizationToken,
  })[0]

  return useMemo( factory: () => {
    return new Api( options: {
      domain: config.apiBaseUrl,
      commonHeaders: [
        ...(
          authorizationToken
            ? [{ name: 'Authorization', value: `Bearer ${authorizationToken}` }]
            : []
        ),
      ],
    })
  }, deps: [authorizationToken])
}
```

Figure 12. useApi hook implementation

Having that we can introduce “useApiRequest” which, given an asynchronous API function can return states of its result, error and loading flag along with a function that actually initiates a call to that API function. Using Typescript’s generics, we can also properly link request and return signatures of an API function to make this hook even more convenient. It also optionally receives “onSuccess” and “onError” callbacks which can be used as event handlers. A code implementation is shown in Figure 13.

```
export default function useApiRequest<Req, Res>(
  {
    apiMethod,
    onSuccess,
    onError,
    initialValue,
  }: UseApiRequestProps<Req, Res>,
): UseApiPropsReturn<Req, Res> {
  const [loading, setLoading] = useState( initialState: false)
  const [error, setError] = useState( initialState: null)
  const [res, setRes] = useState(initialValue)

  const makeRequest = useCallback( callback: async (props: Req) => {
    setLoading( value: true)
    try {
      const res = await apiMethod(props as Req)
      onSuccess && onSuccess(res)
      setRes(res)
      setError( value: null)
    } catch (err) {
      onError && onError(err)
      setError(err)
    }
    setLoading( value: false)
  }, deps: [
    setLoading,
    apiMethod,
    setError,
    onError,
    setRes,
    onSuccess,
  ])

  return [res, loading, error, makeRequest] as UseApiPropsReturn<Req, Res>
}
```

Figure 13. useApiRequest hook implementation

One final bit that is missing out of the whole picture is hook called “usePersistedApi” which utilizes both “usePersistedState” and “useApiRequest” in order to make a new hook that not only allows conveniently make an API request inside a container, but also to share the results with other containers that may want to utilize the same data without necessarily making requests themselves.

```
const api = useApi()
const [
  auditoriums,
  auditoriumsFetching,
  auditoriumsError,
  getAuditoriums,
] = usePersistedApi({
  apiMethod: api.getAllAuditoriumsBriefUsingGET,
  entityName: entitiesConstants.auditoriumsBrief,
  initialValue: [] as Array<AuditoriumBrief>,
})
useEffect( effect: () => {
  getAuditoriums( props: {})
}, deps: [getAuditoriums])
```

Figure 14. Example of an API usage

An example shown in Figure 14. Suppose we have a container that is responsible for rendering a piece of UI that edits accesses to auditoriums for a specific profile. Logically, one of things to do here is to get auditoriums from the backend to put them as options of a multi-select component. In order to implement this, we at first call “useApi” hook to load an instance of an API. Due to the logic of the application, if current component renders, that means that somewhere a Login Page container has already been rendered and it saved an authentication token to the Redux store. Therefore, an API object that will be returned here will have that authentication token bounded inside its closure and

during all consequent rerenders of the “EditAccesses” container, it will always be the same unchanged instance. Then we utilize this instance in “usePersistedApi” hook, specifying a specific request method (“getAllAuditoriumsBriefUsingGET”), entity name (which is a unique string constant) and an initial value. Since an API instance is not going to change, the whole container is going to rerender only under specific circumstances: whenever a request is initiated and whenever it is finished, which will change, correspondingly, “auditoriums”, “auditoriumsFetching”, “auditoriumsError” which later can be used in display components like auditoriums selectors, loaders and error popups correspondingly. “getAuditoriums” function will always have the same reference thanks to the unchanged API instance and “useCallback” hook we have put inside “useApiRequest”. Therefore, it can be used inside a dependency list of “useEffect” which, in that case, will trigger this function exactly once – after the initial render of a container.

Later, we can utilize similar approach in completely different container but instead of “useEffect” pass “getAuditoriums” as an event handler to a button’s click, for example. That will allow us to render previously fetched auditoriums without making an additional request, but in case it is needed, reload content using a button.

Everything said above gives a great reusable mechanism for an API requests inside an application. It allows developers to reuse entities between containers, track request’s loading and handle their errors with ease. Having that setup in the project, every page of the application will be developed significantly faster and the received result would bring a joy to both developers and product user.

3.4. Code generation

Before diving into the markup, last step that would be nice to take care about is components generation. In in relatively small ReactJS projects, the number of components is growing pretty rapidly as it is generally recommended to make them as small as possible in order to comply with single-responsibility principle and make them as reusable as

possible. Usually, every component is usually consisting of three files which are styles files, markup file and index file which declares component's prop types and shortens its import path by reexporting a markup. As the project grows it becomes more and more annoying to create similar directories manually, therefore, I decided to create a mustache-based utility using NodeJS that, given a new component's name, creates a corresponding directory from 3 template files inside a project. During the development it turned out to be a very helpful console utility that not only saves time but also encourages developer to break down huge components into a few smaller ones since with this utility it is extremely fast and satisfying process to do.

3.5. Design and Markup

Having a preparation process described previously in this section done, almost a half of the KMAuditoriums frontend project can be considered done. Because usually building a markup for a platform like KMAuditoriums takes up only 20-30% of the development time and the rest is usually spent on adding a business logic to that markup. However, thanks to swagger generation tools and our reusable hooks this process is going to look like a constructor where we have to connect big code blocks with simple interfaces. In this coursework we will not go through markup's implementation details, since it carries no useful information, nor this process is highly educational.

The resulting application ended up looking nice. Students & Managers have 2 sections in a side menu, namely, "Find Auditorium" and "Events". Admins do have 2 additional ones: "Accesses" and "Statistics".

"Find Auditorium" tab is shown in Figure 15. Auditoriums can be found by its name or description part. Results can be paginated and even filtered by auditorium's availability on specific dates and periods. Clicking on auditorium from the list opens its brief details.

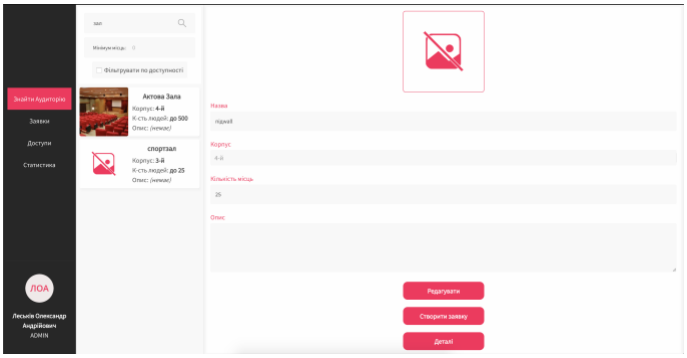

Figure 15. Find auditoriums Page

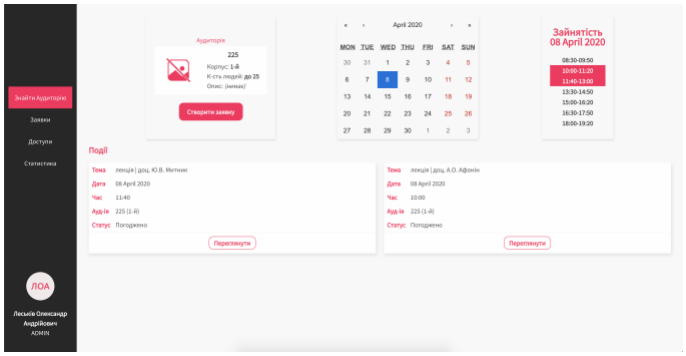

Figure 16. Auditoriums Details

Admins can additionally edit auditoriums using this page. Detailed info is available on another page where user can also see events in the auditorium at a specified date (Figure 16). New events are also created there.

In the events page (Figure 17) admins and managers can see two tabs containing both not-reviewed events as well as the ones that he/she has already approved or declined. Similarly, students see there their own pending and not-pending events. Event details view can is shown in Figure 18.

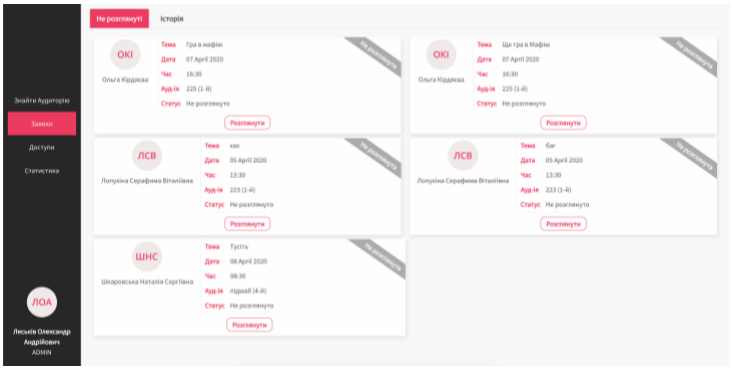

Figure 17. Events Page

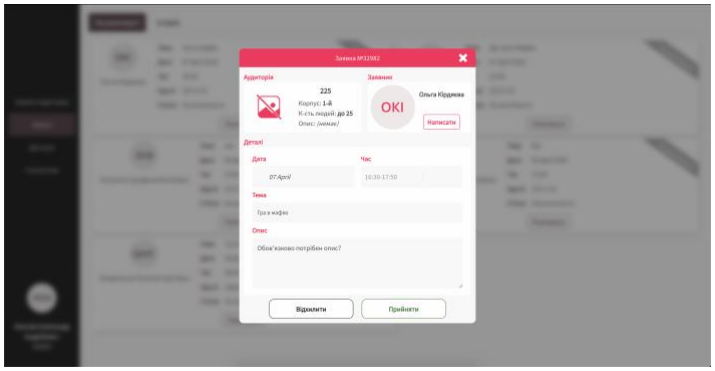

Figure 18. Events Details

Accesses page allows to paginate users in the system and grant them accesses to auditoriums like shown in Figure 19.

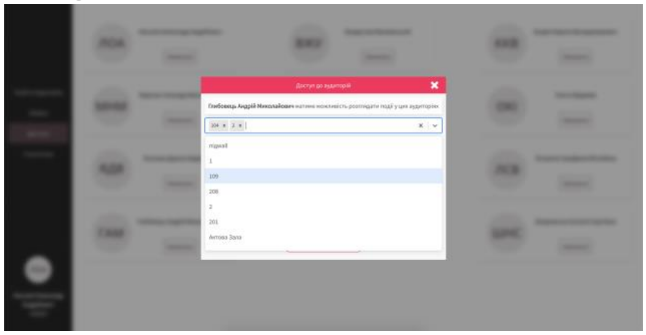

Figure 19. Accesses page

Lastly, in the “Statistics” tab shown in Figure 20, we can see four selectors that allow administrator to select auditoriums of interrest as well as wekks, weekdays and periods. After hitting an “Apply” button, the admin can see statistics like shown in Figure 21.

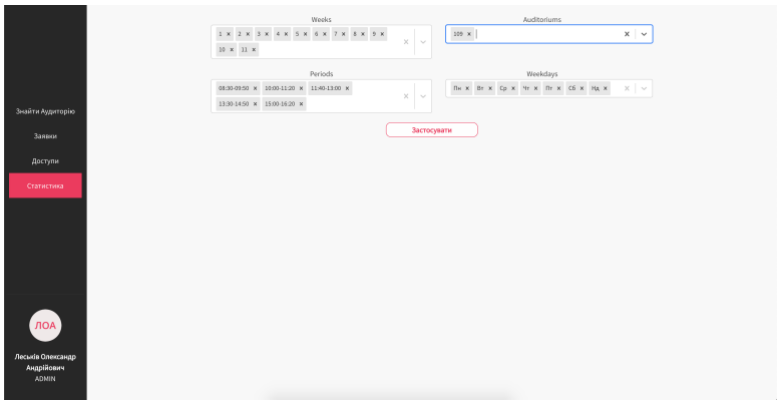

Figure 20. Statistic selectors

The look and behavior of a resulting website corresponds to and fulfills all of the requirements described in the Business Requirements section.

With that being said, this platform is ready to be deployed and filled with data.

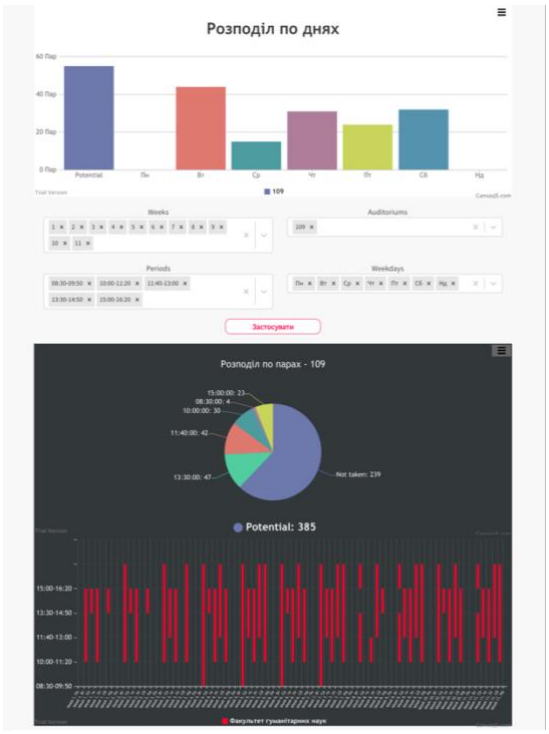

Figure 21. Statistics Example

Section 4. Deployment

After implementation of both backend and frontend, it is now time to make that code available for somebody else. In this section we will see how both backend and frontend will be deployed to server and tied together.

Nowadays the most progressive way to deploy applications is via a Docker. In a nutshell, this technique allows developers to setup deployment environment locally without worrying about the machine being used as a server and its environment. It is particularly useful not only during incremental deployments but also whenever automatic horizontal scaling is required for an application.

```
FROM node:10
WORKDIR /kmauditoriums-web

RUN yarn global add serve

ADD ./package.json ./package.json
RUN yarn

ARG REACT_APP_API_BASE_URL
ENV REACT_APP_API_BASE_URL=$REACT_APP_API_BASE_URL

ARG SWAGGER_URL
ENV SWAGGER_URL=$SWAGGER_URL

ADD . .
RUN chmod +x ./launch.sh
EXPOSE 5000
CMD ["./launch.sh"]
```

Figure 22. Web Dockerfile

```
#!/bin/bash

until $(curl --output /dev/null --silent -k --head --fail "$SWAGGER_URL"); do
    sleep 5
done

echo 'SwaggerFile from core is available!'
yarn build:client "$SWAGGER_URL"
yarn build
cd ./build && yarn serve-https
```

Figure 23. Supplementary web Dockerfile script

First step in the implementation would be creating two Dockerfile-s. The one for web is shown in Figure 22. Supplementary launch.sh file is shown in Figure 23. A node:10 image is used as a base because NodeJS is required to convert source of a React app into plain html, js & css files. It also builds an API client prior to that and since a swagger file is generated dynamically in the API service, a build process had to be moved from Dockerfile step to a script that is launched upon within the container. It takes both Swagger URL and REACT_APP_API_BASE_URL as its arguments. We will later setup Swagger URL to point to an API service placed in the sibling container and REACT_APP_API_BASE_URL – to a url where the application will be hosted.

Dockerfile for our API service, a Spring Boot application, is shown in Figure 24. Similarly the web, the whole application cannot be built entirely in the Dockerfile, because

it dynamically requires database connection in order to run migration, therefore, a Dockerfile is provided with an additional script that is launched upon the container creation that build and launches Spring Boot application (Figure 25).

```
FROM maven:3.6.3-jdk-13
WORKDIR /kmauditoriums-api
COPY pom.xml .
RUN mvn dependency:go-offline

COPY src/ ./src
ADD launch.sh ./
RUN chmod +x ./launch.sh
EXPOSE 8080
CMD ["./launch.sh"]
```

Figure 24. Spring Boot Dockerfile

```
#!/bin/bash

mvn package
java --enable-preview -jar ./target/kmauditoriums-api-1.0.1.jar
```

Figure 25. Spring Boot launch script

To tied these together, a simple docker-compose was written that launches these services as well as a Postgres database with a mapped to the host volume in order to preserve data upon relaunches.

In order to launch this setup on a server, a Docker had to be installed there, and a couple of SSH keys generated in order to allow it access web and API repositories. Since the office 365 authentication requires websites it is being utilized at to use HTTPS protocol, a domain called “kmauditoriums.fun” was bought. I used a free online service to generate self-signed certificates and put them directly to the server using SSH. A couple of TXT records to verify certificates on a domain as well as a “A record” pointing to the NaUKMA virtual server used for deployment in combination with Spring Boot’s https configuration and Node JS SPDY package for static files serving, and the setup is almost done.

After the application was successfully deployed and accessible through a public domain, a corresponding URL was added to an Azure active directory. Having that almost completes the whole configuration of a production application. The last uncompleted part lays in the absence of a data with which it can work. In order to fix that, I used KMAScheduler. It is a another NaUKMA service which has information about the schedule that is automatically

formed from schedule documents accessible on the official NaUKMA website. Since I'm a tech lead developer of that project, I have access to the production database of that service. In order make a scheduler transfer, I connected two databases to my computer using SSH tunneling like shown in Figure 26.

```
ssh -fN -L 8100:localhost:2605 leskiv@194.44.142.74
ssh -fN -L 8101:localhost:5432 lknm@194.44.143.139
```

Figure 26. SSH tunneling for databases

Having that, it was easy enough to create a Python script that launched a query on a KMAScheduler's database shown in Figure 27 and used data to generate a couple of tens of thousands schedule events along with their auditoriums, buildings and faculties. A script used partially shown in Figure 28.

```
SELECT replace(
         regexp_replace(
           regexp_replace(
             lower(lessons.auditorium),
             '.*кмі.*', '4-кмі'),
           '.*спортзал.*', '3-спортзал'),
         ' ', ''
       ) AS auditorium,
       lessons.weekday,
       lessons.start_week,
       lessons.end_week,
       lessons.start_time,
       lessons.end_time,
       subjects.full_name,
       groups.full_name,
       faculties.full_name
FROM lessons
       INNER JOIN groups ON lessons.group_id = groups.id
       INNER JOIN subjects ON groups.subject_id = subjects.id
       INNER JOIN specializations ON subjects.specialization_id = specializations.id
       INNER JOIN faculties ON specializations.faculty_id = faculties.id
WHERE replace(lessons.auditorium, ' ', '') != ''
  AND NOT lessons.auditorium LIKE '%*%'
  AND NOT lessons.auditorium LIKE '%?%'
  AND NOT lessons.auditorium LIKE '%,%'
  AND NOT lessons.auditorium LIKE '%уточн%'
  AND NOT lessons.auditorium LIKE '%ІАНАНУ%'
  AND NOT lessons.auditorium LIKE '%ЦПЕС%'
  AND NOT lessons.auditorium LIKE '%Центр%'
  AND NOT lessons.auditorium LIKE '%центр%'
  AND NOT lessons.auditorium LIKE '%' || chr(10) || '%'
ORDER BY lessons.auditorium
```

Figure 27. KMAScheduler lessons query

```
trimester_start = date(2020, 1, 13)
lessons = kmascheduler.fetchall()

for i, lesson in enumerate(lessons):
    print(f"\rProcessing {i + 1}/{len(lessons)}", end='..')
    auditorium = lesson["auditorium"]
    weekday = lesson["weekday"]
    start_week = lesson["start_week"]
    end_week = lesson["end_week"]
    start_time = lesson["start_time"]
    end_time = lesson["end_time"]
    subject_name = lesson["subject_name"]
    group_name = lesson["group_name"]
    faculty_name = lesson["faculty_name"]

    for week in range(start_week, end_week + 1):
        if '-' not in auditorium:
            print(auditorium)
            continue

        lesson_date = trimester_start + timedelta(weeks=(week - 1), days=weekday)
        building_number, auditorium_name, *rest = auditorium.split('-')

        faculty_id = get_faculty_id(faculty_name)
        building_id = get_building_id(f'{building_number}-й')
        auditorium_id = get_auditorium_id(auditorium_name, 25, building_id)

        kmauditoriums.execute("""
            INSERT
            INTO event(
                auditorium_id, faculty_id,
                title, description,
                date, start_time, end_time,
                type, status
            )
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)
        """, [
            auditorium_id, faculty_id,
            group_name, f"Предмет: {subject_name}",
            lesson_date, start_time, end_time,
            "SCHEDULE", "APPROVED"
        ])
```

Figure 28. KMAuditoriums data filler script

Conclusions

During this coursework an application called “KMAuditoriums” was developed to simplify auditorium’s occupation management in National University of “Kyiv-Mohyla Academy”. It allows students to easily create requests to organize events in NaUKMA, which can be reviewed by platform administrators or managers that were granted a management access to corresponding auditoriums by platform administrators. Besides student-organized events, managers and platform administrators are also capable of creating events and even associate them with a specific faculty. Naturally, NaUKMA’s auditoriums can be occupied not only with application-based events but also with schedule ones. For that purpose, a service is also capable to communicate with a schedule service and obtain the data about lectures, practice sessions and other types of educational processes happening in NaUKMA.

Auditoriums initially appear in the service via a schedule service. However, they can be edited by platform administrator. Therefore, their names can be changed, description added as well as their photo and capacity information. Every user of the service can utilize auditoriums search in order to find an auditorium accordingly to his/her needs. Approved events are also visible to everybody so users can plan their events based on their information.

Additional useful feature for a service would be an auditoriums statistic. This powerful tool has been admitted useful by people in leadership positions of NaUKMA during this study. It allows to overview lessons distributions among weeks, weekdays and periods for any auditorium with arbitrary ranges of weeks, weekdays and periods, which can be extremely helpful during optimization of auditoriums occupation before trimesters.

This coursework also encouraged me to learn new technologies and improve my skills working with the ones I have already been familiar with. Specifically, I learned how to use Spring Boot to create simple but production-ready backend and I also developed a few new

approaches for frontend development using ReactJS library in order to make the whole development process easier and more enjoyable.

I also took care of deployment of the whole service and also managed to replace not yet existing Schedule Service with simple Python backend that using KMAScheduler's Telegram bot database filled information about auditoriums and lessons that are happening in NaUKMA based on the official schedule that can be found on my.ukma.edu.ua, which made the service look complete. A platform was deployed to a NaUKMA's server and mapped to a domain bought on GoDaddy service.

This project is not over yet as integration with other NaUKMA eco-system will take place at some point of time in the future, but everything was done to simply that process when the time comes.

References

1. REST [Online resource] - Representational state transfer. Access mode: https://en.wikipedia.org/wiki/Representational_state_transfer.
2. React AAD MSAL [Online resource] - Microsoft Authentication Library with Azure Active Directory for a React app. Access mode: https://www.npmjs.com/package/react-aad-msal.