CRITERION A

Defining the problem

‘Ewenty’ is a small company which specializes in organizing sport competitions (called events). The events range in types (e.g. football match or basketball match), and each has a specific number of places available. Currently the company has two venues for its events: a sports field and a swimming pool. Attending an event requires the client to book it in advance - currently this can be done by calling the company’s office and agreeing on a date. The company is growing, so phone booking event places starts to become very inconvenient. Five months ago, when I attended a swimming competition organized by ‘Ewenty’, a worker told me about the issue. I realized that this was a good opportunity for my IA, so I offered to build a website. The company accepted my proposition and we scheduled a meeting to further discuss the details.

Rationale for solution

The website for company ‘Ewenty’ will be divided into two sections: a section for company workers and clients. All users, after registration, will be able to book and unbook event places. In additional company workers will be able to create, edit and delete events. They will also be able to view, which users booked which events, as well as, details the users gave during registration. This will help solve the company’s problem, because users will be able to book events at any time, without any interaction with the company’s staff, which, on the other hand, will have all the users, who plan to attend a specific competition, grouped.

The website will consist of two parts: the front-end and an API. The API will be built in Node.js – a JavaScript runtime environment – using the Express framework. I decided to use Node mainly because I am already familiar with it. I also think that it is a good choice for this project because it is scalable and fast, so it will be able to serve the company even if it grows very large. Node.js also has NPM (Node Package Manager) built in, which gives developers access to thousands of libraries and therefore makes development easier. I will use Express.js because it provides a level abstraction for Node.js and therefore makes development faster. The API will also have to communicate with a database, specifically MongoDB, which will be used to store users and event details. I decided to use this NoSQL database because I think it will work well Node.js as it stores records in JSON format, which is very convenient to use with JavaScript.
For my front-end I decided to use React.js. First of all, React makes it possible to easily create a single page application which my client requested. Secondly, React provides a component based structure which allows greater reusability and therefore better code organization.

Success criteria

1. Clients and admins can login with existing accounts
2. Clients can create accounts themselves

3. Admins (company workers) can add and edit events
4. Admins can delete events which causes the cancelation of reservations for all users who booked that specific event
5. Admins can view users’ information
6. Users can book/unbook events
7. If the number of places for an event has been exceeded the website doesn’t allow any additional users to book a place for this event
8. Website doesn’t refresh
9. All user inputs must be validated
10. Clients can book both a swimming and field event on the same day

Words: 457

CRITERION B

Database schemas:

User schema:

```
{
    name: String,
    surname: String,
    address: {
        city: String,
        street: String,
        number: String
    },
    password: String,
    email: String,
    joinDate: String,
    admin: {type: Boolean, default: false},
    swimmingEvents: [{type: Schema.Types.ObjectId, ref: 'Swimming'}],
    fieldEvents: [{type: Schema.Types.ObjectId, ref: 'Field'}]
}
```

Swimming event schema:

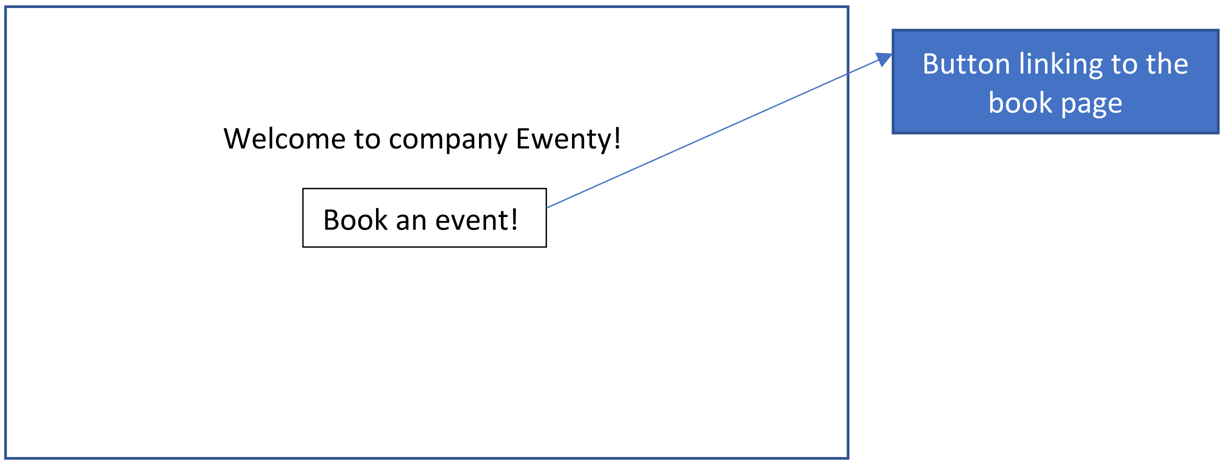
```
{
    jacuzziAvailable: {type: Boolean, default: false},
    style: String,
    date: Date,
    places: Number,
    users: [{type: Schema.Types.ObjectId, ref: 'User'}]
}
```

Field event schema:

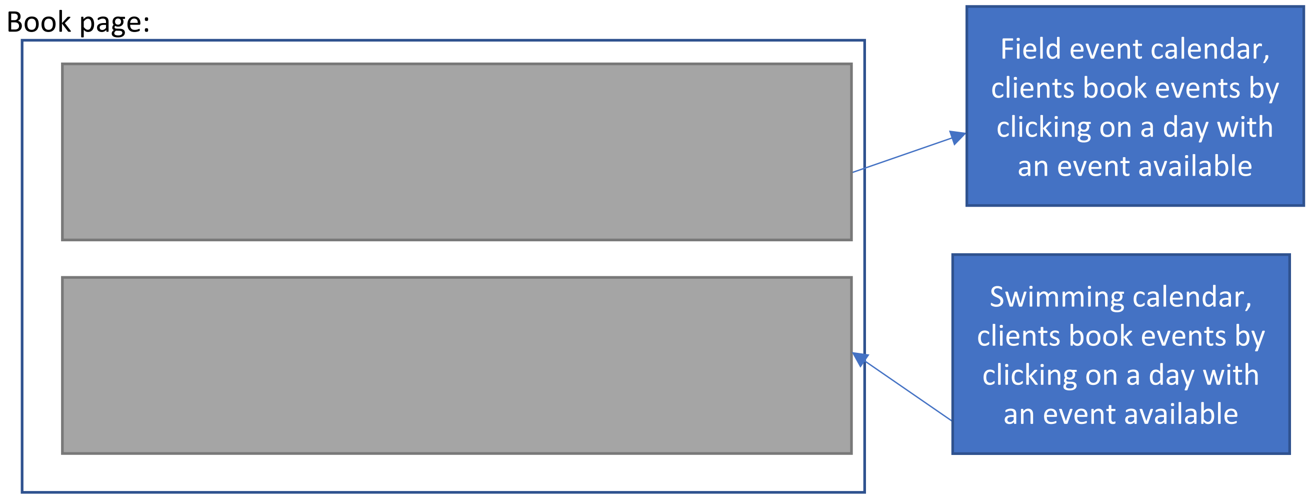
```
{
    type: String,
    date: Date,
    places: Number,
    users: [{type: Schema.Types.ObjectId, ref: 'User'}]
}
```

Interface:

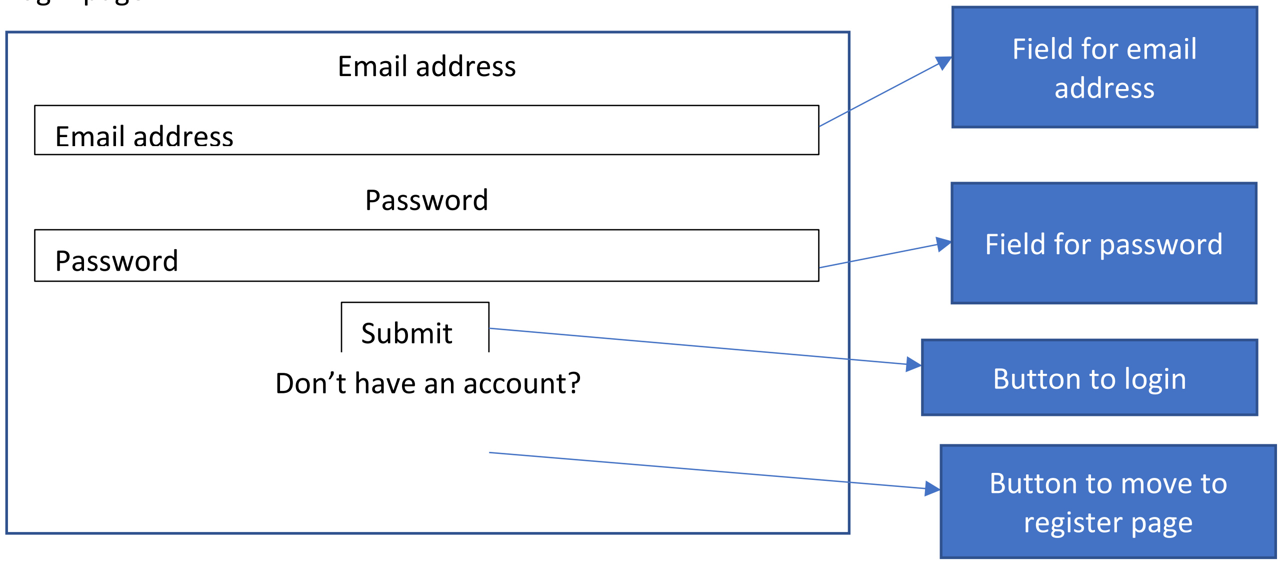
Home page:



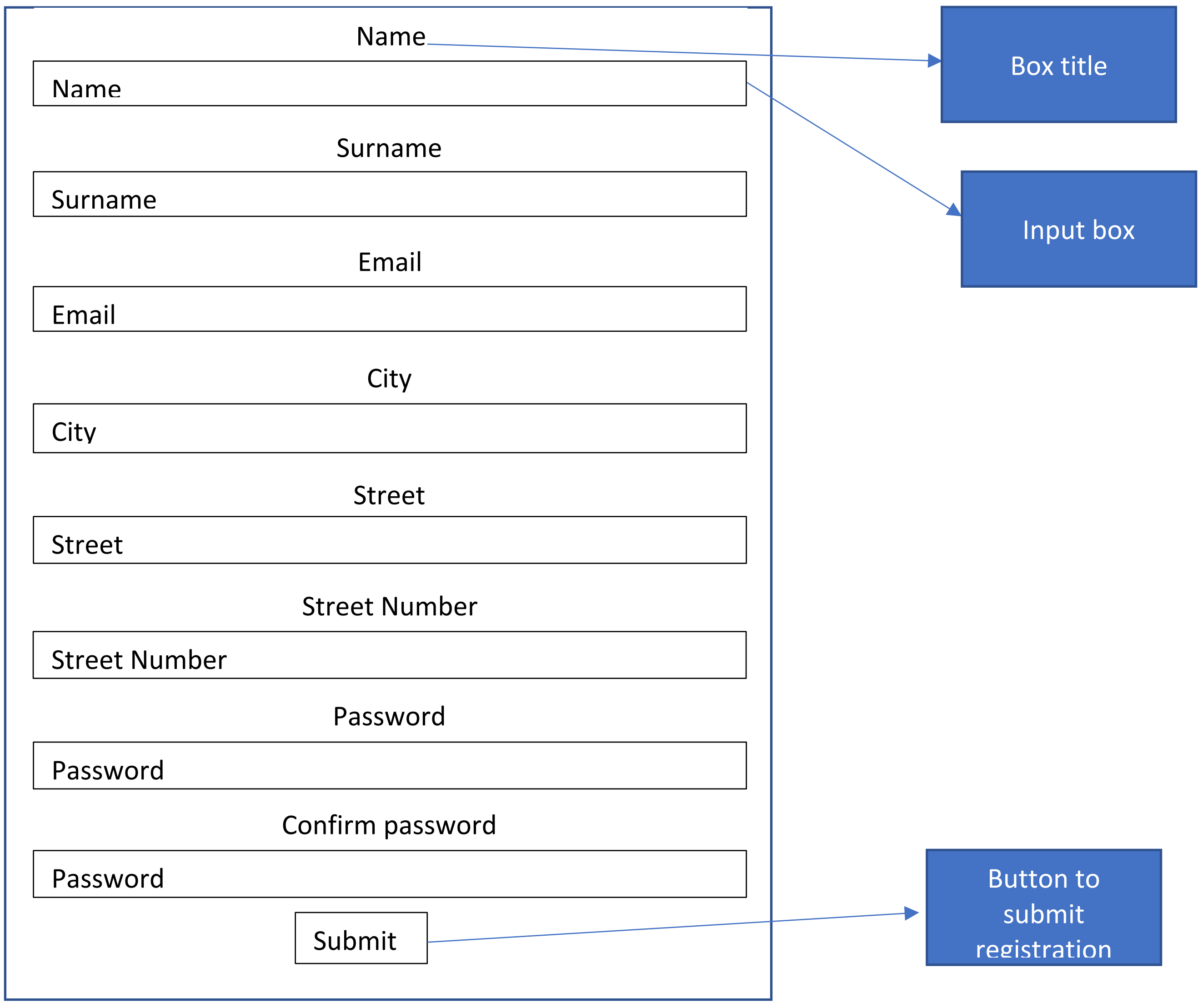
Book page:


Field event calendar, clients book events by clicking on a day with an event available
Swimming calendar, clients book events by clicking on a day with an event available

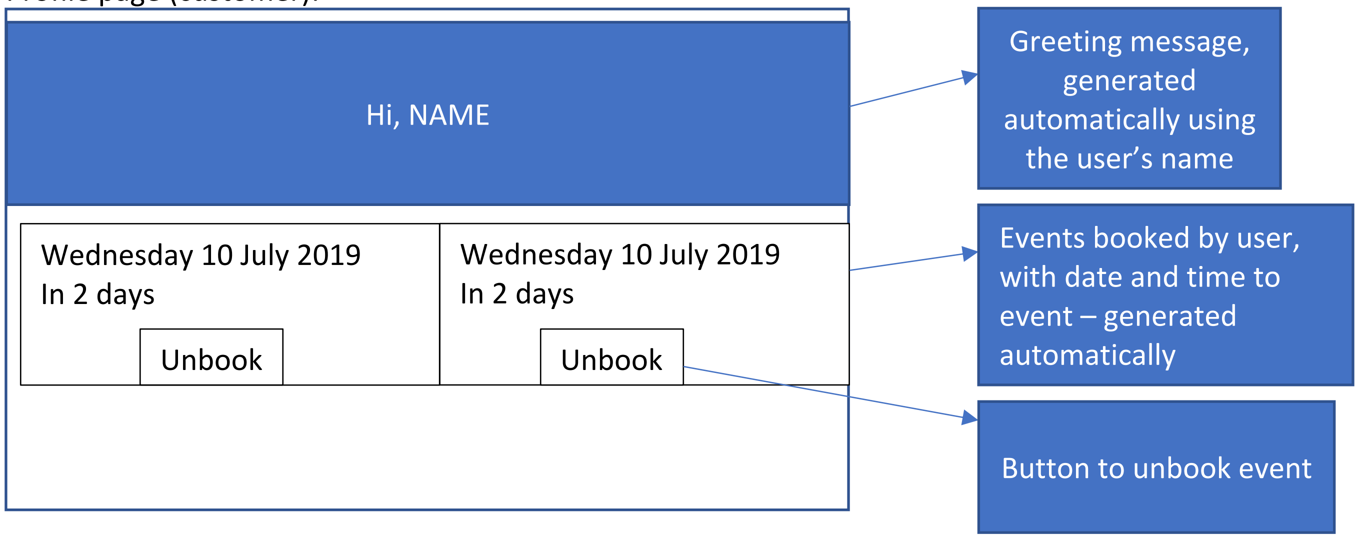
Login page:



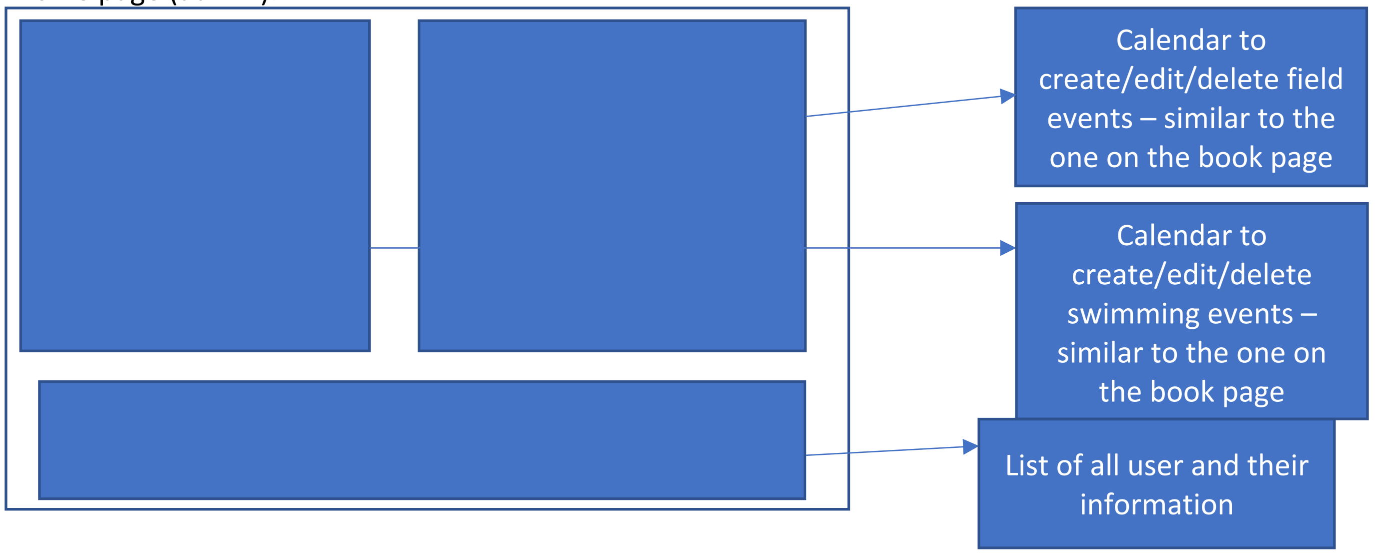
Register page:



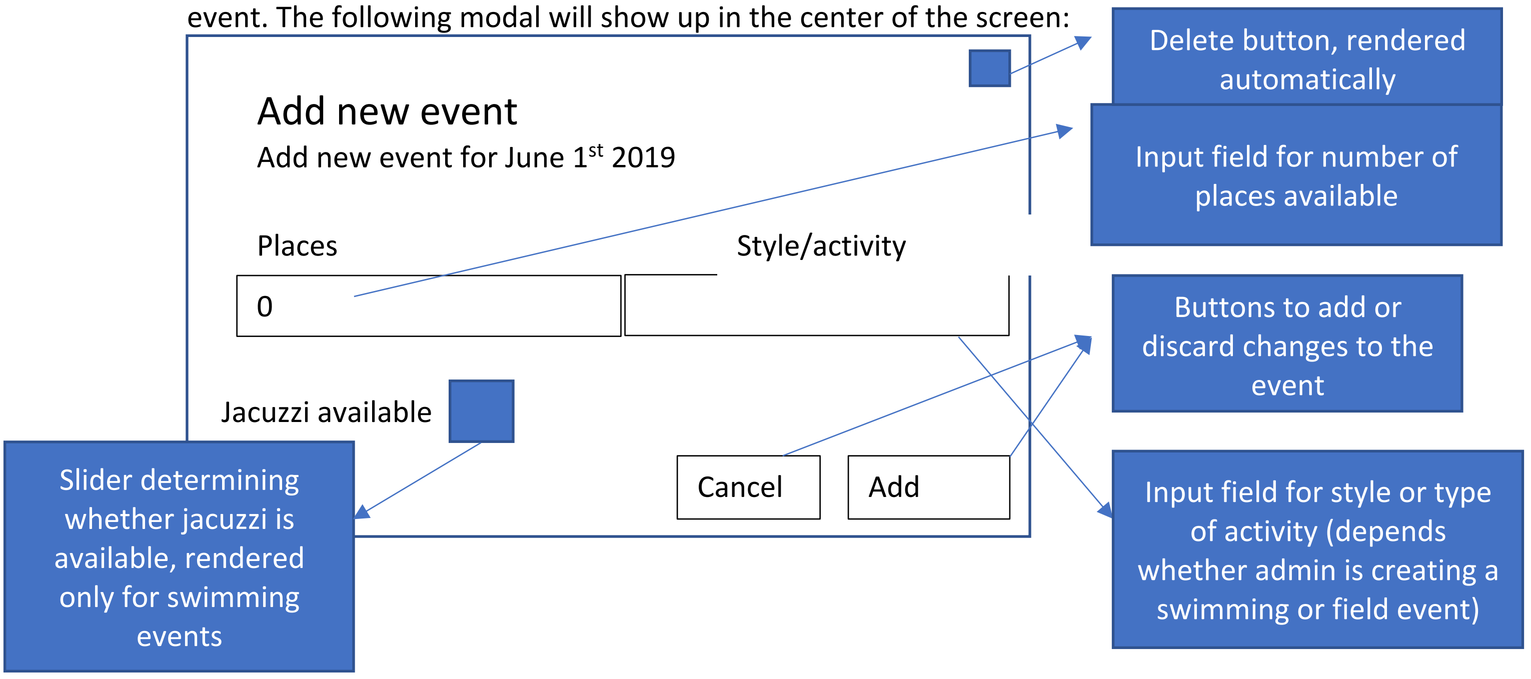
Profile page (customer):


Hi, NAME
Wednesday 10 July 2019
In 2 days
Unbook
Wednesday 10 July 2019
In 2 days
Unbook
Greeting message, generated automatically using the user’s name
Events booked by user, with date and time to event – generated automatically
Button to unbook event

Profile page (admin):

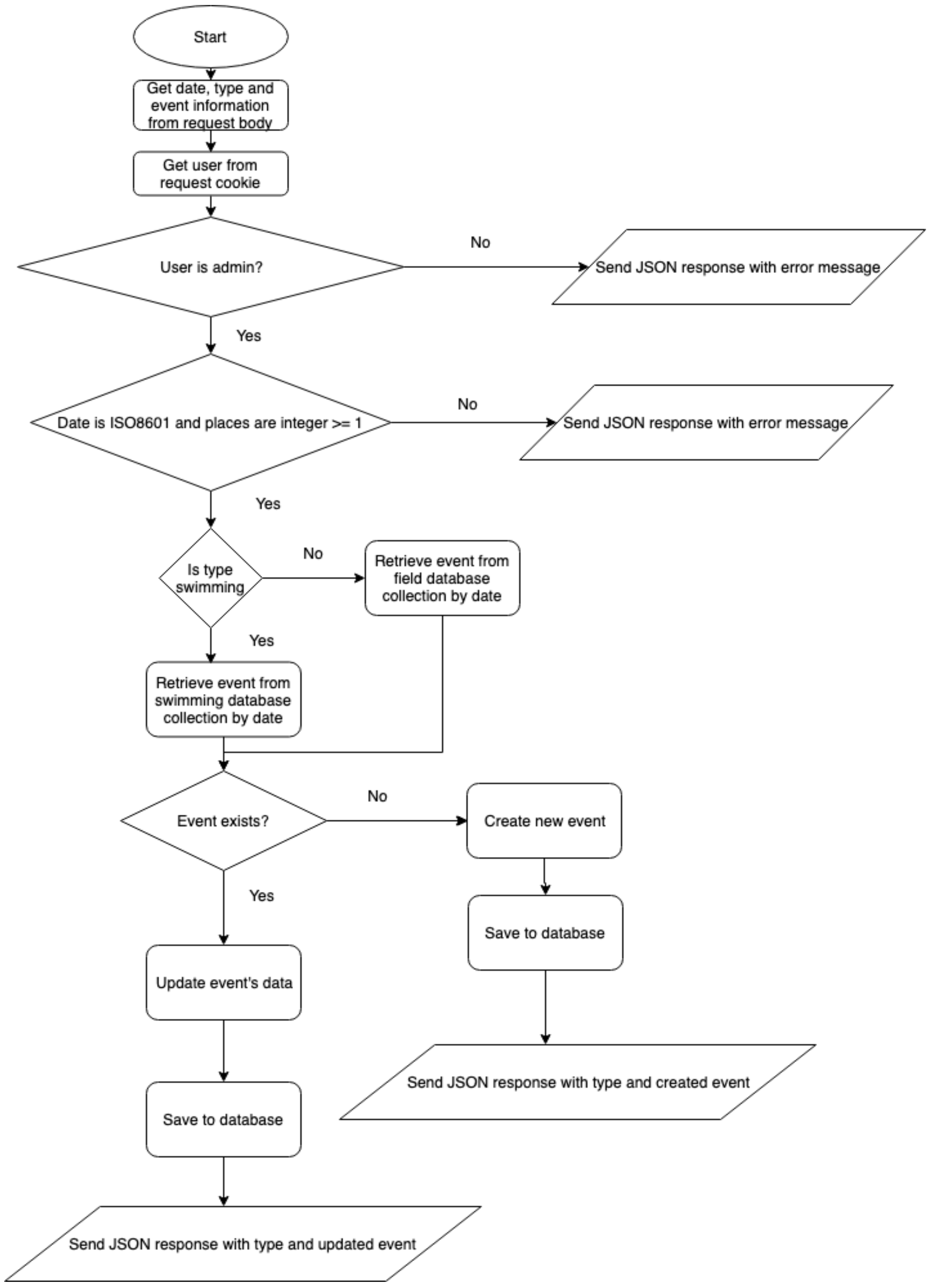

Calendar to create/edit/delete field events – similar to the one on the book page
Calendar to create/edit/delete swimming events – similar to the one on the book page
List of all user and their information

To create/edit/delete an event, the admin must click on the day, on the calendar, of the event. The following modal will show up in the center of the screen:

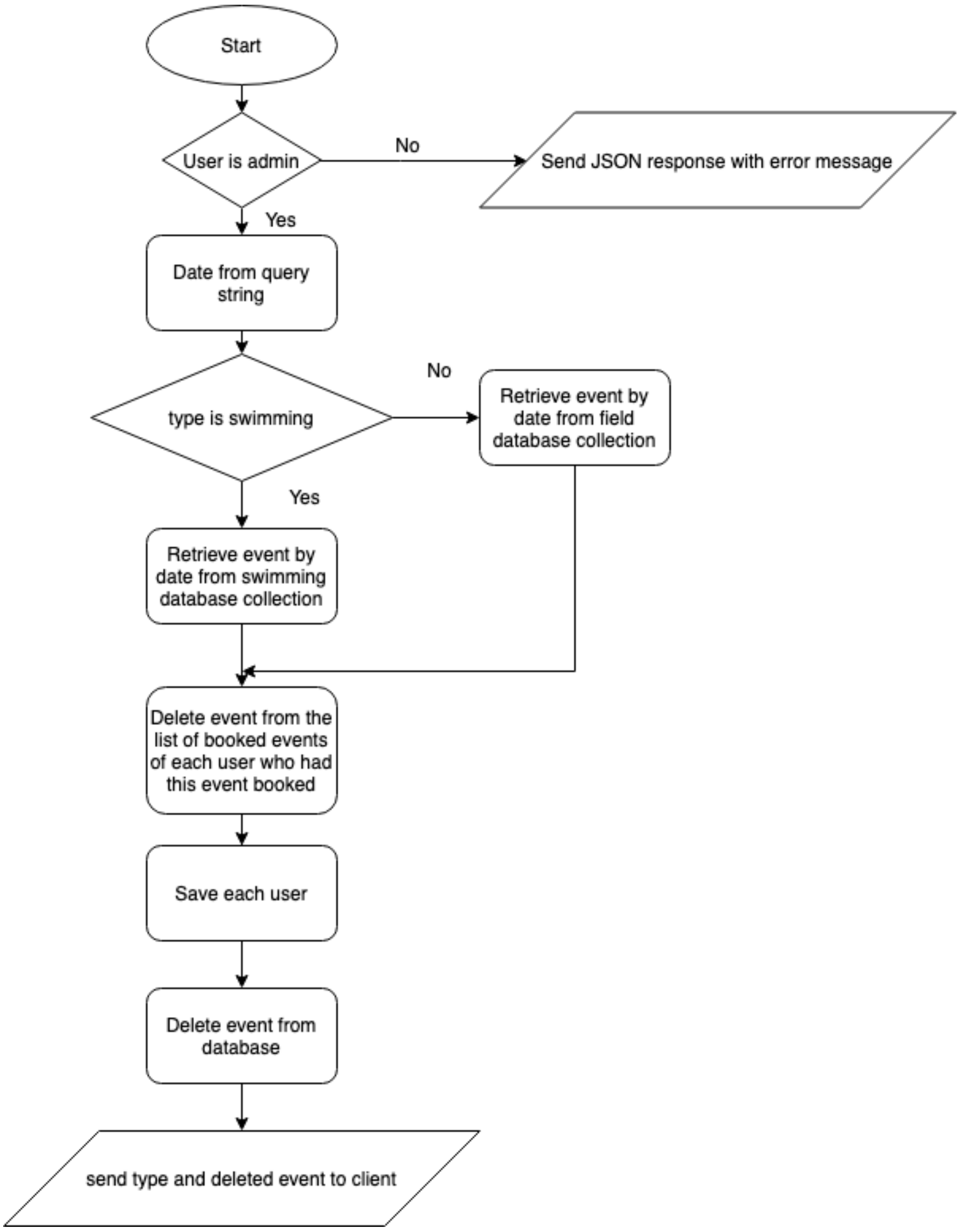

Add new event
Add new event for June 1st 2019
Places
0
Style/activity
Jacuzzi available
Cancel
Add
Delete button, rendered automatically
Input field for number of places available
Buttons to add or discard changes to the event
Input field for style or type of activity (depends whether admin is creating a swimming or field event)
Slider determining whether jacuzzi is available, rendered only for swimming events

Flowcharts

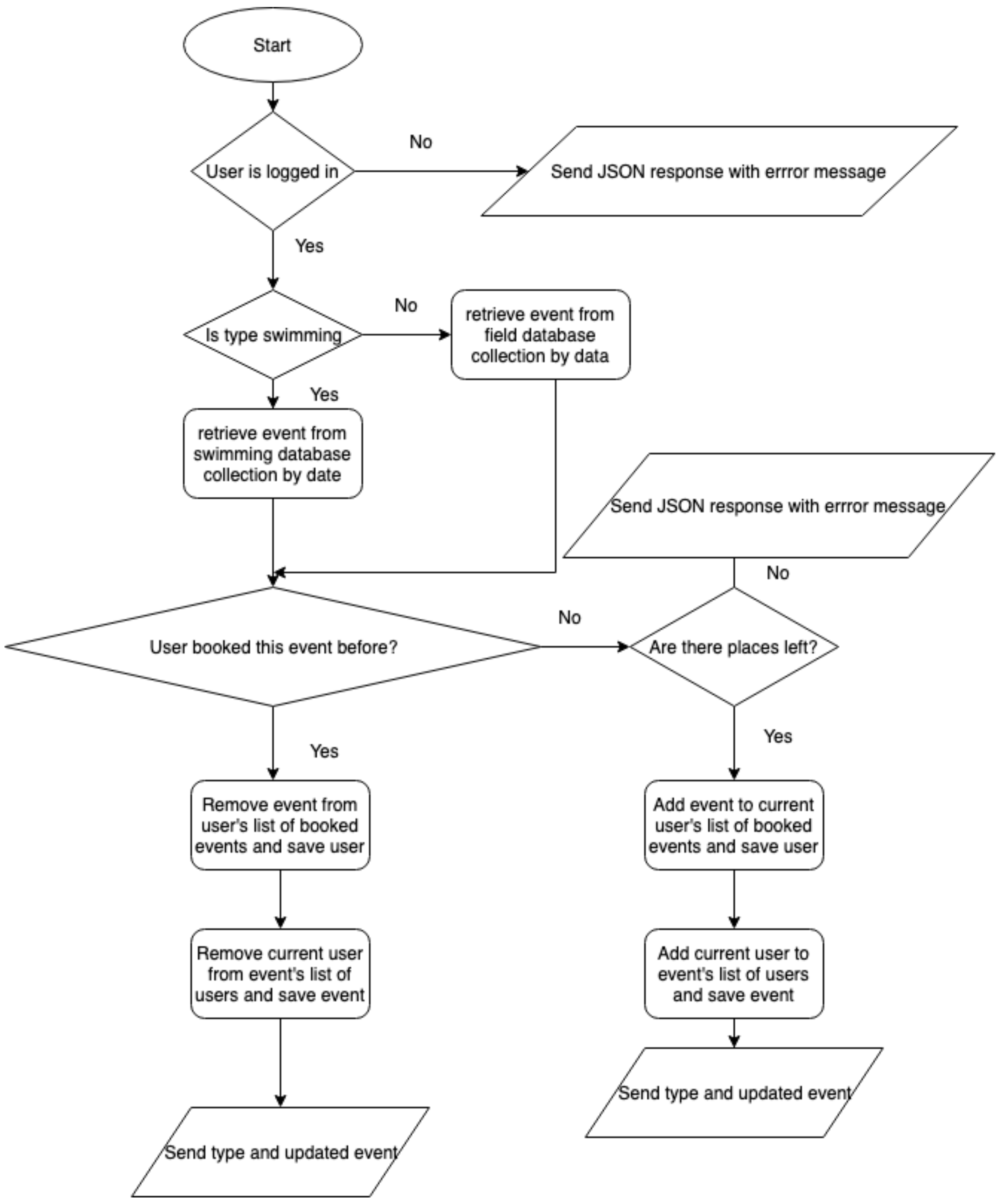
Create/update event


Start
Get date, type and event information from request body
Get user from request cookie
User is admin?
No
Send JSON response with error message
Yes
Date is ISO8601 and places are integer >= 1
No
Send JSON response with error message
Yes
Is type swimming
No
Retrieve event from field database collection by date
Yes
Retrieve event from swimming database collection by date
Event exists?
No
Create new event
Save to database
Send JSON response with type and created event
Yes
Update event's data
Save to database
Send JSON response with type and updated event

Delete event


Start
User is admin
No
Send JSON response with error message
Yes
Date from query string
type is swimming
No
Retrieve event by date from field database collection
Yes
Retrieve event by date from swimming database collection
Delete event from the list of booked events of each user who had this event booked
Save each user
Delete event from database
send type and deleted event to client

Book/unbook event


Start
User is logged in
No
Send JSON response with errror message
Yes
Is type swimming
No
retrieve event from field database collection by data
Yes
retrieve event from swimming database collection by date
Send JSON response with errror message
No
User booked this event before?
No
Are there places left?
Yes
Yes
Remove event from user's list of booked events and save user
Add event to current user's list of booked events and save user
Remove current user from event's list of users and save event
Add current user to event's list of users and save event
Send type and updated event
Send type and updated event

Test plan

Action to be tested	Test method	Expected result	Success criteria
Clients are able to register	Input appropriate data into the form on the register page. Submit.	User created in database	2
Users are able to login	Go to the login page and input correct credentials (of user who	User cookie, which corresponds to the appropriate user, gets created.	1

	is already in the database).		
Admins can add events	Go to the admin dashboard. Select a date on a calendar. Input number of places to the modal being displayed. Click ‘submit’.	Event gets added to the database with appropriate data.	3
Admins can edit events	On the calendar (on the admin dashboard), click on a day which has an event already created. Change number of places and submit.	Event in database has the number of places changed.	3
Admins can delete events	On the calendar (on the admin dashboard) click a day which has an event on it. Click the delete icon on the model.	Event no longer exists in database. The reservation of this event is canceled from all users who booked it.	4
Admins can view users’ information	On the admin dashboard check, with database, if the table contains correct information.	Table contains all the user records from the database.	5
Users can book events	On the book page click on a day with an event available.	Event was added to user’s list of booked events in database.	6
Users can unbook events	On the book page click on a previously booked event.	In database, event was removed from user’s list of booked events.	6
Don’t allow to book event if no places are left	Check if the event which has all of its places occupied is disabled on the calendar (i.e. clicking it doesn’t trigger any action)	Event is disabled on the calendar.	7
Page doesn’t refresh	Go to each page of the application (using the navbar links), check if the page doesn’t reload.	Website doesn’t refresh at all.	8
Input validation	Input inappropriate data to each form.	Error message displayed and no other action taken.	9

Clients can book both a swimming and field event on the same day	On the book page click on a field and swimming event on the same day in order to book each one of them.	Both events were added to the user's list of booked events in the database	10

CRITERION C

Dependencies

Backend:

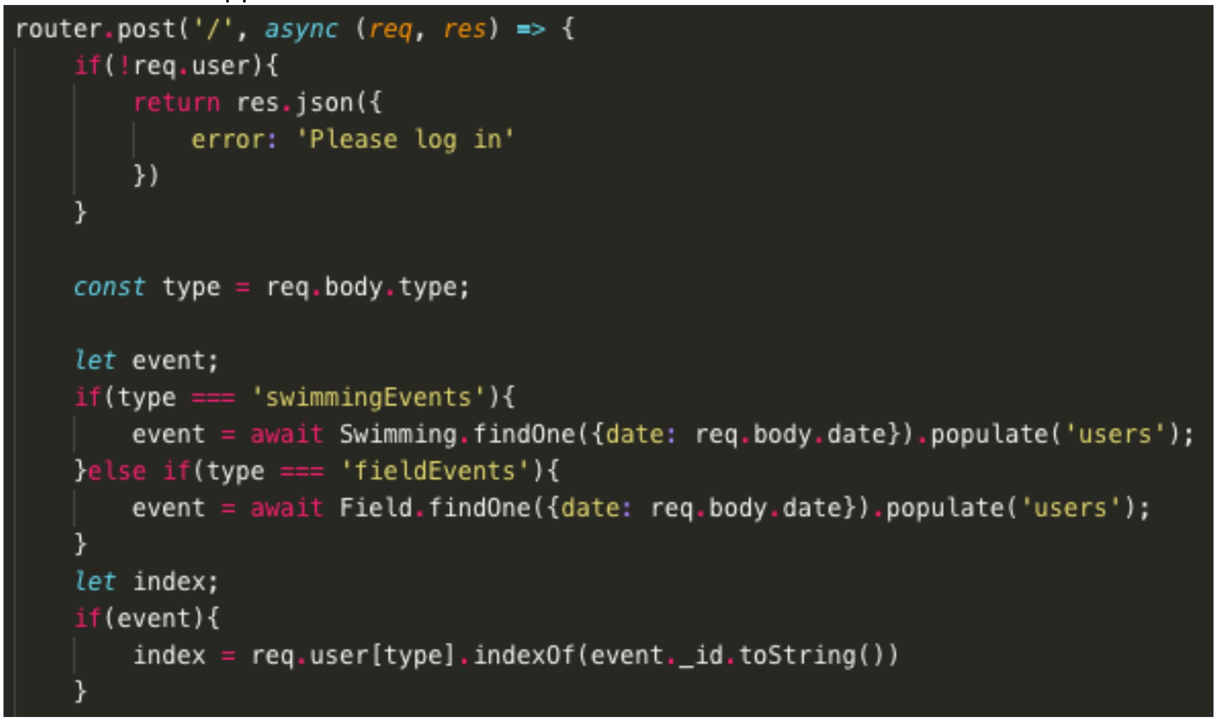
1. Bcrypt.js – used for hashing and comparing hashed users' passwords
2. Body-parser – used for parsing the body of POST requests
3. Cookie-session – used to create a session cookie on the client when a user logs in
4. Express – Node.js framework which speeds up development
5. Express-validator – used to validate user input
6. Mongoose – Used to communicate with the database (MongoDB)
7. Passport & passport-local – used for user authentication

Frontend:

1. Material-ui – pre-built components, used to make the website look nice
2. Axios – HTTP client, used to send HTTP requests to the server
3. Moment – Used for parsing dates
4. React – library for creating user interfaces
5. React-calendar – calendar component
6. React-redux – Binds react with redux
7. Redux – used as a global store of data on the client

Handling asynchronous code

A lot of actions in this web app, such as database queries and HTTP requests, are asynchronous – they run separately from the primary application thread. This asynchronous nature is very useful because it means that the execution of code isn't blocked by a single action that might take long (e.g. a database query might take 100 milliseconds) and in result makes the web app run faster.

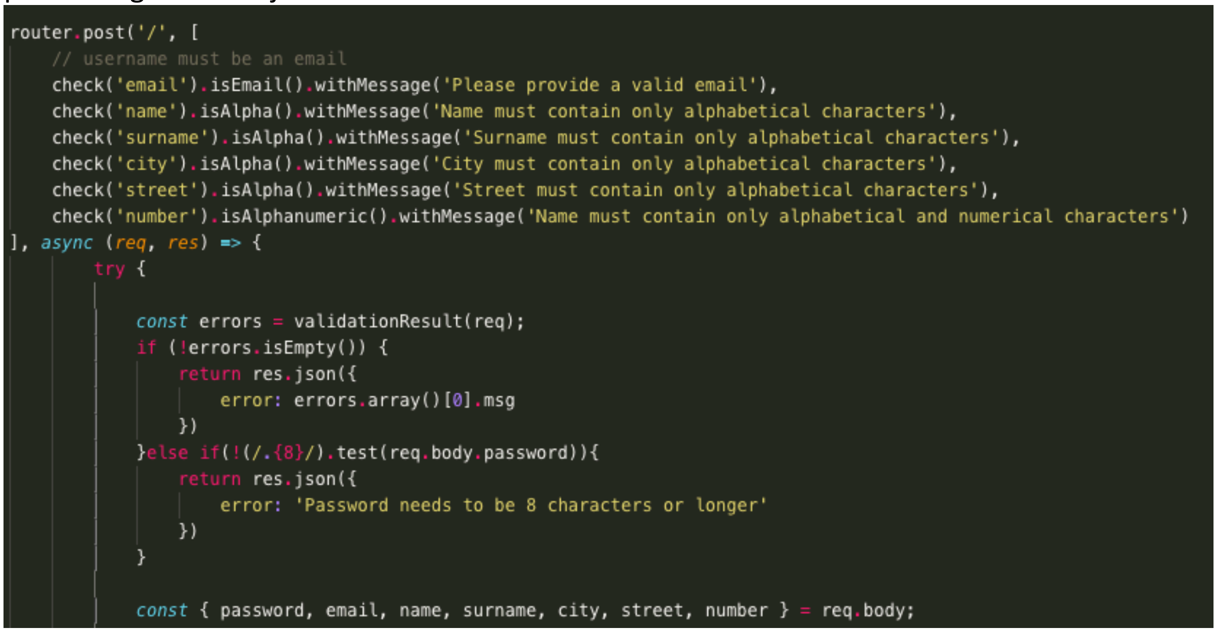


```
router.post('/', async (req, res) => {
    if(!req.user){
        return res.json({
            error: 'Please log in'
        })
    }

    const type = req.body.type;

    let event;
    if(type === 'swimmingEvents'){
        event = await Swimming.findOne({date: req.body.date}).populate('users');
    }else if(type === 'fieldEvents'){
        event = await Field.findOne({date: req.body.date}).populate('users');
    }
    let index;
    if(event){
        index = req.user[type].indexOf(event._id.toString())
    }
```

The code above is a part of the server route responsible for booking an event (a flowchart of it was presented in the Design section). In the first line an asynchronous callback function is created using the *async* keyword. Asynchronous functions allow to use the *await* keyword inside them which pauses the execution of this function and waits for the passed Promise's resolution. This allows for a very clear syntax and therefore good code readability. An example of the use of the *await* keyword can be seen in line 12. Here, it pauses the code until the query of the Swimming event collection finishes and is assigned to the event variable. Without this *await* keyword the code would just proceed without waiting for the database query to finish and in result the event variable would be undefined.

Validation

All user input is validated on the server to make sure that it's in a correct format. This not only prevents errors on the server but also greatly increases security by, for example, preventing NoSQL injection attacks.

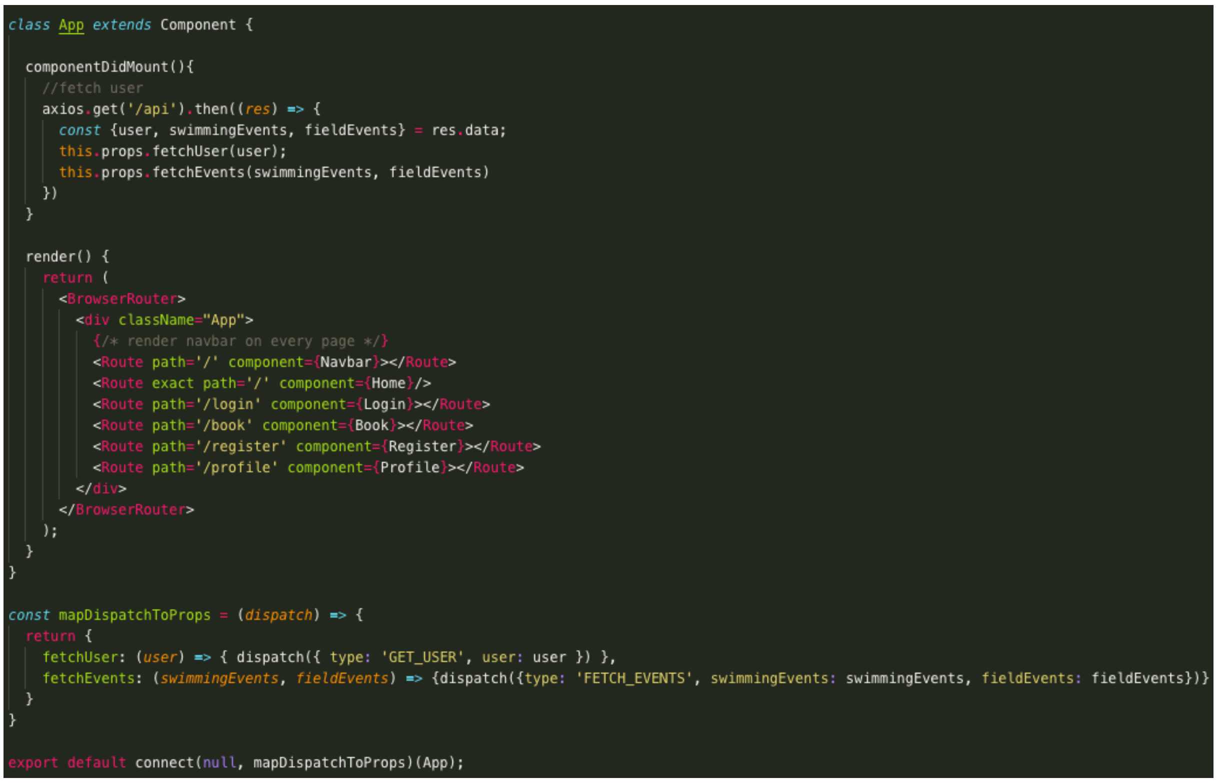


```
router.post('/', [
    // username must be an email
    check('email').isEmail().withMessage('Please provide a valid email'),
    check('name').isAlpha().withMessage('Name must contain only alphabetical characters'),
    check('surname').isAlpha().withMessage('Surname must contain only alphabetical characters'),
    check('city').isAlpha().withMessage('City must contain only alphabetical characters'),
    check('street').isAlpha().withMessage('Street must contain only alphabetical characters'),
    check('number').isAlphanumeric().withMessage('Name must contain only alphabetical and numerical characters')
], async (req, res) => {
        try {

            const errors = validationResult(req);
            if (!errors.isEmpty()) {
                return res.json({
                    error: errors.array()[0].msg
                })
            }else if(!(/.{8}/).test(req.body.password)){
                return res.json({
                    error: 'Password needs to be 8 characters or longer'
                })
            }

            const { password, email, name, surname, city, street, number } = req.body;
```

The code above is a part of the server route responsible for user registration. A request is sent to it from the client when a user submits the registration form. Most validation here is performed using the express-validator library. Here, it is used as middleware, which means that it runs before the callback function. The validation results are attached to the request object. If any validation errors occurred they can be accessed in the callback function using the *validationResult()* function (imported from express-validator). If there are any errors a JSON response is returned (which exits the callback and therefore stops later code from executing) with the message of the first validation error that occurred. Also, if no validation errors occurred the password is validated to be eight characters or longer using a regular expression. If it fails, the validation also a JSON response with an error message is returned. If all validation is successful, the rest of the code continues.

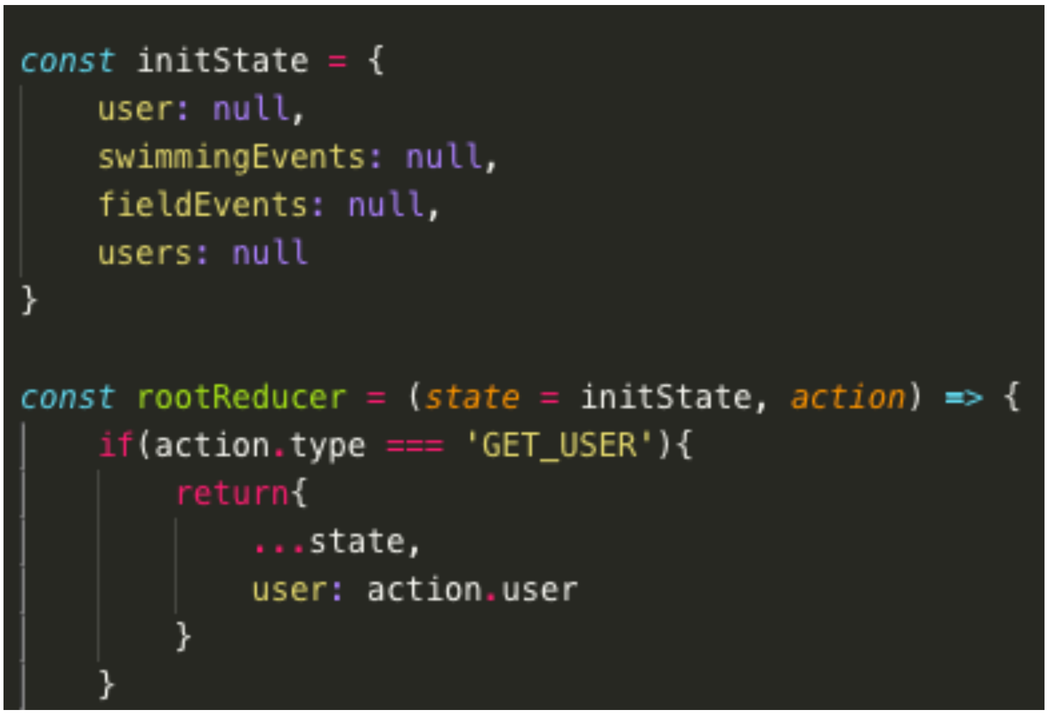
Redux

During the development of my front-end I realized that as the website started to consist of more and more components the data flow was harder to manage. After some research I decided to use Redux - a state container for React applications, which acts as a global store of data.



```
class App extends Component {

  componentDidMount(){
    //fetch user
    axios.get('/api').then((res) => {
      const {user, swimmingEvents, fieldEvents} = res.data;
      this.props.fetchUser(user);
      this.props.fetchEvents(swimmingEvents, fieldEvents)
    })
  }

  render() {
    return (
      <BrowserRouter>
        <div className="App">
          {/* render navbar on every page */}
          <Route path='/' component={Navbar}></Route>
          <Route exact path='/' component={Home}/>
          <Route path='/login' component={Login}></Route>
          <Route path='/book' component={Book}></Route>
          <Route path='/register' component={Register}></Route>
          <Route path='/profile' component={Profile}></Route>
        </div>
      </BrowserRouter>
    );
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    fetchUser: (user) => { dispatch({ type: 'GET_USER', user: user }) },
    fetchEvents: (swimmingEvents, fieldEvents) => {dispatch({type: 'FETCH_EVENTS', swimmingEvents: swimmingEvents, fieldEvents: fieldEvents})}
  }
}

export default connect(null, mapDispatchToProps)(App);
```

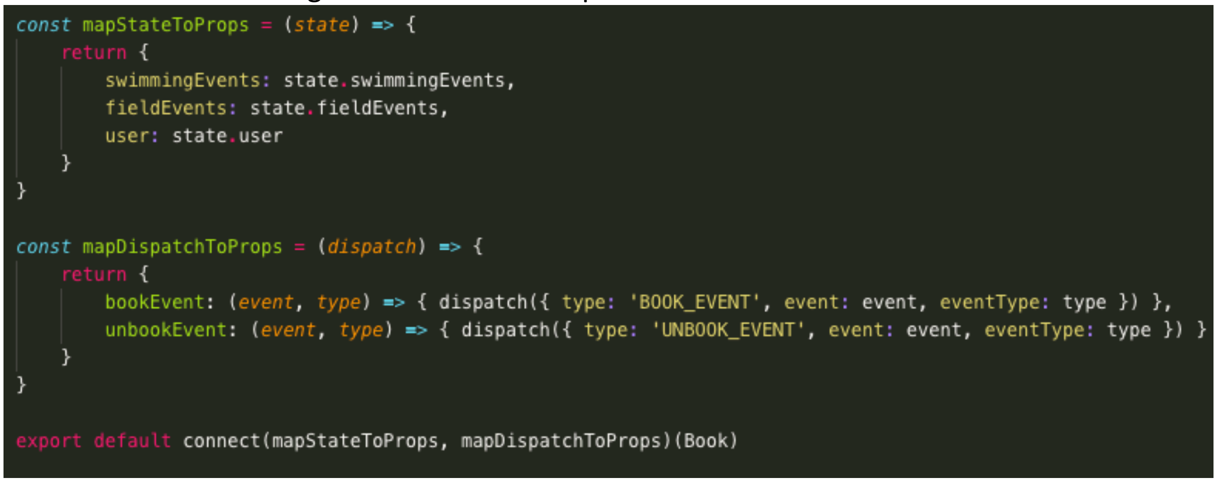
Both the logged in user and all the swimming and field events are fetched from the server when the main component (App.js) mounts. The *mapDispatchToProps()* function, that adds two functions (*fetchUser()* and *fetchEvents()*) to the component's props, is created and connected to the global store (Redux) using the *connect()* function imported from the react-redux library. After the user and events are fetched they are sent to the global store controller using the *fetchUser()* and *fetchEvents()* functions. Part of this global store controller is shown below:



```
const initState = {
    user: null,
    swimmingEvents: null,
    fieldEvents: null,
    users: null
}

const rootReducer = (state = initState, action) => {
    if(action.type === 'GET_USER'){
        return{
            ...state,
            user: action.user
        }
    }
```

It receives the actions dispatched (called) from other components. Each action has a certain payload and a type based on which the global state is updated.

To access the global state (store) from a different component a function *mapStateToProps()* must be created and together with that component connected to Redux:

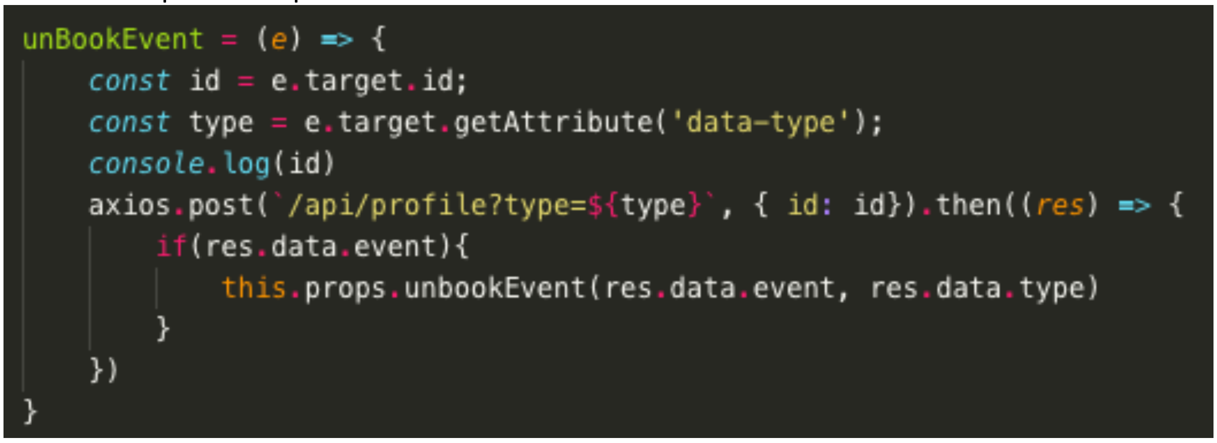


```
const mapStateToProps = (state) => {
    return {
        swimmingEvents: state.swimmingEvents,
        fieldEvents: state.fieldEvents,
        user: state.user
    }
}

const mapDispatchToProps = (dispatch) => {
    return {
        bookEvent: (event, type) => { dispatch({ type: 'BOOK_EVENT', event: event, eventType: type }) },
        unbookEvent: (event, type) => { dispatch({ type: 'UNBOOK_EVENT', event: event, eventType: type }) }
    }
}

export default connect(mapStateToProps, mapDispatchToProps)(Book)
```

The example above shows the *Book* component, responsible for booking events, which has both the state and actions which change it connected to it.

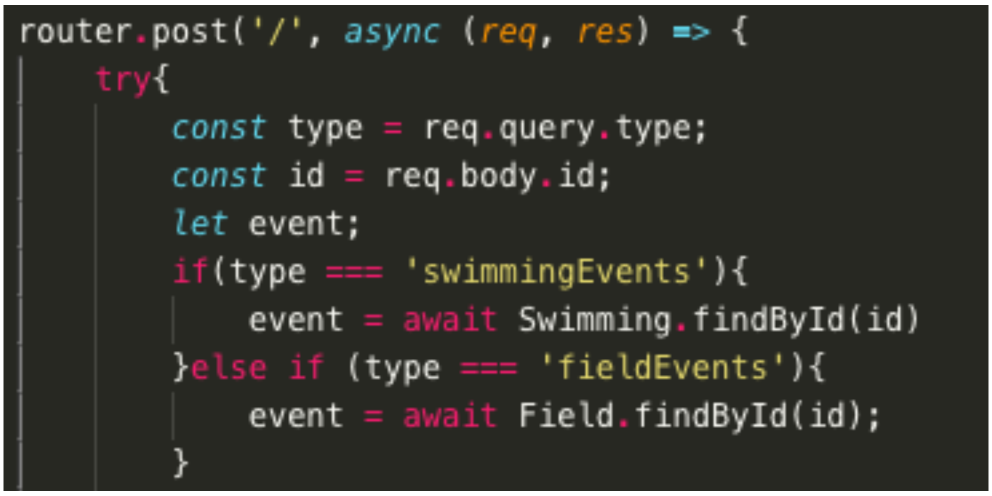
Query strings for reusability

The web app has two types of events: swimming and field. At the beginning I was developing separate handlers for each event type i.e. there was a separate route for unbooking a swimming event and a separate one for unbooking a field event. I soon realized that this resulted in a lot of repetition and I tried looking for an alternative. After some experimentation I decided to settle on using query strings to send the appropriate type to the API and perform operations based on that.



```
unBookEvent = (e) => {
    const id = e.target.id;
    const type = e.target.getAttribute('data-type');
    console.log(id)
    axios.post(`/api/profile?type=${type}`, { id: id}).then((res) => {
        if(res.data.event){
            this.props.unbookEvent(res.data.event, res.data.type)
        }
    })
}
```

This is the function that is called when the user clicks the unbook event button on his profile page. Each button has a *data-type* attribute that is either set to *swimmingEvents* or *fieldEvents* (same name as fields of events in user records in the database) depending on the event type. After extracting the *type* and *id* attributes from the clicked button a post request is sent to the profile route (which is responsible for unbooking events from the profile page). The id of the event being unbooked is sent in the body of the request while the type is included in the query string (although the type could also be sent in the body, some other HTTP request performed by the web app, such as PUT, can't have a body but still need to send the type which can only be achieved in a query string. Therefore, here the type is sent in a query string for consistency).



```
router.post('/', async (req, res) => {
    try{
        const type = req.query.type;
        const id = req.body.id;
        let event;
        if(type === 'swimmingEvents'){
            event = await Swimming.findById(id)
        }else if (type === 'fieldEvents'){
            event = await Field.findById(id);
        }
```

The code in the profile route extracts the type from the query string and based on it, it either fetches the event from the Swimming or Field collection.

Words: 1049

CRITERION E

1. Clients and admins can login with existing accounts
    a. MET
    b. Users can login on the /login page, if the credentials are correct the user is logged in, otherwise an error message is displayed.
2. Clients can create accounts themselves
    a. MET
    b. Users can register on the /register page, if no other user with the same username exist, they will by successfully registered.
3. Admins (company workers) can add and edit events
    a. MET
    b. Users with admin privileges can add or edit (if the event already exists) an event by clicking on a specific day of the calendar and filling out the necessary fields.
4. Admins can delete events which causes the cancelation of reservations for all users who booked that specific event
    a. MET
    b. Admins can delete an event by clicking on its bin icon.
5. Admin's can view users' information
    a. MET
    b. A table with all the users and their corresponding information is displayed on the admin page.
6. Users can book/unbook events
    a. MET
    b. On the /book page users can book (or unbook if they already booked it) events by simply clicking on the day with the desired event.
7. If the number of places for an event has been exceeded the website doesn't allow any additional users to book a place for this event
    a. MET
    b. Calendar tiles with events which have no more places left, are disabled and therefore don't allow to be booked.
8. Website doesn't refresh
    a. MET
    b. All the pages of the web app cab be accessed (using the navbar) without it refreshing
9. All user inputs must be validated
    a. MET
    b. Each endpoint of the API validates user input. If the input doesn't pass the validation an error message is sent and the code stops executing.
10. Clients can book both a swimming and field event on the same day
    a. MET
    b. On the /book page users can book both a swimming and field event for the same day.

Improvements

- Confirmation question before deleting an event. The client mentioned that, during testing, workers accidently deleted a few events while trying to edit them. The confirmation dialog would make sure that the user double checks before deleting.
- Mobile responsibility: Although some parts of the app are responsive, most aren't. The client asked to make the whole website responsive, to provide a better experience for potential users who would visit it using a mobile device.

Extensibility

The client was very satisfied with the final result, as it met all of his requirements. After some additional discussion we concluded that the following future extensions could be added:

- Live updating the number of places for each event – as long as the user stays on the /book page the number of places isn't updated, which means that sometimes it might be misleading (an event isn't disabled event though all the places for are already booked). WebSockets could be used to provide live changes.
- Allowing users to edit their profile – currently users are unable to change any information. This means that if they, for example, move houses, they have to create a new account with the updated information. This results in a duplicate accounts.

Words: 501

Interview with CEO of company Ewenty

Me: Good morning, as agreed, I will build a website, which will allow you company employees to add events, which later can be booked online by clients. Could you please provide me with a bit more details regarding the overall design of the web app?

CEO: Yes, of course. First, it would be very nice if clients were able to create an account rather than provide their personal details every time, they book an event. Is programming this possible?

Me: Absolutely. What information exactly do you want the clients to provide during registration?

CEO: First name, surname, email address, city, street and street number.

Me: Ok, thank you. Could you also provide a bit more detail regarding the creation of events by company workers?

CEO: As you might already know, our company organizes two types of events: a swimming competition and a field competition. The company workers should be able to create each event on a desired date and specify the number of places available for it. Additionally, for swimming events the swimming style as well as whether the jacuzzi is available should be specified, while for field events the activity type. Also, the company workers should be able to edit/delete the events once they are created and view the name and surname of the users who booked them.

Me: Fine, I am thinking of placing two calendars on the admin dashboard – one for swimming events and the other for field events. Once the employee clicks on a specific day a modal would be displayed, where the employee could add/edit information about the event on that specific day. He could also delete it by clicking a dedicated button on the model. Does that sound fine?

CEO: Yes, perfect. Could you also apply the same design to the page where clients can book events? It would be nice to also have two calendars on that page with information about an event on each day. Clients would book/unbook events by clicking on the day with an event available.

Me: No problem. The days with booked events will have a green background color so that clients can easily distinguish between them and the one which they haven't booked.

CEO: Also, don't worry a lot about the home page. Since we already have an information website, the homepage doesn't need to be very detailed. A button encouraging users to book events would be enough.

Me: OK

CEO: Could you also include a client's profile page, where each client would have a list of all the events he booked. This will make it easier for clients to manage their events.

Me: No problem, would you like an unbook button next to each event on the client's page?

CEO: If it's not a big problem, yes. The company would also highly appreciate if the website wouldn't reload when the user changes pages, as this will create an impression in clients that the website is more modern. Is it possible to build something like this?

Me: Yes, that won't be a problem.

Final Interview with CEO of company Ewenty

Me: Good morning, how did you like the website I presented to you?

CEO: A lot, thank you! The thing that could be improved in the future is making the website mobile friendly since right now some of the elements don't look that nice when viewed on a smartphone.

Me: I see. Is there anything else?

CEO: I think this is a minor thing but two of the admins accidentally deleted events. Displaying a confirmation dialog before deleting an event would be nice.

Me: Ok. How do you think the website could be further extended in the future?

CEO: I think that allowing users to edit their profile information such as the address would be suitable. This would allow users to change their account information when they are for example, moving homes, instead of creating a second account.

Me: I personally also think that some sort of live update should be added to the booking page so that users don't have to refresh the page to see live changes.

CEO: That's an excellent idea, thank you!

Me: Thank you!