

Nexios Frontend Bootcamp: Live Messaging App

This is the playbook for the 2022 Nexios IT Frontend bootcamp. The goal is to build a frontend in React for an existing API, building a live chat application built for desktop.

Getting up and running

First thing to do is make sure we have all the tools installed that we need to develop a React application. After this we will setup a boilerplate application with all the packages, we need installed. We are assuming you are using a Windows installation. If you use MacOS or Linux, some steps might be different.

1) Install an IDE or Text Editor

Of course, we can't do anything without a text editor. We recommend Visual Studio Code for lightweight React development.

<https://code.visualstudio.com/>

2) Install Node.js

Everything we do is dependent on an installation of Node.js. Download and install the latest version from the official website. Installing Node.js will also install the npm and npx command, which we will use later.

<https://nodejs.org/en/>

3) Use CRA to setup a React boilerplate app.

- Make a directory for your repos if you haven't already got one. We recommend something short like C:/Repos
- Open a Cmd or Powershell **in your repo directory (C:/Repos)**.
- Enter the following command and let the installation run. This command will fetch a boilerplate React application with a typescript template pre-installed.

`"npx create-react-app live-chat-frontend --template typescript"`

This will make a new folder called "live-chat-frontend" and install the boilerplate application inside.

After the installation is complete you can enter it with `"cd live-chat-frontend"`, open the directory in VSC with `"code ."` and finally start the development server with `"npm start"`.

If all went well, you should see a new window appear in your default browser with your newly created React app!

Hint: You can also use the built in Powershell offered by VSC if you don't like working with detached windows.

4) Link our new project directory to the Github Repo.

Now that the basic installation has been completed, we have something to push to our Github repo as a base commit.

Follow the steps here to add your given Github repo as an origin and push everything. <https://www.techieclass.com/convert-a-folder-to-a-git-repository/>

Create-React-App already comes with a .gitignore file so don't worry about pushing files you're not supposed to.

Alternatively, you can use the built in Git tools supplied by VSC to do this.

5) Make our first feature branch

The basic installation is not enough however, we will be using a couple important packages from npm to make our app function.

Since our repo has been initialized, we will work with feature branches from now on.

Create a new local branch, for example “feature/install-packages”. The feature/ in the name will organize this branch under a folder called feature.

Checkout the branch and open the Cmd or Powershell window again in the root directory.

- Install **Material UI & Material Icons** (Web components, styles & icons)
 - o “[npm i @mui/material @emotion/react @emotion/styled](#)”
 - o “[npm i @mui/icons-material](#)”
 - o Add Roboto font to the application. Go into public/index.html and enter the following in the head of the html file.

```
<link
  rel="stylesheet"
  href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=
swap"
/>
```

- Install **React-Router (V6)** (For routing)
 - o “[npm i react-router-dom](#)”
- Install **Axios** (For API calls)
 - o “[npm i axios](#)”
- Install **Classnames** (Useful for working with classnames and css modules)
 - o “[npm i classnames](#)”

When all this is done your “dependencies” array in package.json should look a little like this:

```
"dependencies": {
  "@emotion/react": "^11.10.4",
  "@emotion/styled": "^11.10.4",
  "@mui/icons-material": "^5.10.9",
  "@mui/material": "^5.10.10",
  "@testing-library/jest-dom": "^5.16.5",
  "@testing-library/react": "^13.4.0",
  "@testing-library/user-event": "^13.5.0",
  "@types/jest": "^27.5.2",
  "@types/node": "^16.18.0",
  "@types/react": "^18.0.22",
  "@types/react-dom": "^18.0.7",
  "axios": "^1.1.3",
  "classnames": "^2.3.2",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "react-router-dom": "^6.4.2",
  "react-scripts": "5.0.1",
  "typescript": "^4.8.4",
  "web-vitals": "^2.1.4"
},
```

When this is done, commit all the changes and publish your branch. Go to your Github repo page and make a new PR to merge this new branch into main.

Hint: This is the general workflow you will follow for any new feature or bugfixes after this. Make feature branch, make changes, commit & publish, PR to merge into main. Features can go under features/ folder and bug fixes can go under bug/.

That’s it, you are now fully ready to start developing! In the next part we will outline the minimal viable product, as well as a list of extra features you can implement if you have extra time.

Project Description

In this section we will outline what we expect as a minimal solution, as well as supply a list of extra features you could implement if you have time.

The API has been deployed at the following URL:

<https://api.chat.bootcamp.nexiosdevops.be>

To connect to the websocket you can use the following string:

<wss://api.chat.bootcamp.nexiosdevops.be>

You can see the API endpoints and documentation by browsing to the url with `"/docs"` attached at the end in your browser. A Swagger page should appear that details the available endpoints and allows you to test them if you'd like to.

We are building a live chat application. That means the following things **need** to be in place:

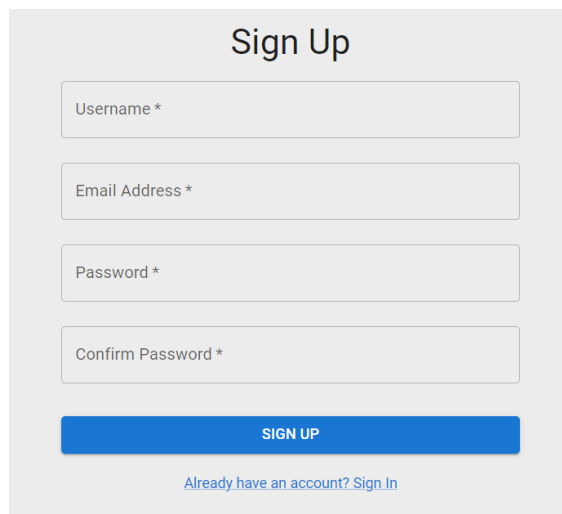
- Register a new user, with an email address and a display name (username)
- Login/Logout as a registered user
- View a list of chatrooms the user is a part of
- Create a new chatroom with any amount of other registered users
- View the messages for a given chatroom
- Send a new message to a given chatroom
- Receive & process live updates from a Websocket containing:
 - o New users
 - o New chatrooms
 - o New messages

Styling and CSS is not the focus of this workshop, but you are free to get creative if you want to. Using Material UI allows us to build a usable frontend without having to write much CSS.

Requirements (minimum):

- Register Page

- Fields for Username, Email, Password, Password Confirm
- Basic formchecking, all fields are required, passwords match
- Possible API errors displayed:
 - Username already in use
 - Email already in use
- Eg:



Sign Up

Username *

Email Address *

Password *

Confirm Password *

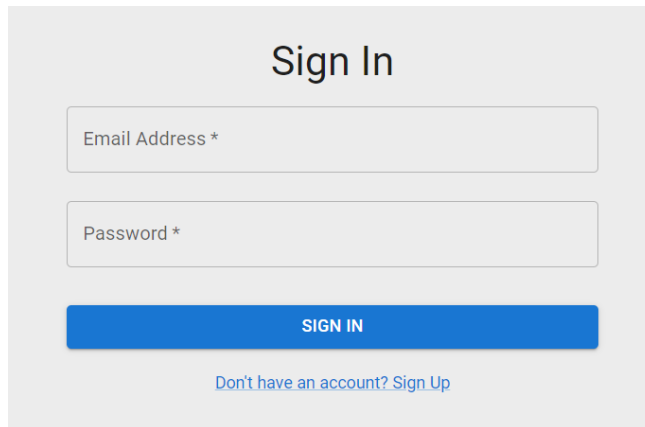
SIGN UP

[Already have an account? Sign In](#)

- Login Page

- Fields for Email, Password
- Basic formchecking, all fields are required
- Possible API errors displayed:
 - Unauthorized

- Eg:



A sign-in form with a light gray background. At the top, the text "Sign In" is centered in a bold, dark gray font. Below it are two input fields: "Email Address *" and "Password *", both with light gray borders and placeholder text. A blue button with the text "SIGN IN" in white capital letters is positioned below the password field. At the bottom, there is a link that says "Don't have an account? Sign Up" in a small, blue font.

- **Main Page**

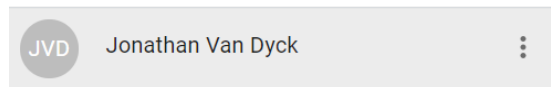
- Sidebar component

- Sidebar Header with user display & controls

- Contains a context menu that has actions:

- Create a new chatroom
- Log out the user

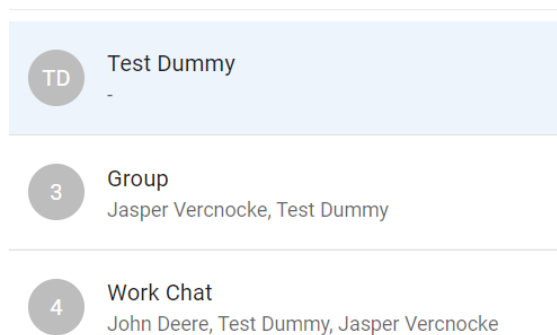
- Eg:



- Chat List with a list of current chatrooms

- Acts as a navigation menu, highlights selected chatroom

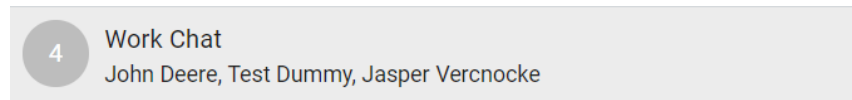
- Eg:



- Chat View component

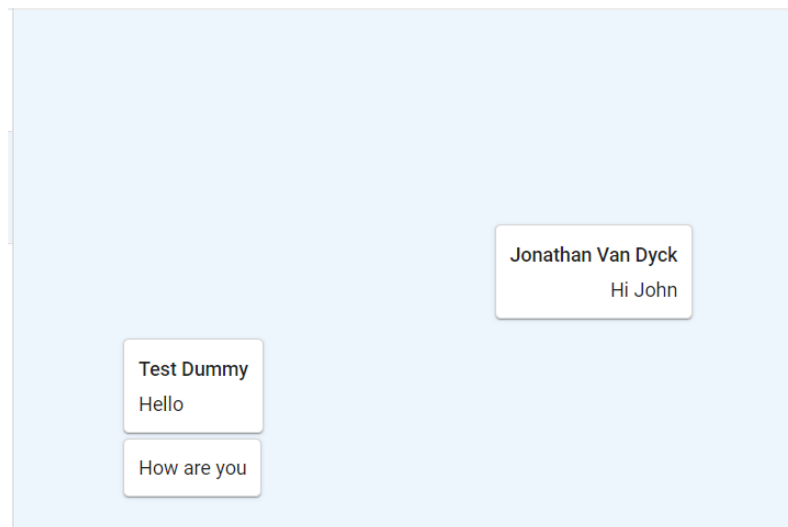
- Chat View Header

- Contains information about the currently selected chatroom
- Eg:



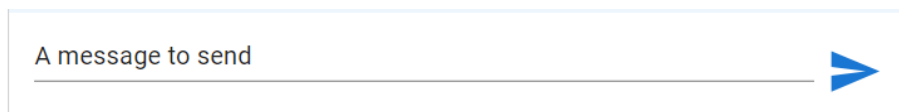
▪ Chat History

- Contains a scrolling history of messages for the currently selected chat
- Behaves the same way as for example Whatsapp Web. Messages appear from the bottom to the top
- Eg:



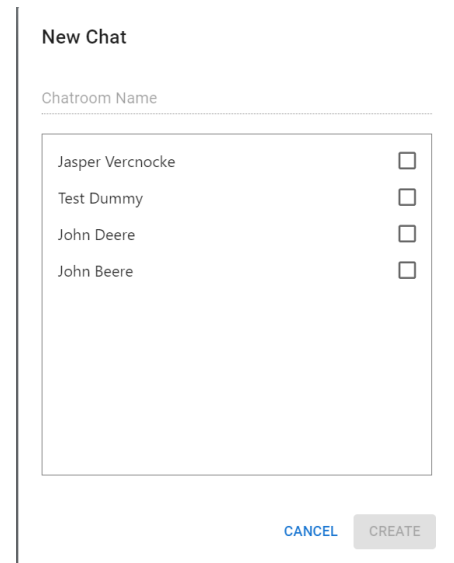
▪ Chat Input

- Contains an input and a submit button
- Eg:



- Add chat popup

- A popup modal that allows the user to create a new chatroom
- Contains a field for the chat name and a list of current users to add to it
- Eg:

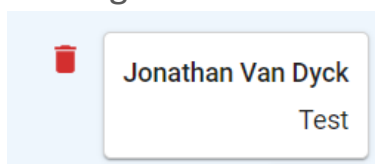


The image shows a 'New Chat' modal form. At the top, it says 'New Chat'. Below that is a text input field labeled 'Chatroom Name'. Underneath the input field is a list of four users: 'Jasper Vercnocke', 'Test Dummy', 'John Deere', and 'John Beere'. Each user name is followed by a small square checkbox. At the bottom right of the modal, there are two buttons: a blue 'CANCEL' button and a grey 'CREATE' button.

Additional Features

Here is a list of possible features you could implement if you are done with the minimal viable product.

- Make a distinction between Single and Group chats.
 - Displayed differently in the chat list (see examples above)
 - When creating a new chat, if it is a single chat, the name is no longer needed since it's not visible. However, the backend should still receive something as the name, or it will throw an error
- Implement a search field for the chat list
 - When searching, group chats are filtered by name and single chats are filtered by chat partner.
- Implement delete message. Allow the user to delete one of their own messages. On hover an icon button could appear which will delete a message when clicked.



There is already a websocket message in place that will trigger when a certain message has been deleted.

- Implement delete chat. Allow the user to delete a chat they are a part of. Warning, the delete chat endpoint supplied will delete a chat from the database, not just remove it from the users list.

Same as with messages, there is a websocket in place that will trigger when a chat has been removed.

- Implement unread messages. If a chat contains new messages (hint: from Websocket) it will show a notification bubble letting the user know, until they open the chat, at which point it clears



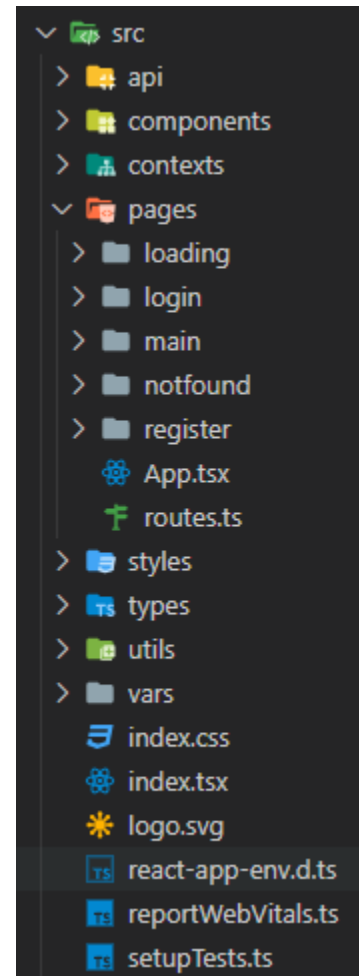
- Make it so the user doesn't have to log back in on page refresh. This is accomplished by saving the users authentication token to the browsers SessionStorage.

Information & Guidelines

If you are a bit unsure how to proceed, here is a list of hints, techniques and suggestions you could follow to get going.

- Suggested file structure

- **/api** contains files with functions that call API endpoints and return the result.
- **/components** contains components that are generic and reusable. These components do not contain application specific logic.
- **/contexts** contains our React Contexts
- **/pages** contains our different pages, in separate folders with each their own **/components** directory for page specific components. As well as our main App component and a routes file which contains our different routes as string variables.
- **/styles** contains any global css files
- **/types** contains any global Typescript interfaces
- **/utils** contains utility functions
- **/vars** contains global constants



- Basic Routing

First thing you should do is setup a basic routing system using React Router. You only have 3 routes to worry about: root (main page), login and register. Initially, since there is no user authentication yet, all routes are accessible at any point by anyone. We will of course later automatically reroute users if they are on a page they shouldn't be on.

Hint: For our app we will be needing a BrowserRouter.

- First Context and User authentication

To handle our user login and registration we will be making our first Context. Because this concept might be a bit hard to grasp for beginners, we will provide an example of what this context might look like.

```
export interface IUserContext {
  loading: boolean;
  user?: IUser;
  jwt?: string;
  signIn: (email: string, password: string) => Promise<LoginResult>;
  signOut: () => void;
}

const UserContext = createContext<IUserContext | null>(null);

export const UserContextProvider = ({ children }: IProviderProps) => {
  const [loading, setLoading] = useState<boolean>(false);
  const [user, setUser] = useState<IUser | undefined>();
  const [jwt, setJwt] = useState<string>();

  const signIn = async (email: string, password: string): Promise<LoginResult> => { ...
  };

  const signOut = async () => { ...
  };

  return <UserContext.Provider value={{ loading, signIn, signOut, user, jwt }}>{children}</UserContext.Provider>;
};

// Create a hook to access the user context
export const useUserContext = (): IUserContext => {
  const context = useContext<IUserContext | null>(UserContext);

  if (!context) {
    throw new Error("User context must be used within a Provider.");
  }
  return context;
};
```

Then consume the context in any React Component or other Context:

```
const { jwt, user } = useUserContext();
```

The example contains the definition of the user context, as well as the hook which can be used to access it from within the children of the provider.

```
root.render(  
  <React.StrictMode>  
    <ThemeProvider theme={theme}>  
      <UserContextProvider>  
        <AppContextProvider>  
          <App />  
        </AppContextProvider>  
      </UserContextProvider>  
    </ThemeProvider>  
  </React.StrictMode>  
)
```

As you can see from the image on the left, this is how you use the Provider. We place the Provider around the entire application at the top level so everything can access it.

Here you can also see the second Context that will be created, the AppContext. This context works in the same way but manages a global data and

UI state for the entire application. The AppContext is located inside the UserContext because it uses the data it supplies like the logged in user and its authentication token. How this AppContext is implemented is left to you.

Once the user authentication is working, it can be used in the routing to automatically reroute a user if they are somewhere they shouldn't be.

This can be achieved in multiple ways, some better than others.

Hint: You can wrap a route inside another route that simply renders its child routes, on a certain condition.

- API & Typescript interfaces

It's important our Typescript interfaces are well defined so they can help us define data structure, both for data coming from the API, and internal data interfaces.

- Separate the backend interfaces from the internal interfaces and create mappers to change between the two.

For example if the backend returns something such as `allowed_users` but internally you want to use `allowedUsers` to follow Javascript naming conventions.

You can define the API interface as having `allowed_users` and then create a mapper function that turns that into your internal interface that has `allowedUsers`. This way your internal interfaces are decoupled from those that the API responds with.

- You can use interface inheritance to define an API result.

```
export interface ApiResultBase {
  isSuccess: boolean;
  error?: string;
}

// Authentication api
export interface LoginResult extends ApiResultBase {
  accessToken?: string;
}
```

- Example of an authenticated API call that maps the API interface to the internal one and returns it as a successful API result.

```
const response = await axios.get(url, { headers: getDefaultHeaders(jwt) });

if (response.data && Array.isArray(response.data)) {
  return {
    isSuccess: true,
    users: response.data.map((user: IApiUser) => mapApiUser(user)),
  };
}
```

- Material UI

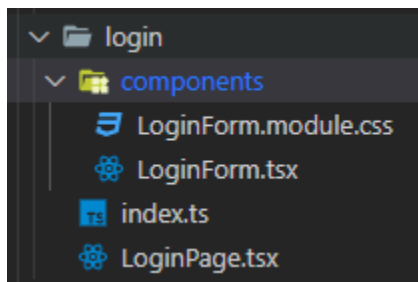
It is advisable to make as much use of the components supplied by Material UI as possible. This way you write as little CSS as possible to still end up with a satisfying and functioning application. We recommend you look through the documentation thoroughly, so you know what is available to you.

- CSS & CSS Modules

Some CSS will be needed, however.

- For global styles that apply to the entire application, you can use regular .css files and import them in App.tsx
- For component specific styles, it is recommended to use CSS modules.

First make a x.module.css file next to the component it is for.



There you can define classes and styles the same way you would in a normal CSS file.

```
.loginFormContainer {  
  max-width: 450px;  
}
```

Then, import the module in your React Component.

```
import styles from "../LoginForm.module.css";
```

And finally apply the classname to a component.

```
<Box className={styles.loginFormContainer}>
```

This ensures that at runtime, there will never be issues with conflicting classes or styles, because that CSS file is bound to the component, as a module.

- You can make CSS variables to easily reuse certain styles, such as colors or margins.

- Websockets

Make sure the connection to the websocket happens cleanly. Meaning that the connection is only made once the user has logged in, and the connection is closed when the user logs back out.

Under the section below there is a link to a stackoverflow page that has a decent example how to get started.

However, contrary to the example given, it is advised to configure the websocket inside of the AppContext. This means it is easier to only have the connection be established after login and the access to the data to make changes is right there.

This part is probably the most challenging of the project, ask for help if you get stuck.

Online Documentation Sources

Below is a list of locations online where you can find the documentation for everything that is used.

- Visual Studio Code
<https://code.visualstudio.com/>
- Node.js
<https://nodejs.org/en/>
- Getting Started With React:
[https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side JavaScript frameworks/React getting started](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started)
- React:
<https://reactjs.org/docs/getting-started.html>
- MaterialUI:
<https://mui.com/material-ui/getting-started/overview/>
- React Router
<https://www.freecodecamp.org/news/how-to-use-react-router-version-6/>
- Axios
https://axios-http.com/docs/api_intro
- Websockets for React
<https://stackoverflow.com/questions/58432076/websockets-with-functional-components>