Documentation

Partner Network

Linus Holtkamp
linus.holtkamp@rwth-aachen.de

Moritz Langenhan
moritz.langenhan@rwth-aachen.de

Ecurie-Aix
RWTH Aachen
Version 1

Contents

1 Introduction

This document serves the purpose of helping future developers handling this piece of software. We opted to making this network with the (currently) most renowned JavaScript framework 'React'. This provides us with the most accessible packages for current and future support addition options and documentation for the sake of accessibility. Furthermore, we used a package manager called 'Vite', which enabled us a quick start and overall structure managing, with the cost of now being forced to use TypeScript, a JavaScript-like programming language, optimized for client-sided website developing. While this in theory is the most optimal language for us to use, it required some new skills to be developed for us, so please do not threat us if some code is 'gibberish' or not perfectly optimized :).

1.1 Initial Setup Guide

Because we are not using a framework like WordPress, we are forced to host our website locally. To get the project running you will first need to follow these steps:

Node

Install the latest version of Node.js from:

```
https://nodejs.org/en/download
```

Afterwards, if not yet working properly, restart your PC to ensure that the installation is finished.

npm Installs

Type these commands into a console of your choice, line by line:

```
npm install -D tailwindcss-animate
npm install react-router-dom
npm install -D tailwindcss postcss autoprefixer
npm install -D @types/node
npm install appwrite@13.0.0
npm i @tanstack/react-query
npm install react-dropzone
npm install react-intersection-observer
npm install gapi-script
```

It is important that appwrite is getting installed at the specified version, otherwise certain query functions won't execute correctly.

Hosting

To host the website locally, run

```
npm run dev
```

Should you stumble into some errors, here is the solution that fixed most of our problems:

Important Bugfixes

Try running these actions and commands to reinstall node after navigating to the location where your project is stored (for shell commands, lookup under '2. Git'):

```
delete node_modules from the folder structure
npm install
```

This should reinstall node and fix most of your issues. If your problem still persists, try googling it or contact us.

1.2 Database

As a back-end, in this project we used a relatively new tool from AppWrite, which is still in beta and to our luck it therefor provides its services free of charge.
To access the dashboard, visit this webpage, using the following credentials to login:

AppWrite

`https://cloud.appwrite.io/login`

Credentials

moritz.langenhan@icloud.com
ecurie_aix_internship

2 Git

How to install Git

Install Git on your PC.

Go to `https://git-scm.com/download`, select your operationg system and the download should start automatically.

The first thing you should do when you installed Git is to set your user name and email address. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

How to initialize Git

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Now that your git is all set up, we can start with cloning the repository to our machine.

Open your console of choice and navigate to the desired location using the following Shell commands:

Shell Commands

```
$ cd <sub_directory>                          (change directory to sub_directory)
$ cd ..                                             (go back to parent_directory)
$ ls                                        (lists content of current directory)
```

The chosen location is where the project is going to be saved. When you are ready, use this command:

Clone the Repository

```
$ git clone
https://git.rwth-aachen.de/ecurieaix/web-development/partner-network.git
```

The network folder structure should now be saved under the chosen location.

3 File Structure

This project follows a pretty default way of file structuring for webprojects. First we have our typical folders. Which are **node_modules**, **public** and **src**. Then there are also the standalone files in the root of our directory.

3.1 node_modules

node_modules is generated by the previous installation of npm and should not be changed, removed or extended due to its crucial role in the code of this site.

Just leave it alone and you'll be fine :).

3.2 public

The **public** directory contains all the necessary (vector)-images and videos that we embed on our website.
icons contains all the vector images (.svg) used to display each icon on the website. If you want to add any icons, e.g. for buttons or filters, put them here.
images, as you can imagine, holds all the images used on the website, e.g. backgrounds or logos. Note that images like profile pictures and uploaded images are stored in the back-end and thus can not be found here.
videos, I'm sick of spelling it out anymore, contains videos used on the website. Including the background video in the sign-up section.

3.3 root files

These **root files** contain the basic setup of our project.

3.3.1 .env.local

This file contains the id's of our back-end address points. These can be found in the dashboard of the back-end. For further information, read under '1.2 Database'.

3.3.2 .eslintrc.cjs

This file works as a tool for identifying patterns within our code. For further information, please read under this link:

`https://en.wikipedia.org/wiki/ESLint`

3.3.3 .gitignore

This file specifies the files that are being ignored when you commit your changes to git.

3.3.4 components.json

This file is a configuration file for our project using the ShadCN UI library for design components and Tailwind CSS for overall styling.

3.3.5 index.html

This file is the reason why the icon and text is displayed at the top on the tab-section of your browser.

3.3.6 package-lock.json

This file works as sort of an inventory of node and package versions we are using.

3.3.7 package.json

This file is the same as above, but private.

3.3.8 postcss.config.js

This file works as a plugin config, which we are not using, it's just there, duh.

3.3.9 tailwind.config.js

This is quiet the important file. Here we specified our styling themes, our colors we are using and our fonts. For further information, visit the tailwind docs:

`https://tailwindcss.com/docs/configuration`

3.3.10 tsconfig.json

This file acts as our TypeScript config.

3.3.11 tsconfig.node.json

This file contains further compiler options for TypeScript.

3.3.12 vite.config.ts

In here the only important thing is this:

'@' Resolving

```
resolve: {
    alias: {
      "@": path.resolve(__dirname, "./src"),
    },
},
```

This resolves the '@' to "./src". which makes it easier for us to refer to components in our file structure.

3.4 src

The **src** folder is where the magic happens. All the logic is stored in here.

3.4.1 App.tsx

This file is the entry point of our project, it always gets looked at first by the compiler. It set's up our routing by using the `react-router-dom` package.

Public Routes

```
    <Route element={<AuthLayout />}>
      <Route path="/sign-in" element={<SigninForm />} />
      <Route path="/sign-up" element={<SignupForm />} />
    </Route>
```

These are our public routes, which are the pages that are accessible without authentication. They are getting imported at the top of the file.

The Route component takes two arguments, the first one being the path, which is set to the relative path to the component we are refering to. The second one is the element being the component itself.

Private Routes

```
<Route element={<RootLayout />}>
    <Route index element={<Home />} />
    <Route path="/explore" element={<Explore />} />
    <Route path="/saved" element={<Saved />} />
    <Route path="/all-users" element={<AllUsers />} />
    <Route path="/create-post" element={<CreatePost />} />
    <Route path="/update-post/:id" element={<EditPost />} />
    <Route path="/posts/:id" element={<PostDetails />} />
    <Route path="/profile/:id/*" element={<Profile />} />
    <Route path="/update-profile/:id" element={<UpdateProfile />} />
  </Route>
```

These are our private routes. They work the same as the public ones, but you need to be logged in to be able to access them. This layout makes it so that these pages are layered ontop of each other, making it easier to access them.

Toaster

```
<Toaster />
```

This is an additional package we added, that displays error messages. For further infromation, look at their docs:

`https://react-hot-toast.com/docs/toaster`

3.4.2 globals.css

This is the main styling file. Here, we defined our stylings to make them applyable by just referencing the name of it. E.g.:

Private Routes

```
.post-card_img {
  @apply h-64 xs:h-[400px] lg:h-[450px] w-full rounded-[24px]
  object-cover mb-5;
}
```

With this snippet, we now may reference it with 'post-card_img' in the `className` of some component to apply the styling, which is then applied by the `apply` keyword from the tailwind api. For further styling infos, please visit the tailwind docs:

`https://tailwindcss.com/docs/`

The base layer ...

Tailwind Styling

```
@layer base {
  @font-face {
    font-family: 'Univers LT Std';
    src: url('./fonts/UniversFontpack/UniversLTStd-LightCn.eot?#iefix')
    format('embedded-opentype'),
         url('./fonts/UniversFontpack/UniversLTStd-LightCn.woff2')
         format('woff2'),
         url('./fonts/UniversFontpack/UniversLTStd-LightCn.woff')
         format('woff'),
         url('./fonts/UniversFontpack/UniversLTStd-LightCn.ttf')
         format('truetype');
    font-weight: normal;
    font-style: normal;
  }

  @font-face {
    font-family: 'Univers_LT_Std_57';
    src: url('./fonts/UniversFontpack/UniversLTStd-Cn.eot')
    format('embedded-opentype'),
         url('./fonts/UniversFontpack/UniversLTStd-Cn.woff2')
         format('woff2'),
         url('./fonts/UniversFontpack/UniversLTStd-Cn.woff')
         format('woff'),
         url('./fonts/UniversFontpack/UniversLTStd-Cn.ttf')
         format('truetype');
    font-weight: normal;
    font-style: normal;
  }

  body {
    font-family: 'Univers LT Std', system-ui, sans-serif;
    background-color: #000000;
    color: #ffffff;
  }
}
```

...defines the base styling. This means that all elements and components get this styling applied even if there is none specified for that component. Here we applied 2 fonts to make it easier to switch in between them later. While UniversLTStd 47 is used as the default font (as defined by layer base), UniversLTStd 57 can easily be used due to its additional definition in here.

3.4.3 main.tsx

This is the second entry point of our application.

Create DOM

```
const root = ReactDOM.createRoot(document.getElementById("root")!);
```

This creates a root element in the DOM, where the React application will be rendered. The

document.getElementById(”root”)! targets an HTML element with the id root.

Create DOM

```
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <QueryProvider>
        <AuthProvider>
          <App />
        </AuthProvider>
      </QueryProvider>
    </BrowserRouter>
  </React.StrictMode>
);
```

(i) React Strict Mode
Wraps the application to help identifying problems during development.

(ii) Browser Router
Wraps the application to to enable routing capabilities throughout the app. It allows navigation and routing using URL paths.

(iii) Query Provider
Wraps the application to provide context for data fetching and caching using react-query. This enables efficient data management and synchronization with server data.

(iv) Auth Provider
Wraps the application to manage authentication state and provide authentication-related functionality across the app.

(v) App
Renders the App component

3.4.4 vite-env.d.ts

This references the vite environment. Don’t ask why, it’s just there.

3.4.5 _auth

This folder contains all the necessary forms and logic to the authentication and login or signup process.

AuthLayout.tsx

Our login and signup pages are rendered above our background video. This is handled by this file.

isAuthenticated

```
const { isAuthenticated } = useUserContext();
```

This uses the useUserContext function to determine whether the user has a cookie in his local cache, which we get from our back-end that provides this function.

Login/ SignUp Logic

```
return (
    <>
      {isAuthenticated ? (
        // If user is authenticated, redirect them to the home page
        <Navigate to="/" />
      ) : (
        <>
          {/* Full-screen background video */}
          <video
            src={VideoBg}
            autoPlay
            loop
            muted
            className="fixed top-0 left-0 w-full h-full object-cover
            -z-10"
          />

          {/* Semi-transparent overlay */}
          <div className="fixed top-0 left-0 w-full h-full bg-black
          bg-opacity-60 -z-10"></div>

          {/* Container for child components */}
          <section className="flex flex-1 justify-center items-center
          flex-col py-10 relative z-10">
              <Outlet />
          </section>
        </>
      )}
    </>
  );
```

This shows the background video and conditionally renders -whether the user is already authenticated or not- the child components referenced by the Outlet component. The Outlet component is a placeholder used by react-router-dom to render child routes. It allows you to define a layout that wraps around nested routes, making it possible to create a consistent layout structure while changing the content based on the route. In our case, these are the signup and login forms, as already specified in our App.tsx file.

3.4.1 forms

Here lie our two forms for the login and signup screen.

SigninForm.tsx

useNavigate

```
  const navigate = useNavigate();
```

This function from react-router-dom returns a function that is being stored in the navigate variable. It is then able to navigate our application, taking one argument being the relative path to a page and

navigating there.

isLoading

```
const { checkAuthUser, isLoading: isUserLoading } = useUserContext();
```

This function provides the context from earlier and also proposes an isLoading variable which is true while the data is being fetched from the back-end.

Our login and signup pages both use a shadcn form element, where one single input line is looking like this:

Login/ SignUp Logic

```
const form = useForm<z.infer<typeof SigninValidation>>({
resolver: zodResolver(SigninValidation),
defaultValues: {
  email: "",
  password: "",
},
});

<FormField
    control={form.control}
    name="email"
    render={({ field }) => (
      <FormItem>
        <FormLabel className="shad-form_label">E-Mail</FormLabel>
        <FormControl>
          <Input type="text" className="shad-input" {...field} />
        </FormControl>
        <FormMessage />
      </FormItem>
    )}
  />
```

This specific one refers to the email input from the user.

Once all the information is being submitted via the press of the lower button, this is being called:

onSubmit

```
onSubmit={form.handleSubmit(handleSignin)}
```

Here, we call the handelSubmit function:

handleSignin

```
const handleSignin = async (user: z.infer<typeof SigninValidation>)
=> {
    const session = await signInAccount(user);

    if (!session) {
      toast({ title: "Login failed. Please try again." });

      return;
    }

    const isLoggedIn = await checkAuthUser();

    if (isLoggedIn) {
      form.reset();

      navigate("/");
    } else {
      toast({ title: "Login failed. Please try again.", });

      return;
    }
  };
```

This is an async function that gets one argument. A session is being created by taking the email and password from the form and signing in the user with the signInAccount function. This also handles that if the user already has an active session, he is being redirected to the private routes, in this case the root page being the Home Feed.

SignupForm.tsx

This file is setup almost the same way as the signin, but with more form fields and an additional drop down menu using a shadcn component.

Dropdown

```
<DropdownMenu>
  <DropdownMenuTrigger className="bg-dark-4 text-left px-4 py-2
  outline outline-2 rounded outline-dark-4 flex justify-between
  items-center w-full">
    <span>{selectedRoles.length > 0 ? selectedRoles.join(', ') :
    'Rolle auswaehlen'}</span>
    <span className="ml-2">^</span>
  </DropdownMenuTrigger>
  <DropdownMenuContent className="bg-dark-1 border-4 border-dark-4
  w-full">
    <DropdownMenuLabel>Choose Roles</DropdownMenuLabel>
    {["Ecurie-Aix", "Alumni", "Partner", "Hersteller"].map(role
    => (
      <DropdownMenuCheckboxItem
        key={role}
        checked={selectedRoles.includes(role)}
        onCheckedChange={() => handleRoleChange(role)}
        className="hover:bg-ecurie-babyblue"
      >
        {role}
      </DropdownMenuCheckboxItem>
    ))}
  </DropdownMenuContent>
</DropdownMenu>
```

This specific Box for example allows us to allocate roles to our users. In future checks for roles, we will always regard the first one in the array.

RoleChangeHandler

```
const handleRoleChange = (role: string) => {
  const updatedRoles = selectedRoles.includes(role)
    ? selectedRoles.filter(r => r !== role)
    : [...selectedRoles, role];
  setSelectedRoles(updatedRoles);
  form.setValue("role", updatedRoles);
};
```

already Logged In

```
if (isLoggedIn) {
    form.reset();
    navigate("/");
  } else {
    toast({ title: "Login failed. Please try again." });
    return;
  }
} catch (error) {
  console.log({ error });
}
};
```

This code checks whether the user is already authenticated and wields a fallback cookie, if there is one, we reset our form so that no data can be misinterpreted and navigate to the root directory being our Home Page in the private routes.

3.4.1 _root

This directory contains all of our pages that can be visible on our website.

RootLayout

RootLayout

```
const RootLayout = () => {
  return (
    <div className="w-full md:flex">
      <Topbar />
      <LeftSidebar />

      <section className='flex flex-1 h-full'>
        <Outlet />
      </section>

      <Bottombar />
    </div>
  )
}
```

This determines the layering of our components, which in this case states that the Topbar or the LeftSidebar is being rendered, based on our resolution with the Topbad being for mobile and the LeftSidebar for desktop resolution. The Outlet component then renders the corresponding child components as sepcified in the AuthLayout file.

pages

Now we cover our main pages.

AllUsers

This is the dedicated Users page which shows all users.

GetUsers Hook

```
const { data: creators, isLoading, isError: isErrorCreators }
= useGetUsers();
```

This hook uses an appwrite api function (specified under "./lib/appwrite/api.ts"), which returns users.

User Grid

```
<div className="common-container">
      <div className="user-container">
        <h2 className="h3-bold md:h2-bold text-left w-full">
        Alle Nutzer</h2>
        {isLoading && !creators ? (
          <Loader />
        ) : (
          <ul className="user-grid">
            {creators?.documents.map((creator) => (
              <li key={creator?.$id} className="flex-1
              min-w-[200px] w-full  ">
                <UserCard user={creator} />
              </li>
            ))}
          </ul>
        )}
      </div>
    </div>
```

This abomination of code uses the "ul" component, which forms a list and displays our users. To achieve that, we mapped over our returned documents from the GetUsers hook and displayed their UserCard in a grid, as specified in the globals.css file.

CreatePost

This file let's us create new Posts (omg fr?)

The basic file structure is fairly simple, but here comes the magic:

CreatePost

```
<PostForm action="erstellen" />
```

Here, we display our PostForm component and give it the argument "erstellen" (engl.: "create"). This allows us to reuse this component for the PostEdit component, where we then pass the "bearbeiten" (engl.: "update") property, which then calls a different function that does not create a new document in our back-end, but updates an already existing one.

EditPost

see CreatePost.

Explore

This is the page, where we provide a search bar and a filter option for filtering and searching posts.

Infinite Scroll

```
useEffect(() => {
if (inView && !searchValue) {
  fetchNextPage();
}
}, [inView, searchValue]);
```

This React hook function is being called whenever the "inView" or the "searchValue" variable is being updated. The inView variable updates if the end of our current page is being reached and the searchValue updates if the search bar is being used. We then fetch the next page, which uses the GetPosts hook above.

RoleChange

```
const handleRoleChange = (role: string) => {
// Toggle the selection of the role
if (selectedRoles.includes(role)) {
  setSelectedRoles(selectedRoles.filter(r => r !== role));
} else {
  setSelectedRoles([...selectedRoles, role]);
}
};

const filterPostsByRoles = (posts: any[]) => {
if (selectedRoles.length === 0) {
  return posts;
}

// Filter posts based on selected roles
return posts.filter(post => {
  // Check if any selected role matches the post creator's role
  return selectedRoles.some(selectedRole =>
  post.creator.role.includes(selectedRole));
});
};
```

This works the same way like the one in the signup form. We fetch the roles that are being selected in the dropdown and add them to a set, which we then check for roles to apply the filter.

Search Error Handling

```
const shouldShowSearchResults = searchValue !== "";
const shouldShowPosts = !shouldShowSearchResults &&
    posts.pages.every((item) => item.documents.length === 0);
```

This piece of code specifies that the search results should only be shown if the searchValue is not the placeholder and if there actually are posts to be shown for that specific search.

Search Handling

```
<div className="explore-inner_container">
    <h2 className="h3-bold md:h2-bold w-full">Beitraege suchen</h2>
    <div className="flex gap-1 px-4 w-full rounded-lg bg-dark-4">
      <img
        src="/assets/icons/search.svg"
        width={24}
        height={24}
        alt="search"
      />
      <Input
        type="text"
        placeholder="Suchen"
        className="explore-search"
        value={searchValue}
        onChange={(e) => setSearchValue(e.target.value)}
      />
    </div>
  </div>
```

This means, that if the searchValue changes, the setSearchValue is being triggered, which then sets the searchValue to the corresponding value that was being typed in.

Dropdown

```
<div className="flex-center gap-3 rounded-xl px-4 py-2">
  <DropdownMenu>
    <DropdownMenuTrigger className="text-left bg-dark-3 rounded-xl
    gap-2 px-4 py-2 flex justify-between items-center w-full">
      <span>{selectedRoles.length > 0 ? selectedRoles.join(', ')
      : 'Nach Rolle sortieren'}</span>
      <img
        src="/assets/icons/filter.svg"
        width={20}
        height={20}
        alt="filter"
      />
    </DropdownMenuTrigger>
    <DropdownMenuContent className="bg-dark-1 border-4
    border-dark-4 w-full">
      {["Ecurie-Aix", "Alumni", "Partner", "Hersteller"]
      .map(role => (
        <DropdownMenuCheckboxItem
          key={role}
          checked={selectedRoles.includes(role)}
          onCheckedChange={() => handleRoleChange(role)}
          className="hover:bg-ecurie-babyblue"
        >
          {role}
        </DropdownMenuCheckboxItem>
      ))}
    </DropdownMenuContent>
  </DropdownMenu>
</div>
```

This is our filter dropdown menu, which is an imported component from shadcn.

Search Results

```
<div className="flex flex-wrap gap-9 w-full max-w-5xl">
    {shouldShowSearchResults ? (
      <SearchResults
        isSearchFetching={isSearchFetching}
        searchedPosts={searchedPosts}
      />
    ) : shouldShowPosts ? (
      <p className="text-light-4 mt-10 text-center w-full">
      Ende der Beitraege</p>
    ) : (
      posts.pages.map((page: any, index: number) => (
        <GridPostList key={`page-${index}`} posts=
        {filterPostsByRoles(page.documents)} />
      ))
    )}
</div>
```

Home

This is our Home page.

Event Fetching

```
const getEvents = (calendarID: string, apiKey: string): void => {
    function initiate() {
      gapi.client
        .init({
          apiKey: apiKey,
        })
        .then(() => {
          return gapi.client.request({
            path: `https://www.googleapis.com/calendar/v3/calendars/
            ${calendarID}/events`,
          });
        })
        .then(
          (response: any) => {
            const events = response.result.items;
            setEvents(events);
          },
          (err: any) => {
            console.error("Error fetching events:", err);
// Log any errors
          }
        );
    }
    gapi.load('client:auth2', initiate);
  };
```

This hefty piece of code fetches data from googles services and returns all events from a specified

calendar. How does it work? No clue. Check this out:

`https://developers.google.com/docs/api/reference/rest`

Event Fetching

```
useEffect(() => {
if (calendarID && apiKey) {
  getEvents(calendarID, apiKey);
} else {
  console.error("Missing required environment variables");
}
}, [calendarID, apiKey]);
```

This hook makes sure that on every page reload, the events get fetched properly.

Event Sorting

```
const filteredEvents = events
.filter((event) => {
  const eventDate = new Date(event.start.dateTime);
  return eventDate >= new Date();
})
.sort((a, b) => {
  const dateA = new Date(a.start.dateTime).getTime();
  const dateB = new Date(b.start.dateTime).getTime();
  return dateA - dateB;
});
```

This function sorts the events by time of event and sorts out all the events that are prior to the current date and time.

Post List

```
<ul className="flex flex-col flex-1 gap-9 w-full">
  {posts?.documents.map((post: Models.Document) => (
    <li key={post.$id} className="flex justify-center w-full">
      {/* Pass the media type to PostCard */}
      <PostCard post={post} />
    </li>
  ))}
</ul>
```

This list renders all the posts being returned by the search and filter options.

Event List

```
<div className="home-creators">
    <h2 className="h3-bold md:h2-bold text-left w-full">
    Anstehende Events</h2>
    {filteredEvents.length === 0 ? (
      <p className="text-light-1">Bisher stehen keine
      #Events an.</p>
    ) : (
      <ul>
        {filteredEvents.map((event) => (
          <li key={event.id} className="">
            {/* Pass summary and start props to
            Event component */}
            <Event summary={event.summary}
            start={event.start} />
          </li>
        ))}
      </ul>
    )}
</div>
```

This renders all the filtered events in a sidebar and if there are no events, it states it properly.

index.ts

This file specifies the exports of all our main pages for easier importing.

LikedPosts

This is a page on the profile.

LikedPosts

```
<>
  {currentUser.liked.length === 0 && (
    <p className="text-light-4">Keine Beitr ge mit "Gef llt mir"
    markiert</p>
  )}

  <GridPostList posts={currentUser.liked} showStats={false} />
</>
```

Pretty self explanatory.

PostDetails

This is the page that is being shown when we click on a post in the home or the explore page.

MediaType

```
const isImage = mimeType && mimeType.startsWith('image/');
const isVideo = mimeType && mimeType.startsWith('video/');
const isPDF = mimeType && mimeType === 'application/pdf';
```

This code determines the media type, to determine which component to choose to render the file.

Back Button

```
<Button
  onClick={() => navigate(-1)}
  variant="ghost"
  className="shad-button_ghost">
  <img
    src={"/assets/icons/back.svg"}
    alt="back"
    width={24}
    height={24}
  />
  <p className="small-medium lg:base-medium">Zurueck</p>
</Button>
```

Media Depiction

```
<a href={post?.imageUrl} target="_blank" className="w-1/2">
    {isImage && (
      <img
        src={fileUrl}
        alt="post image"
        className="file_uploader-img"
      />
    )}
    {isVideo && (
      <video controls className="file_uploader-img">
        <source src={fileUrl} type="video/mp4" />
        Your browser does not support the video tag.
      </video>
    )}
    {isPDF && (
      <embed
        src={fileUrl}
        width="100%"
        height="500px"
        type="application/pdf"
        className="file_uploader-img"
      />
    )}
    {/* Default fallback for other file types */}
    {!isImage && !isVideo && !isPDF && (
      <div className="post-card_fallback">
        <p>Unsupported file type</p>
      </div>
    )}
</a>
```

This now determines with the media type variables being set, which component to render to display the property accordingly. If we have an Image, it is being rendered via the img tag, if there is a video, a video component is being rendered (if the browser supports it) and the same to the pdf type. The a tag around all of it is to make the file clickable, so that you can fully view it in a new tab (_blank manages that).

Media Deletion

```
<Button
   onClick={handleDeletePost}
   variant="ghost"
   className={`ost_details-delete_btn ${user.id !== post?.creator.$id
  && "hidden"
     }`}>
  <img
     src={"/assets/icons/delete.svg"}
     alt="delete"
     width={24}
     height={24}
  />
</Button>
```

This button being rendered as an svg calls the handleDeletePost function upon being clicked. For further reference, check the api.ts file doc.

Render Corresponding Posts

```
<div className="w-full max-w-5xl">
    <hr className="border w-full border-dark-4/80" />

    <h3 className="body-bold md:h3-bold w-full my-10">
      Aehnliche Beitraege
    </h3>
    {isUserPostLoading || !relatedPosts ? (
      <Loader />
    ) : (
      <GridPostList posts={relatedPosts} />
    )}
</div>
```

Profile

LinkedIn-Link Fix

```
if (currentUser?.linkedin) {
    // Ensure the LinkedIn URL is complete
    linkedInUrl = currentUser?.linkedin.startsWith('http')
      ? currentUser.linkedin
      : `https://${currentUser?.linkedin}`;
  }
  else {
    linkedInUrl = "https://www.youtube.com/watch?v=dQw4w9WgXcQ"
  }
```

This function takes the input of whatever link the user types into the "LinkedIn-Link" input-form when editing their profile. If the user types in the link with 'http' infront the function changes nothing about it, but if the user does not, then the function automatically adds 'https' as a prefix to ensure

that the hyperlink-reference works, which we will need for the LinkedIn-Button to work. More later.

Optional Display of Phone-Number

```
<div className="flex gap-2 pt-4">
    {currentUser.telefon_nr !== null && currentUser.telefon_nr !== ''
    && (
      <>
        <img
          src={"/assets/icons/phone-icon.svg"}
          alt="edit"
          width={20}
          height={20}
        />
        <p className="">
          {currentUser.telefon_nr}
        </p>
      </>
    )}
</div>
```

As an example of all the divs in this file, we'll take a look at this one. This div adds the phone-icon, which can be seen below the username to ones page and writes out the users telephone number if previously added in the 'Edit Profile'-Tab. Now, it is important to note, that as it is optional for a user to input their telephone-number, we need to also deal with the situation if the user does not enter one. This is done by the syntax $x! == null x! == "$ (. This means that when the input-form of x, in our case the $currentUser.telefon_nr$, is not null and not empty, then the phone-icon and the telephone number of the user should be displayed. If it is null or empty then it should not, it is as simple as that. This is only implemented to counteract a random phone-icon being displayed on the profile.

LinkedIn-Button

```
<div className="flex-left pt-2 pr-64">
    {currentUser.linkedin !== '' && currentUser.linkedin !== null && (
      <a href={linkedInUrl} target="_blank" className="h-8 bg-dark-4
      text-light-1 flex items-center rounded-lg px-2 w-fit">
        <img
          src="/assets/icons/linkedin-icon.svg"
          alt="linkedin"
          width={20}
          height={20}
          className="mr-2"
        />
        <span>|</span>
        <span className="pt-1 pl-2">{currentUser.name}</span>
      </a>
    )}
</div>
```

This code displays the LinkedIn-button on the each profile. What this does is, if the user has input a LinkedIn-Link in their 'Edit Profile'-Tab the button while be displayed and on-click a new tab will open itself and go to the link put in the form on the profile. If the user does not put in any link into

the form, then the button will not be displayed.

Beiträge — Gefällt-Mir Button

```
{currentUser.$id === user.id && (
        <div className="flex max-w-5xl w-full">
          <Link
            to={`/profile/${id}`}
            className={`profile-tab rounded-l-lg ${pathname ===
            `/profile/${id}` && "!bg-dark-3"
              }`}>
            <img
              src={"/assets/icons/add-post.svg"}
              alt="posts"
              width={20}
              height={20}
            />
            Beitraege
          </Link>
          <Link
            to={`/profile/${id}/liked-posts`}
            className={`profile-tab rounded-r-lg ${pathname ===
            `/profile/${id}/liked-posts` && "!bg-dark-3"
              }`}>
            <img
              src={"/assets/icons/like.svg"}
              alt="like"
              width={20}
              height={20}
            />
            Mit "Gefaellt mir" markiert
          </Link>
        </div>
      )}
```

This part of the code displays the buttons on the profile, where the user can switch between posts the posted themselves and posts they liked from other creators. Note that only the user that the profile belongs to can switch between these tabs, visitor-user can only see the uploaded posts.

Grid Formation of Posts

```
<Route
      index
      element={<GridPostList posts={currentUser.posts}
      showUser={false}
      />}
    />
    {currentUser.$id === user.id && (
      <Route path="/liked-posts" element={<LikedPosts />} />
    )}
</Routes>
```

This route-protocols job is to display the posts uploaded by the profile, which the user is on, in a grid formation. If the user is the owner of the profile and thus the option of viewing their liked posts is available the route displays the in a groid formation as well.

Saved

Fetching of Saved Posts

```
const savePosts = currentUser?.save
    .map((savePost: Models.Document) => ({
      ...savePost.post,
      creator: {
        imageUrl: currentUser.imageUrl,
      },
    }))
    .reverse();
```

This code fetches the saved posts of the user, converts it to *...savePost.post*, which breaks the large block of saved posts into singular posts and displays them. After all that the image of the current users profile picture is displayed at the bottom-left of each post.

UpdateProfile

This file lets us access the edit profile page on our profile.

default values

```
const form = useForm<z.infer<typeof ProfileValidation>>({
    resolver: zodResolver(ProfileValidation),
    defaultValues: {
      file: [],
      name: user.name,
      username: user.username,
      email: user.email,
      bio: user.bio || "",
      abteilung: user.abteilung || "",
      telefon_nr: user.telefon_nr || "",
      linkedin: user.linkedin || "",
    },
  });
```

This declares the default values for the properties of each account, so that we do not operator on null.

handle update

```
const handleUpdate = async (value: z.infer<typeof ProfileValidation>)
=> {
  const updatedUser = await updateUser({
    userId: currentUser.$id,
    name: value.name,
    bio: value.bio,
    file: value.file,
    imageUrl: currentUser.imageUrl,
    imageId: currentUser.imageId,
    abteilung: value.abteilung,
    telefon_nr: value.telefon_nr,
    linkedin: value.linkedin,
  });

  if (!updatedUser) {
    toast({
      title: `Update user failed. Please try again.`,
    });
  }

  setUser({
    ...user,
    name: updatedUser?.name,
    bio: updatedUser?.bio,
    imageUrl: updatedUser?.imageUrl,
    abteilung: updatedUser?.abteilung,
    telefon_nr: updatedUser?.telefon_nr,
    linkedin: updatedUser?.linkedin,
  });
  return navigate(`/profile/${id}`);
};
```

this updates the properties to the backend using updateUser() fom appwrite.

3.5 forms

FileUploader

This lets us upload files to the backend, which is used in the create post section and the update post section.

upload files

```
const FileUploader = ({ fieldChange, mediaUrl }: FileUploaderProps)
=> {
const [file, setFile] = useState<File[]>([]);
const [fileUrl, setFileUrl] = useState(mediaUrl);

const onDrop = useCallback((acceptedFiles: FileWithPath[]) => {
  setFile(acceptedFiles);
  fieldChange(acceptedFiles);
  const newFileUrl = URL.createObjectURL(acceptedFiles[0]);
  setFileUrl(newFileUrl);
}, [fieldChange]); // Note: remove 'file' from dependency
array to avoid unnecessary recreations of onDrop

const { getRootProps, getInputProps } = useDropzone({
  onDrop,
  accept: {
    'image/*': ['.png', '.jpeg', '.jpg', '.svg'],
    'application/pdf': ['.pdf'],
    'video/mp4': ['.mp4', '.mov'],
  },
});
```

First, we have our states for the media, where the fileUrl will be the link to the media in the backend. the onDrop function enables us to drag and drop our media onto the upload area and use it.

PostForm

This is just straight from their documentation and some ChatGPT. Everything should be explained in the codebase and if not then idk lol

3.6 shared

These are all the components that are used throughout the application.

BetaDisclaimer

This might not even be there anymore when you read this, but it is/ was a Dialog field which opened upon click and showed the Beta progress. For the dialog, you can look up the shadcn docs.

Bottombar

This is for the mobile version of the app, which doesnt work really different then the left sidebar.

Brightness

This is the light/ darkmode trigger on the Home Screen.

switch animation

```
useEffect(() => {
  setIsTransitioning(true);
  const timeout = setTimeout(() => setIsTransitioning(false), 500);
  // Adjust the duration as needed
  return () => clearTimeout(timeout);
}, [theme]);
```

This is the animation that triggers whenever the button is pressed and the theme variable gets changed.

Event

This is the component that is being used on the right sidebar on the homescreen, where all the events are displayed.

events

```
const eventDate = new Date(start.date).toLocaleDateString();
    const eventTime = new Date(start.date).toLocaleTimeString([],
    { hour: '2-digit', minute: '2-digit' });

    // Generate Google Calendar event creation link
    const googleCalendarLink = `https://www.google.com/calendar/render?action=TE
    ${encodeURI(start.date)}&details=${encodeURI(summary)}`;
```

This defines the date and time of the event and connects the corresponding calendar.

*GridPostList This is the component being used in the explore tab. Nothing much to say here, just fetching the posts and displaying them.

LeftSidebar

This is the left sidebar that is being visible everywhere.

name color

```
useEffect(() => {
    const role = user?.role[0] || '';
    console.log(user.role)
    if (role === 'E') {
        setNameColor('text-ecurie-babyblue');
    } else if (role === 'A') {
        setNameColor('text-ecurie-babyblue');
    } else if (role === 'P') {
        setNameColor('text-ecurie-blue');
    }
    else if (role === 'H') {
        setNameColor('text-ecurie-blue');
    }
    else {
        setNameColor('text-ecurie-darkred');
    }
}, [user]);
```

This is some noodle code, that we are not proud of, but it works, yay. What it does is that it takes the user role, looks at the first letter and applies the corresponding color.
The rest of this file is just basic HTML.

Loader

This is the loading svg that is being displayed if posts are not fully fetched yet.

PostCard

This is the component that wraps the posts. This is seen on the Home Page. Here, we take the role, the color, the tags and the media type to display the content accordingly.

PostStats

This is the like and save count at the bottom of the Posts. All pretty basic.

ProfileUploader

This works a bit like the FileUploader, but works for the profile picture with some tweaks, because we did not want mp4s to be accepted as a profile picture lol

ThemeContext

This is the context necessary for the light-/darkmode switch.

Topbar

This is also a component that is only being used in the mobile version of the app. It displays the logo, the profile picture (clickable btw) and the logout svg.

UserCard

This is the component you can see when being in the peoples tab. Here, we just take the link, the role and the color.

3.7 ui

These are all the components we used from shadcn.

shadcn

```
https://ui.shadcn.com/docs/components/accordion
```

3.8 constants

index

This file declares the links and icons for the left and bottom sidebar.

3.9 context

AuthContext

This file provides the Authentication service needed for the user to be logged in automatically after being authorized.

3.10 hooks

"Debouncing is removing unwanted input noise from buttons, switches or other user input. Debouncing prevents extra activations or slow functions from triggering too often."

3.11 lib

Now to the fun part. Api. Appwrite. I hate it here.

appwrite

api

This file contains all the necessary api functions to access and manipulate the backend data. Lets have a quick run through:

createUserAccount

```
/**
 * Creates a new user account and saves the user data to the database.
 *
 * @param user - The user object containing the necessary
 information for creating a new account.
 * @param user.email - The email of the user.
 * @param user.password - The password of the user.
 * @param user.name - The name of the user.
 * @param user.role - The role(s) of the user.
 * @param user.username - The username of the user.
 *
 * @returns A promise that resolves to the newly created
 user document if successful, or an error if failed.
 */
export async function createUserAccount(user: INewUser) {
  try {
    const newAccount = await account.create(
      ID.unique(),
      user.email,
      user.password,
      user.name,
    );

    if (!newAccount) throw Error;

    const avatarUrl = avatars.getInitials(user.name);

    const newUser = await saveUserToDB({
      accountId: newAccount.$id,
      name: newAccount.name,
      email: newAccount.email,
      role: user.role,
      username: user.username,
      imageUrl: avatarUrl,
    });

    return newUser;
  } catch (error) {
    console.log(error);
    return error;
  }
}
```

getFileById

```
/**
 * Retrieves a file from Appwrite storage using the
 provided post's
 imageId.
 *
 * @param post - The post object containing the
 imageId of the file
 to retrieve.
 * @returns A promise that resolves to the file
 object if successful, or
 rejects with an error if unsuccessful.
 *
 * @example
 * ```typescript
 * const post: IUpdatePost = {
 *   postId: "12345",
 *   caption: "Sample post",
 *   imageUrl: "https://example.com/image.jpg",
 *   imageId: "67890",
 *   location: "New York",
 *   tags: "sample, post",
 *   mimeType: "image/jpeg",
 * };
 *
 * getFileById(post)
 *   .then((file) => {
 *     console.log("File retrieved successfully:", file);
 *   })
 *   .catch((error) => {
 *     console.error("Failed to retrieve file:", error);
 *   });
 * ```
 */
export async function getFileById(post: IUpdatePost){
  const result = await storage.getFile(
    appwriteConfig.storageId,
    post.imageId,
  )

  return result
}
```

saveUsertoDB

```
/**
 * Saves a new user to the database.
 *
 * @param user - An object containing user details.
 * @param user.accountId - The unique identifier of
 the user account.
 * @param user.email - The user's email address.
 * @param user.name - The user's full name.
 * @param user.imageUrl - The URL of the user's profile image.
 * @param user.username - The user's username (optional).
 * @param user.role - An array of user roles.
 *
 * @returns A promise that resolves to the newly created user
 document if successful,
 *          or rejects with an error if the operation fails.
 */
export async function saveUserToDB(user: {
  accountId: string;
  email: string;
  name: string;
  imageUrl: URL;
  username?: string;
  role: string[];
}) {
  try {
    const newUser = await databases.createDocument(
      appwriteConfig.databaseId,
      appwriteConfig.userCollectionId,
      ID.unique(),
      user
    );

    return newUser;
  } catch (error) {
    console.log(error);
  }
}
```

signInAccount

```
/**
 * This function is used to sign in a user with their email
 and password.
 *
 * @param user - An object containing the user's email
 and password.
 * @param user.email - The email of the user.
 * @param user.password - The password of the user.
 *
 * @returns A promise that resolves to the session object
 if the sign-in is successful.
 * If the sign-in fails, the promise rejects with an error.
 *
 * @throws Will log the error to the console if the sign-in fails.
 */
export async function signInAccount(user: { email: string;
password: string }) {
  try {
    const session = await account.createEmailSession(user.email,
    user.password);

    return session;
  } catch (error) {
    console.log(error);
  }
}
```

getAccount

```
/**
 * Retrieves the current user's account information
 from Appwrite.
 *
 * @returns {Promise<any>} A promise that resolves to the
 current user's account information.
 * If the retrieval fails, the promise will reject with
 an error.
 *
 * @example
 * ```typescript
 * const currentAccount = await getAccount();
 * console.log(currentAccount);
 * ```
 */
export async function getAccount() {
  try {
    const currentAccount = await account.get();

    return currentAccount;
  } catch (error) {
    console.log(error);
  }
}
```

with this knowledge, you should be able to understand the rest of the api aswell. If not, you can ask Tabnine (extension for VSCode).

config

This is the api config, where the endpoints are set.

validation

The index file in this directory manages the validations. E.g.: the SignupValidation checks, if the name has at least 2 letters.

utils

This file provides utility functions such as the time computing of how long a post has been up for or the name colors depending on the given roles.

3.12 types

The file in this folder manifests the types of the necessary user objects. E.g. IUser is the default structure of a user object, which needs to have an id, a name and so on.