Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

FreelanceHub

Project Engineering

Michael Agbo G00374265

Bachelor of Engineering (Honours) in
Software and Electronic Engineering

Atlantic Technological University

2023 / 2024

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Software & Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Acknowledgements

I would like to thank my project supervisor, Brian O'Shea, for his assistance and guidance throughout the duration of my final year project.
I would also like to thank my Project Engineering coordinators Paul Lennon, Michelle Lynch, and Niall O'Keeffe for their feedback and tutelage on project work throughout the year.

Table of Contents

1 Summary

For my final year project I decided to tackle familiar issue to everyone at some point, requiring help. I often embark on coding projects that interest or excite me, and in my ventures I often encounter hurdles in development and troubleshooting. Recognising this is a shared obstacle in the tech community for hobbyists and professionals alike, I created FreelanceHub. Freelancehub is a web application that allows individuals to seek assistance for their coding dilemmas and optionally offer monetary rewards as gratitude for the solutions.

In the creation of FreelanceHub I utilized a new suite of technologies in order to experiment with the use of a cutting edge tech stack,

The use of tRPC ensured the communication between the frontend and backend was completely type-safe and reliable minimizing errors. For effective content management I incorporated Payload CMS, which provided our admin dashboard that allowed me to adapt it to the needs of my dynamic content.

The full stack framework used was the newest version of Next.js version 14, in which the leveraging of express middleware combined with Node.js, for the backend. This forms our environment for us handling server side logic and API requests. MongoDB was used as the database due to my familiarity with it.

On the frontend I used React with Typescript, using Reacts component-based structure and Typescripts type safety to minimize bugs and ensure type safety. Tailwind CSS was used along with shadcn/ui to handle consistent styling an UI components.
I utilized a GitHub repository to store and manage my project code.

This selection of technologies facilitated the development of a robust and completely type safe application.

In the development on this project I Improved my software skills through working with technologies I had no experience with and building a web application with them from the ground up, the experience I gained is invaluable and encouraged me to engage in future project investigating the use of unfamiliar technologies.

2 Project Architecture

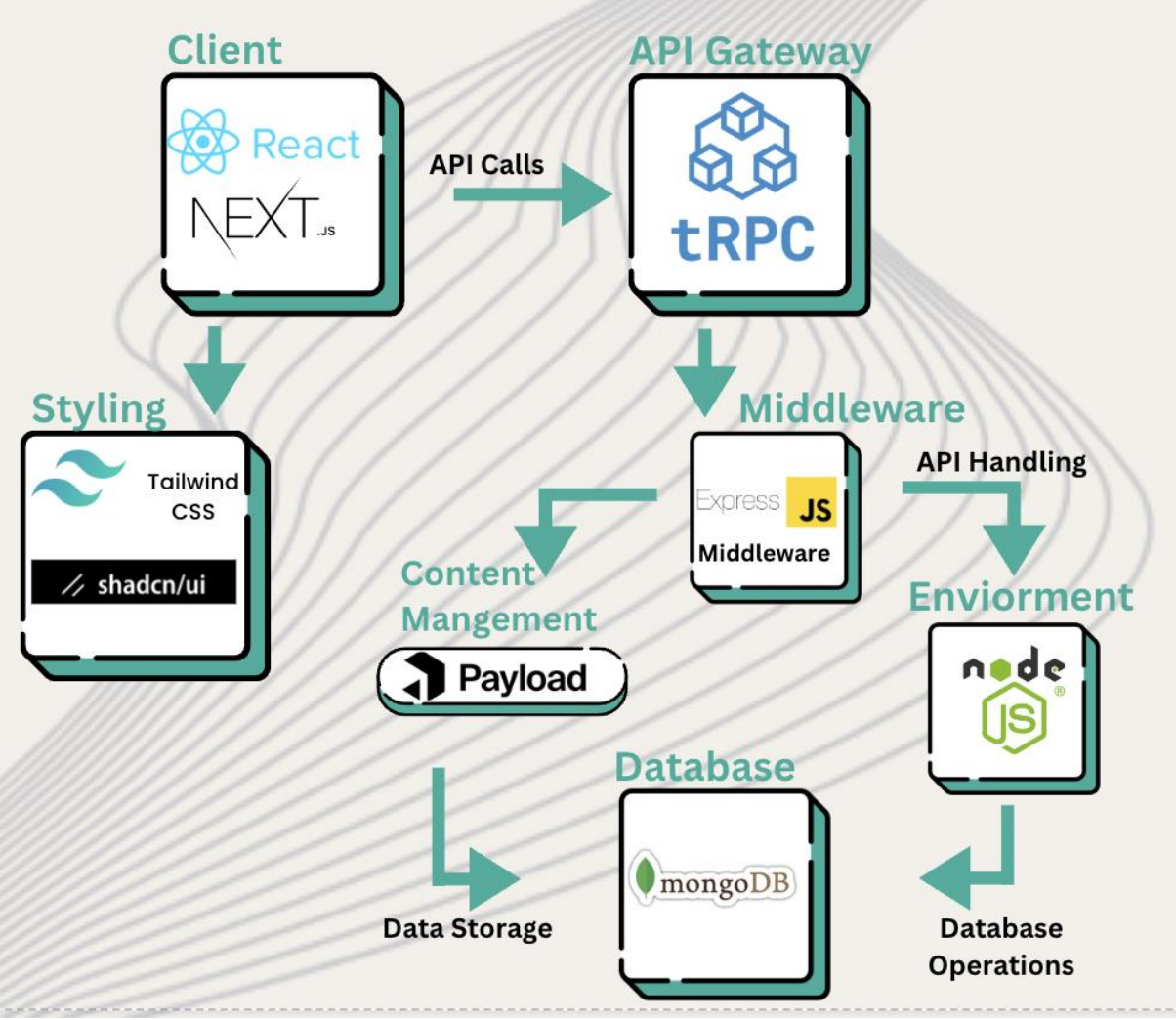


Figure 2.1 – Project Architecture

3 Introduction

3.1 Project Goals

The goals of this project are as follows:

- To create a question and answer web application
- Allow users to post questions
- Allow users to give answers
- Allow users to upvote and downvote replies

3.2 Motivation for tech stack

The selection of typescript came first as I am currently employed as a frontend developer in Genesys and I work mainly in typescript, through research finding out I could write the backend and frontend in typescript peaked my interest which is why I took this tech stack on, the payload integration aided massively as this project idea relies heavily on admin interjection so having a admin dashboard I could integrate with was a no brainer.

3.3 Conclusion

Freelancehub is designed to be a great solution for knowledge sharing aiming to tackle a real world issue and fulfil academic requirements of my final year software project.

4 Poster

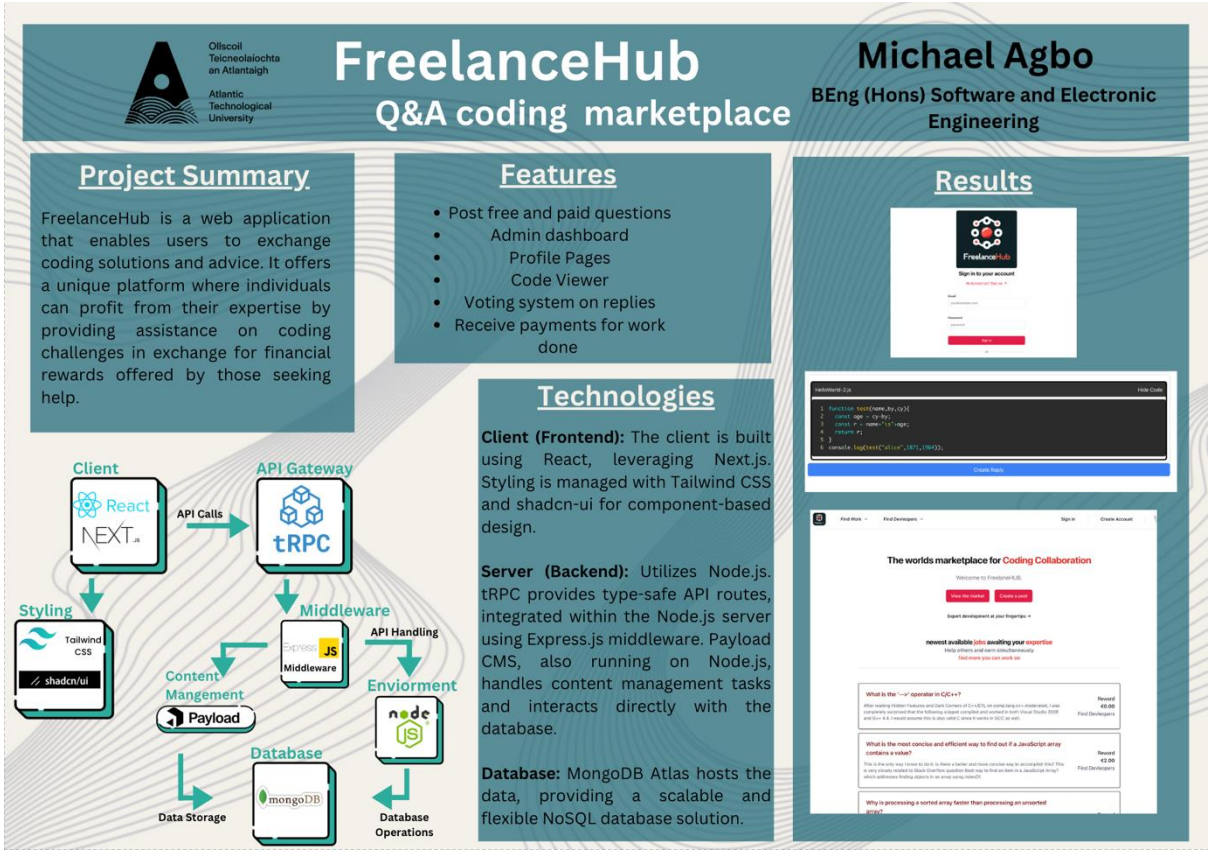


Figure 4.1 – Poster

5 Technologies

5.1 Next.js 14

Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations. [1]

Next.js 14 handled both my client side and server side operations, by utilizing Node.js as the backbone of for its API routes it facilitates the building of the creation of server side APIs without the need for an external server. It also allows for the use of express middleware and routing capabilities, which I make use of in this project.

5.2 tRPC

tRPC is for full-stack TypeScript developers. It makes it easy to write endpoints that you can safely use in both the front and backend of your app. Type errors with your API contracts will be caught at build time, reducing the surface for bugs in your application at runtime. [2]

The use of tRPC allowed me to seamlessly integrate with Payload CMS, allowing for full type safety across client-server communications.

5.3 Payload CMS

Payload is a headless CMS and application framework. It's meant to provide a massive boost to your development process, but importantly, stay out of your way as your apps get more complex. [3]

Payloads integration was a major element of the project as it allowed me to create an admin dashboard in which I could facilitate content management and facilitate the handling of API routes. The payload dashboard allowed me to create an admin dashboard.

5.4 MongoDB Atlas

MongoDB Atlas is a cloud database service that provides mongos NoSQL database capacity on a reliable cloud infrastructure.

5.5 Tailwind CSS

Tailwind CSS works by scanning all of your HTML files, JavaScript components, and any other templates for class names, generating the corresponding styles and then writing them to a static CSS file. [4]

5.6 Shadcn/iu

Shadcn was used for UI components, open source, customizable and accessible Shadcn is a great choice a great choice for components.

6 Collections

In this section I will discuss the collections I have in the project.

6.1 Users

Firstly let's have a look at the users collection, probably the most basic collection so a good starting point but as I will show as we progress through the collections, this collection holds a large bearing on the others.

6.1.1 Collection Config

Here I use the type CollectionConfig from payload/types to ensure type safety for the collection.

```
export const Users: CollectionConfig = {
```

Figure 6.1

6.1.2 Slug

This is the unique identifier for the collection used in URL and database operations set here as users.

```
  slug: "users",
```

6.1.3 Auth

This specifies the authentication settings for the collection

6.1.3.1 Verify

This contains the method related to our email verification.

6.1.3.2 generateEmailHTML

Returns a html string for email verification, it uses the environment variable to create the link attaching a verification token as a query parameter.
// remove

```
auth: {
  verify: {
    generateEmailHTML: ({ token }) => {
      return `<a href='${process.env.NEXT_PUBLIC_SERVER_URL}/verify-email?token=${token}'>Please verify Email</p>`;
    },
  },
},
```

Figure 6.2

6.1.4 Access

This defines the access control for the collection, for this we have enabled the ability to read and create.

```
access: {
  read: () => true,
  create: () => true,
},
```

Figure 6.3

6.1.5 Fields

Here we define the fields that the user can contain, these are all rather similar they require a label and a value and optionally accept more fields, for example required stating whether the fields is required or not and the type relevant to how the field behaves.

```
fields: [
  {
    name: "role",
    admin: {
      condition: ({req}) => req.user.role === "admin"
    },
    type: "select",
    options: [
      { label: "Admin", value: "admin" },
      { label: "User", value: "user" },
    ],
  },
  {
    name: "biography",
    label: "biography",
    type: "textarea",
    required: false,
  },
  {      You, 3 weeks ago • profile pages and workview
    name: "ProfilePicture",
    type: "upload",
    relationTo: "images"
  }

],
```

Figure 6.4

6.2 Work

Now the work collection this collection contains a lot more fields but as I have already spoke about these I will show some points of interest in this collection that we haven't looked at before

6.2.1 User

This field is of note due to its type, relationship. Essentially this field is related to our users collection so this field is a whole user object as part of our work object. The admin field condition is false as I don't want admin dashboard to show this field when creating a work.

Figure 6.5

6.2.2 Hooks

Now this is an interesting point to look at as I specified that I require a user on this collection but hid the field from the admin dashboard and don't allow users to add one when creating a work from the frontend, so how does the user get onto the document? I define a before change hook which I get from Payload collection types. This leads us to our add user hook.

6.2.2.1 Add User

This hook runs before the data changes, it checks whether the user ID is already included or otherwise it will use the user ID from the request. This ensures that a user Is present on the work allowing for ownership / responsibility tracking.

```
const addUser: BeforeChangeHook = ({ req, data }) => {
  const userId = data.user || req.user?.id;      You, 5
  if (userId) {
    return { ...data, user: userId };
  }
  return data;
};
```

Figure 6.6

6.2.3 Access

Here on the approved field I set the access for creating, reading and deleting to admins only. I do this by checking the users role from the request.

```
access: {
  create: ({ req }) => req.user.role === "admin",
  read: ({ req }) => req.user.role === "admin",
  update: ({ req }) => req.user.role === "admin",
},
```

Figure 6.7

6.3 Replies

The reply collection is for replies on work items. This collection contains mostly the same contents as other collections we have looked at but one change of note is the altered before change hook.

6.3.1 Add user and work

In the same way I added the user to the work I add a user to the reply, the change here is we add the work to the reply as well.

```
const addUserAndWork: BeforeChangeHook = ({ req, data }) => {
    const userId = data.user || req.user?.id;

    const work = data.work;

    const newData = {
        ...data,
        ...(userId && { user: userId }),
        ...(work && { work: work })
    };

    return newData;
};
```

Figure 6.7

6.4 Work files and images

These 2 collections are set up to accept files both consist of parts I have already shown but are collections of note.

6.5 Work order

Simple collection to store information about orders

```
const WorkOrder: CollectionConfig = {
  slug: 'workOrder',
  fields: [
    {
      name: 'work',
      type: 'relationship',
      relationTo: 'work',
      required: true,
    },
    {
      name: 'user',
      type: 'relationship',
      relationTo: 'users',
      required: true,
    },
    {
      name: '_isPaid',
      type: 'checkbox',
      required: true,
      defaultValue: false,
      admin: {
        position: 'sidebar',
        description: 'Indicates whether the order has been paid for.'
      },
      access: {
        read: ({ req }) => req.user.role === 'admin',
        create: () => false,
        update: () => false,
      },
    }
  ],
  access: {
    read: () => true,
    create: ({ req }) => Boolean(req.user),
    update: ({ req }) => req.user && req.user.role === 'admin',
    delete: ({ req }) => req.user && req.user.role === 'admin',
  },
  admin: {
    useAsTitle: 'WorkOrders',
  },
};
```

Figure 6.8

7 Components

My web application contains many components ranging in functionality, I will describe the points of interest in various components and give a summary of each components functionallity

// todo add components ss

7.1 Home main bar

The home main bar is a regularly reoccurring component in the app, it displays the work items so they can be viewed. Its functionality depends on the props I have defined.

7.1.1 Use client

This is seen in many places in the app, it allows for client side interactivity.

7.1.2 Interface

"HomeMainBarProps" is an interface which defines the properties expected by the component, it requires a title, sub(subheading) and a query object of type TQueryValidator. The "href" and the limit are optional indicated by the "?".

```
interface HomeMainBarProps {
  title: string;
  sub: string;
  href?: string;
  query: TQueryValidator;
  limit?: number;

}
```

Figure 7.1

7.1.3 Component definition

Here we can see the "FALLBACK_LIMIT" defined, this will be the query limit if there isn't one passed into the component. The props are passed into the component as type "HomeMainBarProps" which is the interface I created. The props are then de-structured.

```
const FALLBACK_LIMIT = 4;

const HomeMainbar = (props: HomeMainBarProps) => {

  const { title, sub, href, query} = props;
```

Figure 7.2

7.1.4 Data fetching

Here we get a look my data fetching with tRPC, we use an infinite query as its useful as this is potentially a large dataset. The only query parameter it requires is the limit, and as stated before if there is not one defined it will use the fallback limit. The second object provided is a configuration object which will provide the customized behaviour of the query. “getNextPageParam” is a function that will tell react query how to fetch the next page, it will receive the data from the last loaded page as its argument and return the parameter needed to fetch the next page.

```
const { data: queryResults } = trpc.getAllWorkForMarketplace.useInfiniteQuery(
  {
    limit: query.limit ?? FALLBACK_LIMIT,
    query,
  },
  { getNextPageParam: (lastPage) => lastPage.nextPage }
);

const workItems = queryResults?.pages.flatMap((page) => page.items);
```

Figure 7.3

7.1.5 Data processing

Here I safely access the pages array in the query results as it may be undefined by having it optional(?). Then the flat map method is used to flatted the array of pages into a single array of “items”. Each page object is expected to contain a property “items” that is an array of work items. I then define map to be an array that contains elements of type “Work” or null and initialize it as an empty array, followed by two conditions, the first checks if “workItems” is not null or undefined and if true it assigns it to map. Inversely the second condition checks that the “workItems” is null or undefined and if true it will fill an array with null values.

```
const workItems = queryResults?.pages.flatMap((page) => page.items);

let map: (Work | null)[] = [];
if (workItems && workItems.length) {
  map = workItems;
} else if (!(workItems && workItems.length)) {
  map = new Array<null>(query.limit ?? FALLBACK_LIMIT).fill(null);
}
```

Figure 7.4

7.1.6 Mapping and rendering work items

Here I make use of the map we have created my iterating over the array map using the map function. Each element of map (workItems) and the index "i" are provided as arguments. For each item in the map a "WorkListings" component is rendered. The key is a unique identifier for each child in the list.

```
<WorkListings
  workItem={workItems}
  index={i}
  key={`workItem-${i}`}
/>
```

Figure 7.5

7.2 Work Listings

In the home main bar analysis I mentioned the use of a component "WorkListings", now an analysis of this component will give us the full scope of what happens in home main bar. This component will dynamically display information about the work items.

7.2.1 Hooks

7.2.1.1 Use state

Here we utilize the use state to control the state of is visible, this react hook allows us to track state. We initialise it to false but by using the set is visible function I will change this later.

7.2.1.2 Use effect

This react hook allows me to change the is visible variable after a delay which is based on the index, essentially staggering the appearance of the items, it then cleans up by clearing the timeout when index changes or the component unmounts

```
const [isVisible, setIsVisible] = useState(false);

useEffect(() => {
  const timer = setTimeout(() => {
    setIsVisible(true);
  }, index * 100);

  return () => clearTimeout(timer);
}, [index]);
```

Figure 7.6

7.2.2 Conditional rendering of work placeholder

The condition set is to check if there is not a work item or the is visible variable is false and if so I return the work place holder which is a component which is not exported defined after this component in the same file as its only used here.

7.2.3 Content rendering

If there is a work item and the visibility is set to true I then render out the work item. As I pass the whole work to this component it is easy to access the properties of it I need without de-structuring by simply referencing the data by “workItem.title”.

7.2.4 Work place holder

For the workplace holder I render out a skeleton which will as the name describes will act as a placeholder for the work until it has rendered.

7.3 User work

The user work component works almost exactly the same as the as the home main bar, the only difference of note is that I use a different endpoint instead of get all from marketplace for the infinite query I use get all work on user. This will return all the work made by that user and only that user and display it using work listings the same way as before.

7.4 User account nav

This is a component which gets used by the navbar to render a dropdown menu when the email is clicked on, it renders a list of 3 items, one allows you to go to profile page, the second facilitates navigation to the admin dashboard, and the final one allows you to log out.

7.5 Reply listing

Here I use the tRPC procedure to get upvotes on replies

```
const { data: upvotes } = trpc.reply.getUpvotesOnReplies.useQuery(
  {
    replyId: replyId || "defaultID",
  },
  { enabled: !!replyId }
);
```

Figure 7.7

The I use the upvote mutation to update add upvote

```
const { mutate: upvote } = trpc.reply.addUpvotesOnReplies.useMutation({
  onSuccess: () => {
    console.log("Successfully upvoted");
  },
  onError: (error) => {
    console.error("Failed to upvote:", error.message);
  },
```

Figure 7.8

Then when the upvote is pressed if it wasn't already highlighted then it will use the mutation and increment the upvote and invalidate the query to refresh the upvotes.

```
if (!upVoteIsHighlighted) {
  upvote(
    { replyId: replyId || "defaultId" },
    {
      onSuccess: () => {
        utils.reply.getUpvotesOnReplies.invalidate();
      },
    }
  );
}
```

Figure 7.9

If it is highlighted and pressed then it will remove the downvote

```
if (upVoteIsHighlighted) {
  upVoteDec(
    { replyId: replyId || "defaultId" },
    {
      onSuccess: () => {
        utils.reply.getUpvotesOnReplies.invalidate();
      },
    }
  );
}
```

Figure 7.10

This implementation is then mirrored for the downvotes to complete the reply component.

7.6 File viewer

The file viewer component is designed to fetch and display the contents of a file uploaded by the user. It displays the contents of the file in a formatted and highlighted view using the react syntax highlighter library. Syntax highlighting component for React using the seriously super amazing lowlight and refractor by wooorm [5]. The implementation of this was easy as the docs are well defined. the files are hidden behind a collapsible container which when opened will display the formatted file.

7.7 Navigation Bar

This navbar implementation was adapted from the practical solution provided by joshtriedcoding[6]

7.7.1 NavBar

This component is a responsive user aware navigation bar that dynamically adjusts its content based on the users authentication status. It contains the user navigation and cart and uses another component NavItems. A point of interest in this component is the user fetching from by retrieving the cookies and then fetching the user details based on the cookie by using the "getServerSideUser" utility from payload

```
const nextCookies = cookies()
const { user } = await getServerSideUser(nextCookies)
```

Figure 7.11

Figure 7.12

7.7.2 Nav Items

The "NavItems" component is designed to dynamically manage and display my navigation items, each navigation item can be toggled opened and closed by clicking on it or outside of it. To achieve this I use a map function over the categories array, each item In the array maps to a nav item component. It also has a handle open function mapped to each nav item allowing me to check if the current index is open clicking it will close it.

```
{CATEGORIES.map((category, i) => {
  const handleOpen = () => {
    if (activeIndex === i) {
      setActiveIndex(null);
    } else {
      setActiveIndex(i);
    }
  };
  const isOpen = i === activeIndex;
```

Figure 7.13

Figure 7.14

7.7.3 Nav Item

"NavItem" is a simple component handles the individual navigation items that go into nav items, clicking on each will route you respectively.

7.8 Stripe registration for pay-outs

These components handle the font-end and make use of the API routes to set the user up to be able to receive payments from the platform.

7.8.1 Stripe Account setup

This component returns a button which allows will use the API route "createStripeAccount" to create a stripe user id.

```
const StripeAccountSetup = () => {

    const { mutate} = trpc.pay.createStripeAccount.useMutation({
      onError: (error) => {
        toast.error(error.message || "Failed to create Stripe account. Please try again!");
      },
      onSuccess: (data) => {
        toast.success(`Stripe account created successfully! Account ID: ${data.accountId}`);
      },
    });

    const handleCreateAccount = () => {
      mutate();
    };

    return (
      <div>
        <Button onClick={handleCreateAccount}>
          {'Create Stripe Account'}
        </Button>
      </div>      You, 13 hours ago • payment path working user stripe account creati…
    );
  };
export default StripeAccountSetup
```

Figure 7.15

7.8.2 Stripe Onboarding Link

This renders a button which when pressed uses the mutation to send the users stripe Id to the API route “generateStripeOnboardingLink” this will route the user to the stripe hosted onboarding. The operation of this is detailed in the stripe docs specifically for onboarding [7]

```
const StripeOnboardingLink = ({ stripePayoutId }: StripeLink) => {
  const router = useRouter();
  const { mutate, isLoading } = trpc.pay.generateStripeOnboardingLink.useMutation({
    onSuccess: (data) => {
      toast.message("created onboarding link, sending you to stripe!");
      router.push(data.link.url);
    },
    onError: (error) => {
      toast.error(
        `Failed to retrieve Stripe login link. Please try again!`
      );
    },
  });

  const handleAccessStripeDashboard = () => {
    mutate({ accountId: stripePayoutId });
  };

  return (
    <div>
      <Button onClick={handleAccessStripeDashboard} disabled={isLoading}>
        Access Stripe Onboarding
      </Button>
    </div>
  );
};
```

Figure 7.16

7.8.3 Stripe link

Finally stripe link will use mutation of stripe ID for the “generateStripeLoginLink” API route which will route the user to the stripe express dashboard login page.

```
const StripeLink = ({ stripePayoutId }: StripeLink) => {
  const router = useRouter();
  const { mutate, isLoading } = trpc.pay.generateStripeLoginLink.useMutation({
    onSuccess: (data) => {
      toast.message("sending you to stripe!");
      router.push(data.url);

      console.log(data);
    },
    onError: (error) => {
      toast.error(
        `Failed to retrieve Stripe login link. Please try again!`
      );
    },
  });
      You, 11 hours ago • Payout system working fixed user page to route
  const handleAccessStripeDashboard = () => {
    mutate({ accountId: stripePayoutId });
  };

  return (
    <div>
      <Button onClick={handleAccessStripeDashboard} disabled={isLoading}>
        Access Stripe Dashboard
      </Button>
    </div>
  );
};
```

Figure 7.17

7.9 View Replies

This component is used for displaying replies on work, this is achieved by,

7.9.1 Infinite query

Using the “getAllRepliesOnWork” route and tRPC infinite query hook to fetch the replies, This is paginated at 100 items per page.

```
trpc.reply.getAllRepliesOnWork.useInfiniteQuery(
  {
    query,
    limit: 100,
  },
  { getNextPageParam: (lastPage) => lastPage.nextPage }
);
```

Figure 7.18

7.9.2 Flatten pages and prepare data

I flatten the pages into a single array so that I can map it easier and then initialize an array named map which will be filled with replies or if data is missing null values.

```
const repliesView = queryResults?.pages.flatMap((page) => page.items);

let map: (Reply | null)[] = [];
if (repliesView && repliesView.length){
    map = repliesView;
} else if (!(repliesView && repliesView.length)){
    map = new Array<null>(100 ?? 100).fill(null);
}
```

Figure 7.19

7.10 Replies

Replies component manages the creation of reply posts by creating a mutation with the values from the form and hits the “createWorkreply” API route , the form is resolved by the Zod reolver ReplyDataValidator which I created. The form state is toggled to open and close the reply form. A point of note is on success I toggle the form and reset the form data then invalidate the “getAllRepliesOnWork” so that the new reply shows up.

```
onSuccess: () => {
    toggleForm();
    reset({ title: '', description: '' });
    utils.reply.getAllRepliesOnWork.invalidate();
},
```

Figure 7.20

7.11 Max Width Wrapper

This serves as a container for its child components and makes sure they are constrained to a maximum width and in the middle of the page.

```tsx
import { ReactNode } from "react";

const MaxWidthWrapper = ({
  className,
  children,
}: {
  className?: string;
  children: ReactNode;
}) => {
  const defaultClassName = 'mx-auto w-full max-w-screen-xl px-2.5 md:px-20';

  const combinedClassName = className ? `${defaultClassName} ${className}` : defaultClassName;

  return <div className={combinedClassName}>
    {children}
  </div>;
};

export default MaxWidthWrapper;
```

Figure 7.21

7.12 Providers

Providers component is important as it is crucial for setting up and providing the contexts for my tRPC and React Query features. The implementation for this came from [6]. And docs on this can be found on the TanStack docs[8].

7.12.1 Initializing query client

This component initializes the query client creating a new instance of the “QueryClient”, The mutations of React Query is managed by this object, its stored in the components state to allow the client instance to persist across renders.

```tsx
const [queryClient] = useState(() => new QueryClient());
```

Figure 7.22

7.12.2 Initializing tRPC client

This sets up the tRPC client which allows my application to make calls to the backend server defined by my tRPC routers. Batch link batches by requests minimizing the total requests sent to the server. The Fetch function includes credentials with every request.

```tsx
const [trpcClient] = useState(() =>
  trpc.createClient({
    links: [
      httpBatchLink({
        url: `${process.env.NEXT_PUBLIC_SERVER_URL}/api/trpc`,
        fetch(url, options) {
          return fetch(url, {
            ...options,
            credentials: "include",
          });
        },
      }),
    ],
  })
);
```

Figure 7.23

8 Pages

Now that I have shown my components I will now show the pages for my web application that bring it to life. A lot of the functionality happens in the components so some pages should be very straightforward by this point.
Note: nextjs14 routing requires all pages to be called “page.tsx” so the names of the pages are the folder they are in. More on the next.js routing can be found on the docs [9].

8.1 Layout

Layout.tsx is not a page but instead it is set up for the global layout structure in which the entire application is wrapped in. This includes the Navbar and the Providers.

8.2 App

App is the “page.tsx” in the app folder in which all other folder reside, essentially this is “/” route. On this page I use the “HomeMainBar” component with the query limit set to 4 along with general HTML / CSS to create my landing page.

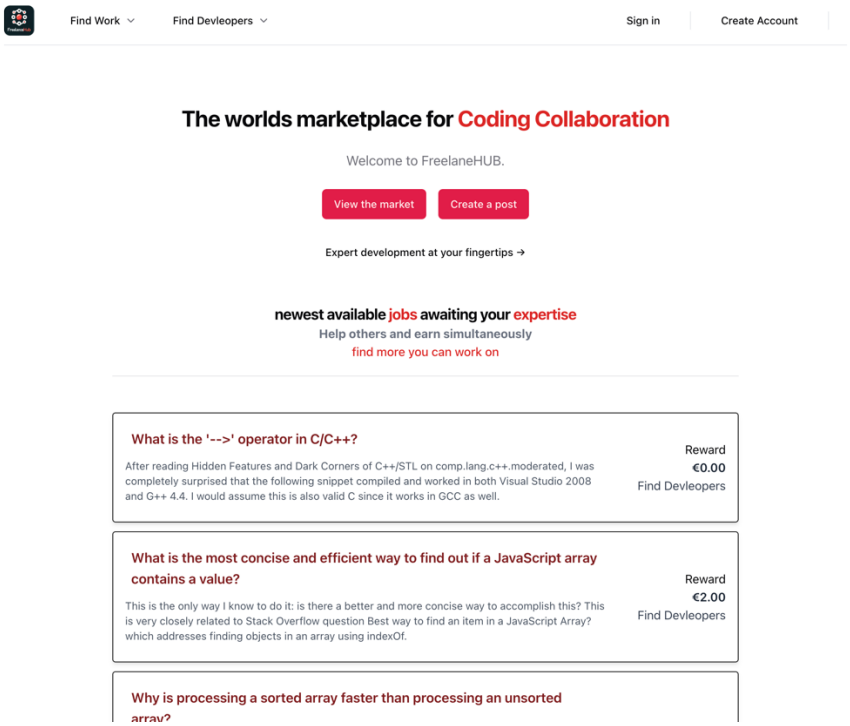


Figure 8.1

8.3 Market

Clicking on view the market routes you to the market page. Market is a very simple page that's makes use of the "HomeMainBar" component with the query set to 100. It allows users to see all the newest work posted.

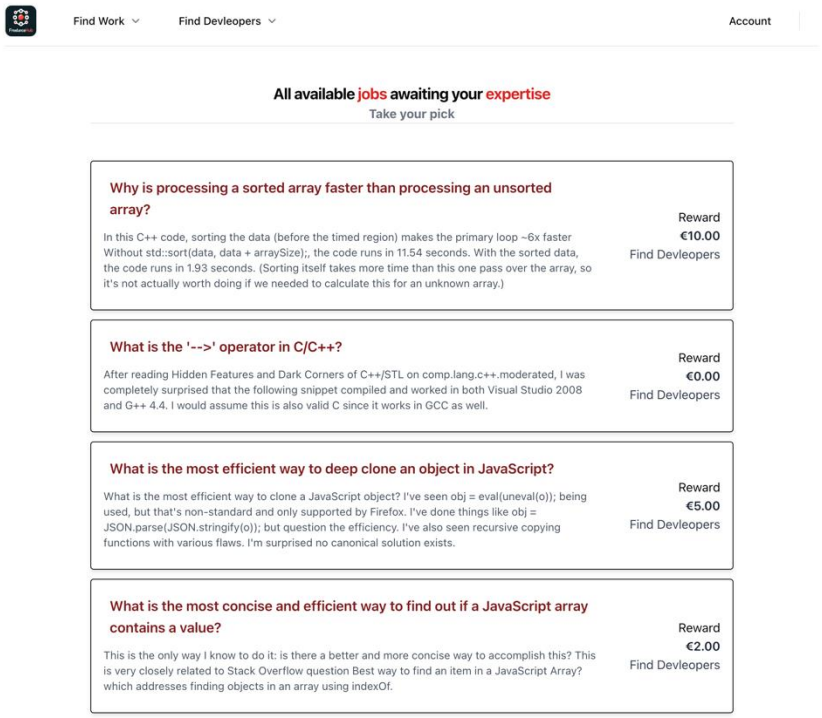


Figure 8.2

8.4 Work

The work page is essentially the same as the market except it just shows the newest jobs first as shown on the home main bar but with the query limit extended. Also between these three first pages the reusability of "HomeMainBar" can be seen as I use the same component with separate implementations.

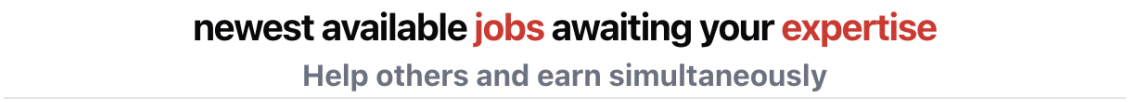


Figure 8.3

8.5 Work view

Clicking on any work item will bring you to the work view page. The structuring on this is an area of note as the [workId] represents a dynamic route parameter. This will allow me to access the route parameter workId which allows me to fetch information specific to the word item with that id.

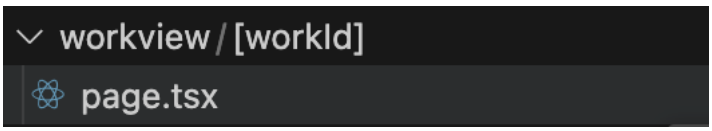


Figure 8.4

The work view page displays a view of the work item including its files its replies and an option to pay if applicable. It achieves this by first fetching the user details using the cookies and extracting the work Id from the params. Then fetches the work details from the work collection and de-structuring the first item in the docs into variable "workview".

```
const Page = async ({ params }: WorkViewPageProps) => {
  const nextCookies = cookies();
  const { user } = await getServerSideUser(nextCookies);

  const { workId } = params;

  const payload = await getPayloadClient();

  const { docs: work } = await payload.find({
    collection: "work",
    limit: 1,
    where: {
      id: {
        equals: workId,
      },
    },
  });

  const [workView] = work;
```

Figure 8.5

Then it checks If it is possible to extract the user if from the "WorkView" object and compares the server-side user to the user on the object to determine ownership. After that it checks if there is workfiles on the work to determine whether or not to show the work container and displays all this information conditionally.

```
let userId: string | undefined;

if (workView.user && typeof workView.user !== "string") {
  userId = workView.user.id;
}

function checkOwnership(
  userId: string | undefined,
  currentUserId: string | undefined
) {
  return userId === currentUserId;
}

const isOwner = checkOwnership(userId, user?.id);

function hasWorkFiles(
  workFiles: (string | WorkFile)[] | null | undefined
): boolean {
  return !!workFiles && workFiles.length > 0;
}
```

Figure 8.6

If there are work files I pass the file url of each to the “FileViewer” component. Also if the work is not approved or rejected I render the “WorkPayment” component allowing them to pay to post it. The “Replies” component is rendered to allow the user to add a reply to the work and I use the “ViewReplies” under that to display all the replies on the work.

```
const WorkFilesDisplay = (work: (string | WorkFile)[]): JSX.Element => {
  return (
    <>
      <div className="max-h-[800px] overflow-y-auto border ■border-gray-300 rounded-lg p-2">
        {work.map((file, index) => {
          if (typeof file === "object" && "url" in file) {
            return (
              <div key={index} className="pt-3">
                <FileViewer fileUrl={file.url} fileName={file.filename} />
              </div>
            );
          }
          return null;
        })}
      </div>
    </>
  );
};

if (!workView) {
  return notFound();
}

return (
  <MaxWidthWrapper>
    {isOwner &&
    workView.approved !== "approved" &&
    workView.approved !== "rejected" ? (
      <WorkPayment workId={workId}></WorkPayment>
    ) : null}
    <div className="relative w-full pt-5">
      <WorkListings workItem={workView} index={1} key={`workItem-${1}`} />
    </div>
    <div className="pb-3">
      {hasWorkFiles(workView.workFiles) &&
        WorkFilesDisplay(workView.workFiles || [])}
    </div>

    <Replies params={{ userId: userId || "", workId: workId }}></Replies>
    <ViewReplies query={{ workId: workId }}></ViewReplies>
  </MaxWidthWrapper>
);
};
```

Figure 8.7

Up to this point I haven't showed much of the HTML as it Is mainly the same everywhere but here is a nice view of many components at work coming together to make my page.

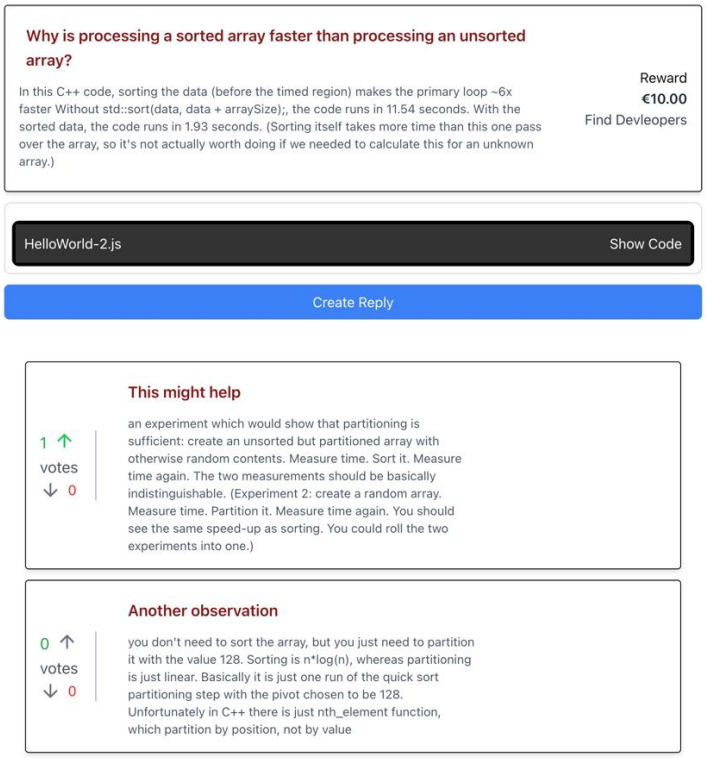


Figure 8.7

8.6 User

The user page works similarly to the "WorkView" page in the sense that it also takes the "userId" from the params to display the user data. Again I fetch the user details from the suers collection depending on the "userId" and extract the first user from the docs and access the data from there. Some HTML, conditional rendering of the stripe components and an implementation of the "UserWork" component and I have my user page.

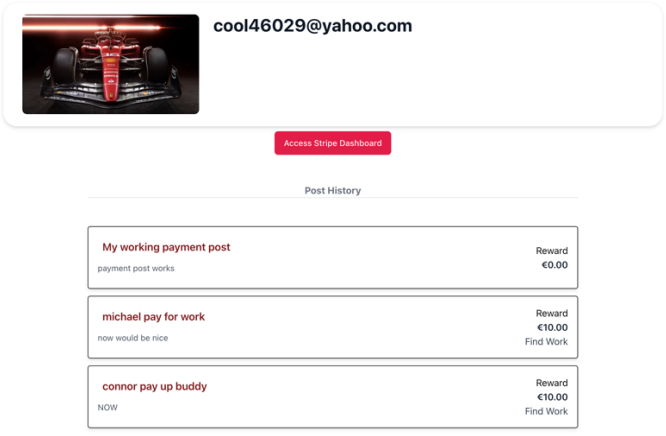


Figure 8.8

```
return (
  <MaxWidthWrapper>
    <div className=" mt-10 z-10 bg-white shadow-md border border-gray-100 rounded-3xl">
      <div className="px-4 py-5 mx-auto max-w-7xl sm:px-6 lg:px-8">
        <div className="flex items-start space-x-6">
          {" "}
          {imageUrl && <Image
            src={imageUrl}
            alt="User Profile"
            width={300}
            height={300}
            className="flex-shrink-0 rounded-lg"
          />}
          <div className="flex flex-col space-y-4">
            {" "}
            <h1 className="text-2xl font-bold text-gray-900 sm:text-3xl">
              {user.email}
            </h1>
            <p className="text-lg text-gray-500">{user.biography}</p>
          </div>
        </div>
      </div>
    </div>
    <div className="flex justify-center pt-2">
      {!user.stripePayoutId ? (
        <StripeAccountSetup />
      ) : user.onboardedStripe === "verified" ? (
        <StripeLink stripePayoutId={user.stripePayoutId}></StripeLink>
      ) : (
        <StripeOnboardingLink
          stripePayoutId={user.stripePayoutId}
        ></StripeOnboardingLink>
      )}
    </div>
    <UserWork
      title=""
      sub={"Post History"}
      query={{ sort: "desc", limit: 4, userId: userId }}
    />
  </MaxWidthWrapper>
);
};
```

Figure 8.9

8.7 Posting

The posting page is the page where I send the user to create posts, this is another form as we have looked at many times with the same implementation of using the mutation to use an API route only this time I use “createWorkPosting”. Now there Is another way to create a work and that’s from the payload dashboard which I haven’t shown till now so I think now is a good time to take a look at that.

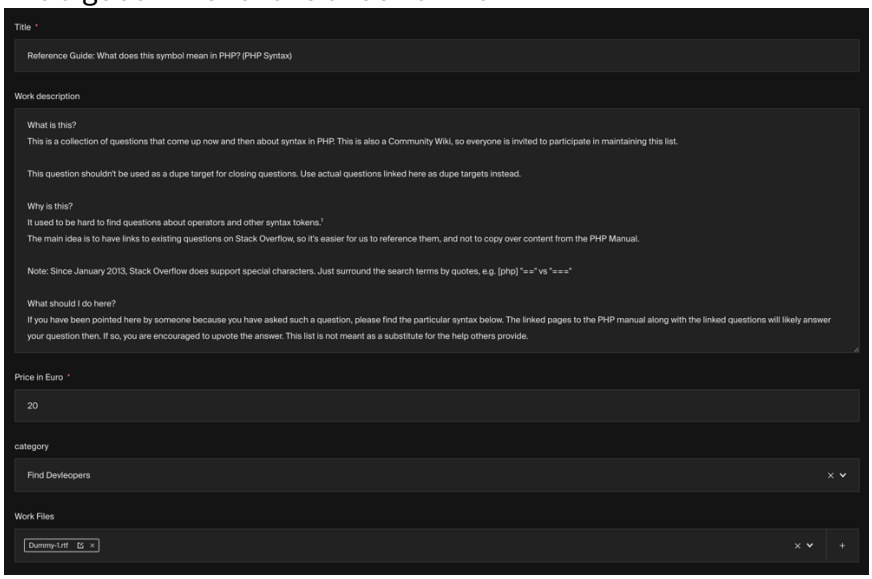


Figure 8.10

Here we can add everything we want and really everything could go through the admin dashboard but that would not be good design but realistically the frontend is just mocking the backend implementation

9 Auth pages

As demonstrated in the implementation by JoshTriedCoding [6]

9.1 Sign up

Here I will show the implementation of our sign up and all relevant and supporting implementations to make our sign up work.

9.1.1 Mutation

For the sign up a mutation is being set up using tRPC through the auth router to create a user.

9.1.1.1 Overview

Methods are defined for Success and Error scenarios,

9.1.1.2 onError

if the email is already in use a toast error message is displayed. Another error condition is defined if there is a validation error and finally a general error message for uncaught issues.

9.1.1.3 onSuccess

On success a toast is displayed containing the email used in a message stating a verification email has been sent to the email, then the router navigates the user to the check email for verification page which indicates an email has been sent to the email used. The email displayed is the de-structured parameter sentToEmail.

```
const { mutate, isLoading } = trpc.auth.createPayloadUser.useMutation({
  onError: (err) => {
    if (err.data?.code === "CONFLICT") {
      toast.error("This email is already in use. Sign in?");
    }

    //technially not needed but good for api security
    if (err instanceof ZodError) {
      toast.error(err.issues[0].message);
    }

    toast.error("somethings happened please try again");
  },

  onSuccess: ({ sentToEmail }) => {
    toast.success(`Email has been sent to ${sentToEmail}`);
    router.push("verify-email?to=" + sentToEmail);
  },
});
```

Figure 9.1

9.1.2 On Submit

This function takes an object containing the email and password the object is typed with TAuthCredentialsValidator indicating that the contents of the object conform to the validation schema which I have defined using Zod. It then triggers the mutate function I showed above.

```
const onSubmit = ({ email, password }: TAuthCredentialsValidator) => {
  mutate({ email, password });
};
```

Figure 9.2

9.1.3 AuthCredentialsValidator

Using Zod I define the AuthCredentialsValidator by defining a z.object which contains the expected structure of the data. The email field is expected to be a string and the email method attached will automatically check that the string is a valid email. The password field is a string that must contain at least eight characters, else Zod will create an error message which I specified. Then I export the type using typescripts type interface capabilities with the Zod infer method to create a Typescript type based on our Zod schema. TAuthCredentialsValidator now represents an object with an email and a password with the minimum length I set.

```
import { z } from "zod";

export const AuthCredentialsValidator = z.object({
  email: z.string().email(),
  password: z
    .string()
    .min(8, { message: "Password Must be Atleast 8 Charachters long" }),
});

export type TAuthCredentialsValidator = z.infer<
  typeof AuthCredentialsValidator
>;
```

Figure 9.3

9.1.4 React Form Hook

I utilize the register function from react hook form to manage the inputs, and the formstate{errors} to access the properties of the forms state.

9.1.5 createPayloadUser

Now we look at the server-side function the auth router createPayloadUser which handles the creation of users.

9.1.5.1 Input and Mutation

The .input indicates that the functions input should be validated by the AuthCredentialsValidator the mutation then takes the input object from which we de-structure the email and password from.

9.1.5.2 payload.find

After getting an instance of the payload client so that we can communicate with the database we use payload.find to query our database in the users collection for an user that has the same email as the input. This will return an object that contains many properties, the one we are interested in is the docs, which contains an array of document objects that satisfied our where clause, in this case users with the same email.

9.1.5.3 TRPCError

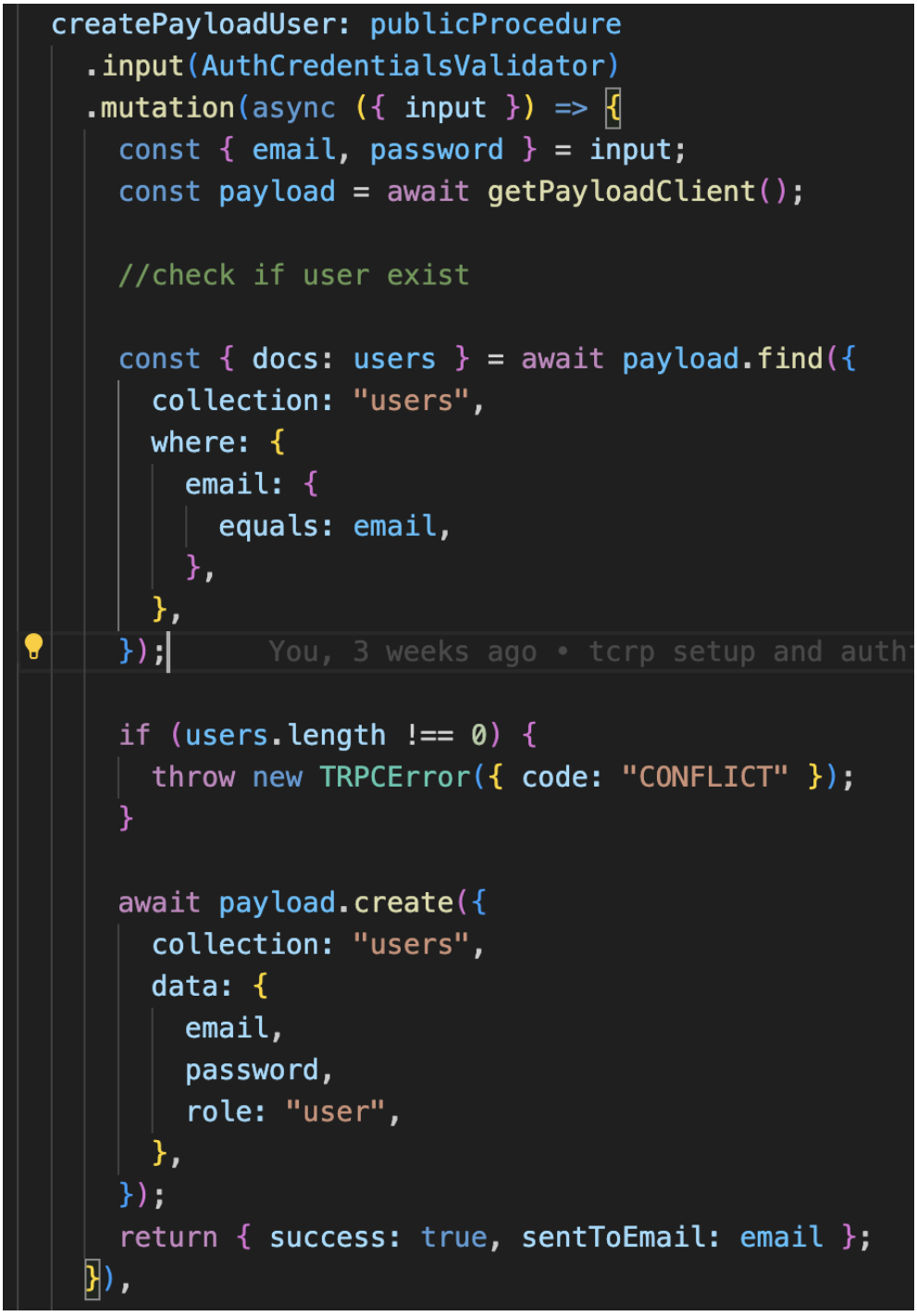
If the users.length (the array containing the users with that email) is not empty we throw an error using TRPCError with the code “CONFLICT”.

9.1.5.4 payload.create

Having checked no user exists with this email we use the payload.create method to create a new document in the database, we specify the collection where the document will be inserted and the data that will be stored in the new document, these fields are defined in our collections schema which we will take a look at later on.

9.1.5.5 Return

On success we return an object containing success as true and the sentToEmail we spoke about earlier as the email from the input.



```
createPayloadUser: publicProcedure
  .input(AuthCredentialsValidator)
  .mutation(async ({ input }) => {
    const { email, password } = input;
    const payload = await getPayloadClient();

    //check if user exist

    const { docs: users } = await payload.find({
      collection: "users",
      where: {
        email: {
          equals: email,
        },
      },
    });      You, 3 weeks ago • tcrp setup and auth

    if (users.length !== 0) {
      throw new TRPCError({ code: "CONFLICT" });
    }

    await payload.create({
      collection: "users",
      data: {
        email,
        password,
        role: "user",
      },
    });
    return { success: true, sentToEmail: email };
  }),
```

Figure 9.4

9.2 Verify email

The verify email page is designed for email verification it displays its content dynamically depending on whether the token is present in the search parameters of the url, if a token is present and a string the verify email component is rendered and the token is passed as a prop, if there is no token a check email UI is displayed.

9.2.1 Verify email component

This component is designed to handle the user verification based on the token passed it interacts with the backend to the “verifyEmail” route, if there is an error in the query an error is displayed. If successful a success message and image are shown and a link to the log in page.

9.2.2 Verify email procedure

This procedure takes an input of an object containing a token, it then queries the database for a user with that token and if it exists it will set _verified to true, if “isVerfied” is null it will throw a new tRPC error if not it will return the object containing success as true.

```
verifyEmail: publicProcedure
  .input(z.object({ token: z.string() }))
  .query(async ({ input }) => {
    const { token } = input;

    const payload = await getPayloadClient();

    const isVerified = payload.verifyEmail({
      collection: "users",
      token,
    });

    if (!isVerified) {
      throw new TRPCError({ code: "UNAUTHORIZED" });
    }

    return { success: true };
  }),
```

Figure 9.5

9.3 Sign in

Sign in is a user auth page where users can sign in, it contains all functionality previously explained and it uses the “signIn” procedure with a mutation of the form data.

9.3.1 Sign in procedure

This takes an input of AuthCredentialsValidator which will validate the email and password the mutation will modify the data state on the server the uses the payload login method with the user credentials and the response object to sign the user in.

```
signIn: publicProcedure
  .input(AuthCredentialsValidator)
  .mutation(async ({ input, ctx}) => {
    const { email, password } = input;
    const {res} = ctx

    const payload = await getPayloadClient();

    try {
      await payload.login({
        collection: "users",
        data: {
          email,
          password,
        },
        res,
      });
      return {success: true}
    } catch (err) {
      throw new TRPCError({code: "UNAUTHORIZED"})
    }
  }),
```

Figure 9.6

10 tRPC

Here I will show my procedures I created across my various routers,

10.1 Setup

Middleware, index, tRPC, and client manage my setup and configuration of the tRPC framework, client side API consumption, and middleware functionality.

10.1.1 Middleware

Its functionality enriches the context and check user permissions. As demonstrated in the implementation by JoshTriedCoding [6].

10.1.2 Client

Client configures the client-side characteristics of the tRPC to allow for communication with the backend using the procedures defined in the routers. As demonstrated in the implementation by JoshTriedCoding [6].

10.1.3 tRPC

Initializez the tRPC setting global configs and using middleware defined in “middleware.ts”. As demonstrated in the implementation by JoshTriedCoding [6].

10.2 Routers

10.2.1 Index

Combines all routers into a single router.

```
auth: authRouter,
work: workRouter,
reply: replyRouter,
pay: paymentsRouter,
```

Figure 10.1

10.2.1.1 Get all from marketplace

This procedure will get all work items on the marketplace that are approved, it contains input validation for the object which should contain a limit a cursor number and a query which satisfies the query validator, then to execute the query I de-structure the cursor

and the query from the input and further de-structure the sort and limit and query options from the query. Then create an instance of the payload client and create the parsed query option object where the “quertyOpts” is transformed into a suitable object for querying. Then the Payload find method will fetch the docs form the work collection using the query options, pagination handled through limit and page calculated from cursor. The results and pagination data are returned.

```
getAllWorkForMarketplace: publicProcedure
  .input(
    z.object({
      limit: z.number().min(1).max(100),
      cursor: z.number().nullish(),
      query: QueryValidator,
    })
  )
  .query(async ({ input }) => {
    const { query, cursor } = input;
    const { sort, limit, ...queryOpts } = query;

    const payload = await getPayloadClient();

    const parsedQerOpts: Record<string, { equals: string }> = {};

    Object.entries(queryOpts).forEach(([key, value]) => {
      parsedQerOpts[key] = {
        equals: value,
      };
    });

    const page = cursor || 1;

    const {
      docs: items,
      hasNextPage,
      nextPage,
    } = await payload.find({
      collection: "work",
      where: {
        approved: {
          equals: "approved",
        },
      },
      sort,
      depth: 1,      You, 3 weeks ago • profile pages and workview underw
      limit,
      page,
    });
    return {
      items,
      nextPage: hasNextPage ? nextPage : null,
    };
```

Figure 10.2

10.2.1.2 Get all work from user

This is essentially exactly the same as get all from marketplace but I alter the query method to only find work where the user on the work is equal to the user Id.

```
getAllWorkFromUser: publicProcedure
.input(
  z.object({
    limit: z.number().min(1).max(100),
    cursor: z.number().nullish(),
    query: UserWorkQueryValidator,
  })
)
.query(async ({ input }) => {
  const { query, cursor} = input;
  const { sort, limit, userId, ...queryOpts } = query;

  const payload = await getPayloadClient();

  const parsedQerOpts: Record<string, { equals: string }> = {};

  Object.entries(queryOpts).forEach(([key, value]) => {
    parsedQerOpts[key] = {
      equals: value,
    };
  });

  const page = cursor || 1;

  const {
    docs: items,
    hasNextPage,
    nextPage,
  } = await payload.find({
    collection: "work",
    where: {
      user:{
        equals: userId
      }
    },
    sort,
    depth: 1,
    limit,
    page,
  });
  return {
    items,
    nextPage: hasNextPage ? nextPage : null,
  };
})
```

Figure 10.3

10.2.2 Reply Router

Router to handle all reply procedures.

10.2.2.1 Create work reply

This procedure exactly the same implementation a seen many times in the app, it takes the validated input and a mutation in which Initialize the payload client and user Payload create to create a reply in the replies collection with the data de-structured from the input.

```
createWorkreply: publicProcedure
  .input(ReplyDataValidator)
  .mutation(async ({ input }) => {
    const { title, description, user, workFiles, work } = input;

    const payload = await getPayloadClient();

    const newReply = await payload.create({
      collection: "replies",
      data: {
        title,
        description,
        user: user,
        workFiles,
        work,
      },
    });
    You, last week • added votes to replies
    return newReply;
  }),
```

Figure 10.4

10.2.2.2 Get all replies on work

I won't show get all replies on work as yet again it is exactly the same as get all work on user but I search the replies collection for replies where the work is equal to the work Id.

10.2.2.2.1 Add / remove / get , upvotes / downvotes, on replies

While the first part of this procedure is more of what I have already shown here is something interesting of note, I use payload update to update the number of upvotes on the reply by updating the reply where the id matched the reply id and the new votes is the old votes +1. Inversely for the remove its -1. This is duplicated for both upvotes and downvotes. The get procedures are just the first part of the other procedure a simple query we have look at many times over.

```
    const newvote = (thisReply.upvotes as number) + 1;

    return await payload.update({
      collection: "replies",
      where: {
        id: {
          equals: replyId,
        },
      },
      data: {
        upvotes: newvote,
      },
    });
  }),      You, last week • voting system finished and cart persist
```

Figure 10.5

```
  addUpvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .mutation(async ({ input }) => {…
    }),

  removeUpvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .mutation(async ({ input }) => {…
    }),

  getUpvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .query(async ({ input }) => {…
    }),

  addDownvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .mutation(async ({ input }) => {…
    }),

  removeDownvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .mutation(async ({ input }) => {…
    }),

  getDownvotesOnReplies: publicProcedure
    .input(z.object({ replyId: z.string() }))
    .query(async ({ input }) => {…
    }),
```

Figure 10.6

10.2.3 Work router

The only item in the work router is the create work posting where once again we use the payload create to create a work item, the input is validated by the PostDataValidator and I create a new work in the work collection with the data from the input with approved set to unverified.

```
  createWorkPosting: privateProcedure
    .input(PostDataValidator)
    .mutation(async ({ input, ctx }) => {
      const { title, description, workFiles } = input;
      const { user } = ctx;

      const payload = await getPayloadClient();

      const newWork = await payload.create({
        collection: "work",
        data: {
          title,
          description,
          workFiles,
          user: user.id,
          approved: "unverified",
          price: 0,
        },
      });

      return newWork;
    }),
});
```

Figure 10.7

10.2.4 Stripe router

The stripe router contains all procedures related to setting up the users for stripe express dashboard and onboarding them and allowing them to log in so that I can give pay-outs to the users from the rewards. The implementation for all of this is extremely well documented in many languages on the stripe documentation [10] . The implementation of this was adapted from the docs, except where otherwise referenced. NB: These are all private procedures and contain the context, you must be logged in to use these.

10.2.4.1 Create stripe account

Create stripe account uses the stripe client to create an account, I specify the data, type of account express dashboard, the country, the email, and the capabilities (important to request transfers so that I can transfer funds to the account). After the account is created I update the users stripe pay-out id field with the id from stripe. Note: the user is taken from context as it's a private procedure.

```
createStripeAccount: privateProcedure.mutation(async ({ ctx }) => {
  const { user } = ctx;
  const userId = user.id;

  try {
    const account = await stripe.accounts.create({
      type: "express",
      country: "IE",
      email: user.email,
      capabilities: {
        card_payments: { requested: true },
        transfers: { requested: true },
      },
    });

    const payloadClient = await getPayloadClient();

    await payloadClient.update({
      collection: "users",
      id: userId,
      data: {
        stripePayoutId: account.id,
      },
    });

    return { accountId: account.id };
  } catch (error) {
    console.error("Failed to create Stripe account:", error);
    throw new TRPCError({
      code: "INTERNAL_SERVER_ERROR",
      message: "Failed to create Stripe account",
    });
  }
}),
```

Figure 10.8

10.2.4.2 Generate stripe onboarding link

This procedure uses the stripe client account links create procedure to use the stripe account Id to create an onboarding link for the user, the return URL is where the user will be directed to after success, the refresh URL is where they will redirected to on cancel. Then the link is created and the user is redirected to complete the onboarding from stripe. On success the users onboarded status is updated in the database.

```
  generateStripeOnboardingLink: privateProcedure
  .input(z.object({ accountId: z.string() }))
  .mutation(async ({ ctx, input }) => {
    const { accountId } = input;

    const accountLink = await stripe.accountLinks.create({
      account: accountId,
      refresh_url: `${process.env.NEXT_PUBLIC_SERVER_URL}/user/${ctx.user.id}`,
      return_url: `${process.env.NEXT_PUBLIC_SERVER_URL}/user/${ctx.user.id}`,
      type: 'account_onboarding',
    });

    const payloadClient = await getPayloadClient();


    await payloadClient.update({
      collection: "users",
      id: ctx.user.id,
      data: {
        onboardedStripe: "verified",
      },
    });
    return{link: accountLink}
  }),
```

Figure 10.9

10.2.4.3 Generate stripe login link

This procedure will create a login link using the stripe client account create login link procedure for the user to stripe to access their dashboard using the account Id on this is will return the link and the front end routes the user to the login.

```
generateStripeLoginLink: privateProcedure
  .input(z.object({ accountId: z.string() }))
  .mutation(async ({ ctx, input }) => {
    const { accountId } = input;
    try {
      const loginLink = await stripe.accounts.createLoginLink(accountId);
      return { url: loginLink.url };
    } catch (error) {
      console.error("Failed to generate Stripe login link:", error);
      throw new TRPCError({
        code: "INTERNAL_SERVER_ERROR",
        message: "Failed to generate Stripe login link",
      });
    }
  }),
```

Figure 10.10

10.2.4.4 Initiate session

This creates a session using the stripe client with the line item which will be the work to be paid. This is achieved by using the stripe client checkout sessions create, it takes a success URL a cancel url payment types, meta data and line items . The implementation of this session was adapted from joshTriedCoding[6] and made to suit my project.

```
const workOrder = await payloadClient.create({
  collection: "workOrder",
  data: {
    _isPaid: false,
    work: workItem.id,
    user: user.id,
  },
});

const line_items: Stripe.Checkout.SessionCreateParams.LineItem[] = [];

if (workItem) {
  line_items.push({
    price: workItem.priceId!,
    quantity: 1,
  });
}

try {
  const stripeSession = await stripe.checkout.sessions.create({
    success_url: `${process.env.NEXT_PUBLIC_SERVER_URL}/ordercomplete?orderId=${workOrder.id}`,
    cancel_url: `${process.env.NEXT_PUBLIC_SERVER_URL}/workpayment/${workId}`,
    payment_method_types: ["card"],
    mode: "payment",
    metadata: {
      userId: user.id,     You, yesterday • some code cleanup and payment flow underway
      orderId: workOrder.id,
    },
    line_items,
  });
  return { url: stripeSession.url };
} catch (error) {
  console.error("Failed to create Stripe session:", error);
  return { url: null };
}
}),
```

Figure 10.11

11 Other files

11.1 Payload utils

Retrieves the authenticated user details from the server side API using the JWT token stored in cookies.

11.2 Validators

The various validator are Zod objects with Zod conditions as shown In the .input of every procedure that does not use a validator.

11.3 Server

Runs an express server that handles the API requests and serves the Next.js application.

11.4 Use auth hook

Script to sign user out

11.5 Payload config

Payload cms instance.

Refrences

[1]Next.js, https://nextjs.org/docs
[2]tRPC, https://trpc.io/docs
[3]Payload, What is Payload, https://payloadcms.com/docs/getting-started/what-is-payload
[4]tailwindcss, Get started with Tailwind CSS, https://tailwindcss.com/docs/installation
[5]npm, React Syntax Highlighter, https://www.npmjs.com/package/react-syntax-highlighter
[6] JoshTriedCoding, https://www.youtube.com/@joshtriedcoding
[7]Stripe, Strip hosted onboarding for Custom accounts, https://docs.stripe.com/connect/custom/hosted-onboarding?lang=java
[8]TanStack, QueryClientProvider, https://tanstack.com/query/v4/docs/framework/react/reference/QueryClientProvider
[9]Next.js, Roles of Folders and Files, https://nextjs.org/docs/app/building-your-application/routing#roles-of-folders-and-files
[10]Stripe docs and quickstart, https://docs.stripe.com/checkout/quickstart
https://codingpr.com/stripe-and-react-typescript/