Challenge: Jumper

Key points

- I use zod with openApi to type safe my Api endpoint.
- I use `@hey-api/openapi-ts` to generate queries to call my api endpoints and to strongly type my frontend.
- Users a store in a simple sqlitedb using prisma as ORM.
- I rely on TanStack query to handle UI State and error in the front end.
- I use wagmi and viem for connectivity with the blockchain

Introduction

The exercise consists of a simple data retrieval exercise directly from the blockchain (Balance, Address) and also through an API (Express JS) to retrieve datas from the blockchain as well and from a data source. There are many criteria mentioned in the expectations, so I chose to apply myself to meet as many of them as possible.

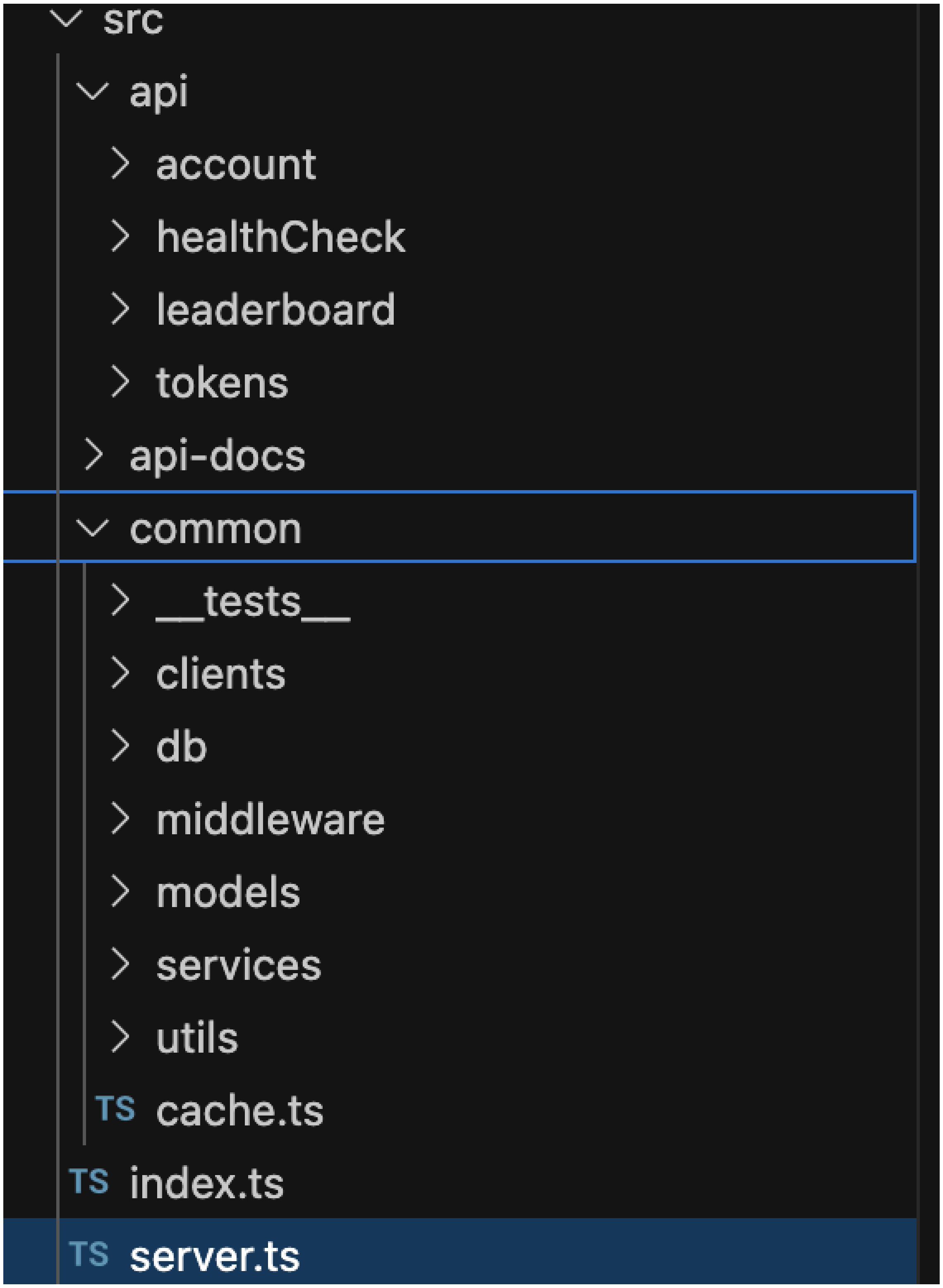
There are several technologies used with which I am not familiar or with which I have only worked recently on a limited basis (MUI, OpenAPI, ...), so I had to spend some time reading the documentation to use them.

This document is not an exhaustive list of all my work but rather a highlight of the points I consider important or the choices I was able to make.

Overall consideration

So as a start I decided to go with PNPM and Biome as package manager and linter/formatter just to use a more recent tech stack.

Regarding the project architecture, I reused the existing structure as much as possible. The endpoints are therefore organized by features and in the common folder I have my different domains. To summarize, my controllers call my services which in turn call either the database or the Alchemy client to retrieve tokens and contain small pieces of logic when necessary.



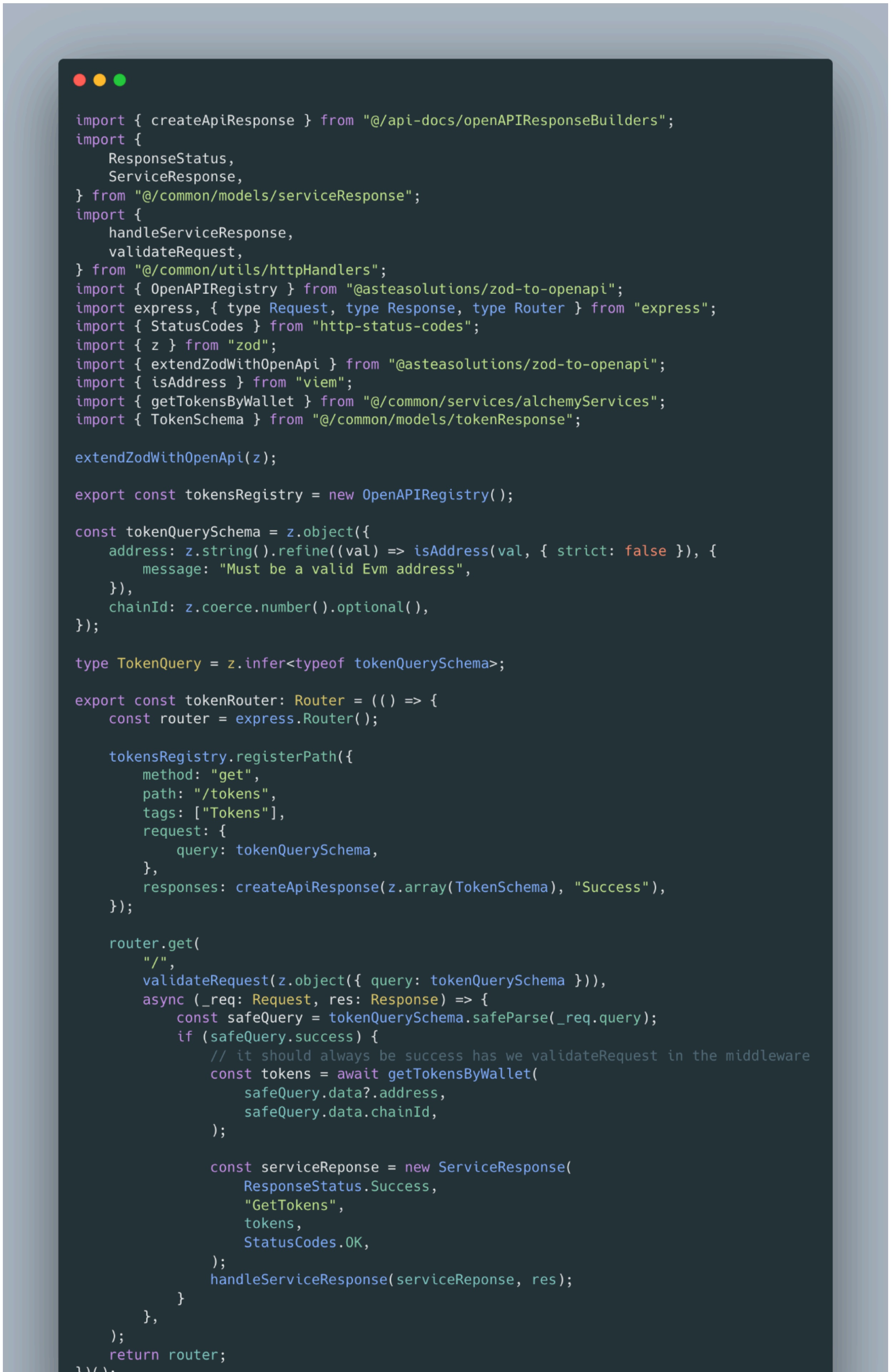
```
src
  api
    account
    healthCheck
    leaderboard
    tokens
  api-docs
  common
    __tests__
    clients
    db
    middleware
    models
    services
    utils
    cache.ts
  index.ts
  server.ts
```

Api endpoint

Then I created my 3 API endpoints: /account for account creation, /tokens for retrieving tokens from an address, and /leaderboard which will be used to retrieve the leaderboard.

/tokens

I understood that I could use Zod in conjunction with the OpenAPI to type my API inputs and outputs (which will be very useful later). So I created my Zod schemas. I also saw that there was a "**validateRequest**" middleware that allowed validating the request beforehand. I wasn't able to achieve strong typing after using this middleware, which is why I do another schema parse to ensure proper typing.

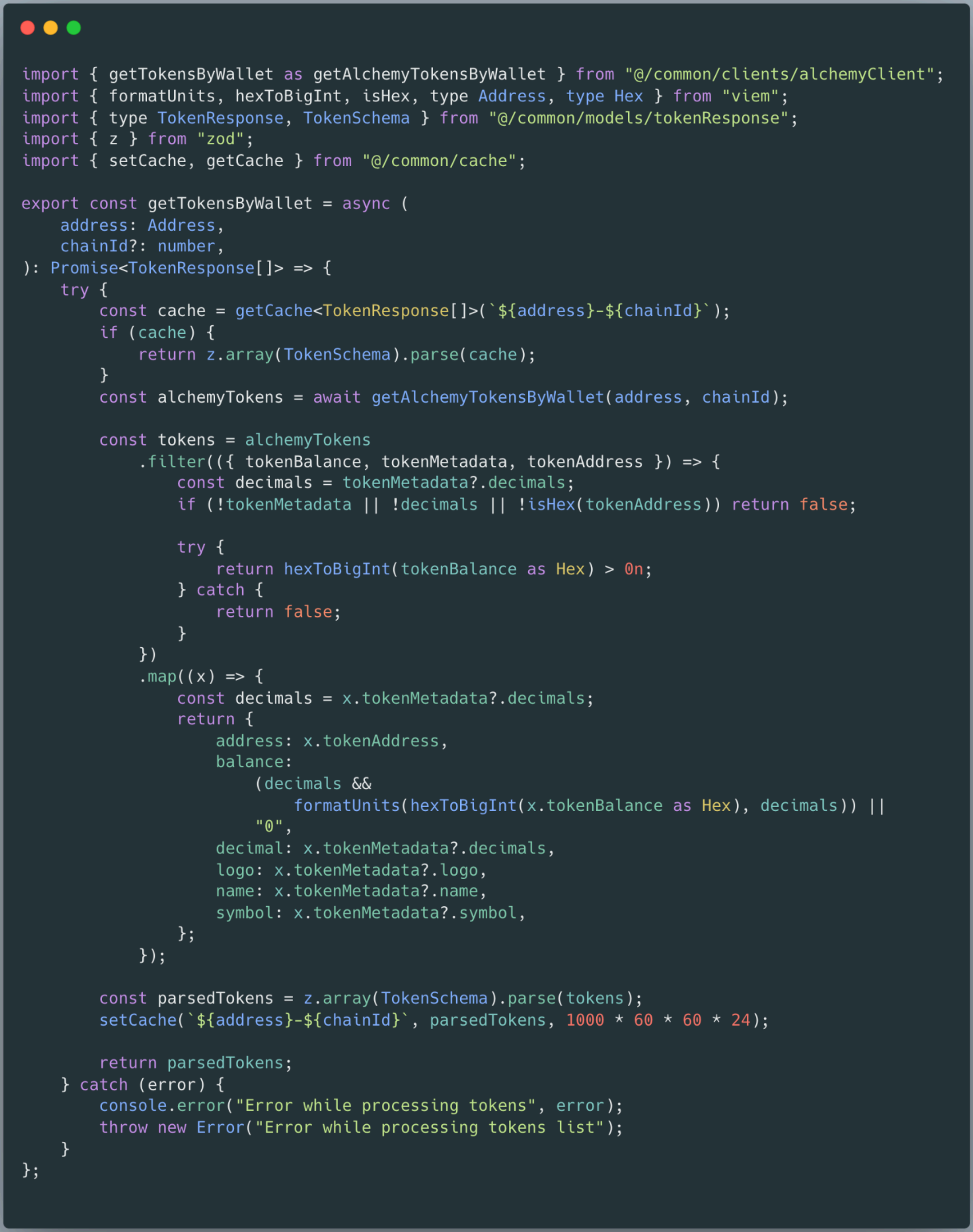


```typescript
import { createApiResponse } from "@/api-docs/openAPIResponseBuilders";
import {
    ResponseStatus,
    ServiceResponse,
} from "@/common/models/serviceResponse";
import {
    handleServiceResponse,
    validateRequest,
} from "@/common/utils/httpHandlers";
import { OpenAPIRegistry } from "@asteasolutions/zod-to-openapi";
import express, { type Request, type Response, type Router } from "express";
import { StatusCodes } from "http-status-codes";
import { z } from "zod";
import { extendZodWithOpenApi } from "@asteasolutions/zod-to-openapi";
import { isAddress } from "viem";
import { getTokensByWallet } from "@/common/services/alchemyServices";
import { TokenSchema } from "@/common/models/tokenResponse";

extendZodWithOpenApi(z);

export const tokensRegistry = new OpenAPIRegistry();

const tokenQuerySchema = z.object({
    address: z.string().refine((val) => isAddress(val, { strict: false }), {
        message: "Must be a valid Evm address",
    }),
    chainId: z.coerce.number().optional(),
});

type TokenQuery = z.infer<typeof tokenQuerySchema>;

export const tokenRouter: Router = (() => {
    const router = express.Router();

    tokensRegistry.registerPath({
        method: "get",
        path: "/tokens",
        tags: ["Tokens"],
        request: {
            query: tokenQuerySchema,
        },
        responses: createApiResponse(z.array(TokenSchema), "Success"),
    });

    router.get(
        "/",
        validateRequest(z.object({ query: tokenQuerySchema })),
        async (_req: Request, res: Response) => {
            const safeQuery = tokenQuerySchema.safeParse(_req.query);
            if (safeQuery.success) {
                // it should always be success has we validateRequest in the middleware
                const tokens = await getTokensByWallet(
                    safeQuery.data?.address,
                    safeQuery.data.chainId,
                );

                const serviceReponse = new ServiceResponse(
                    ResponseStatus.Success,
                    "GetTokens",
                    tokens,
                    StatusCodes.OK,
                );
                handleServiceResponse(serviceReponse, res);
            }
        },
    );
    return router;
})();
```

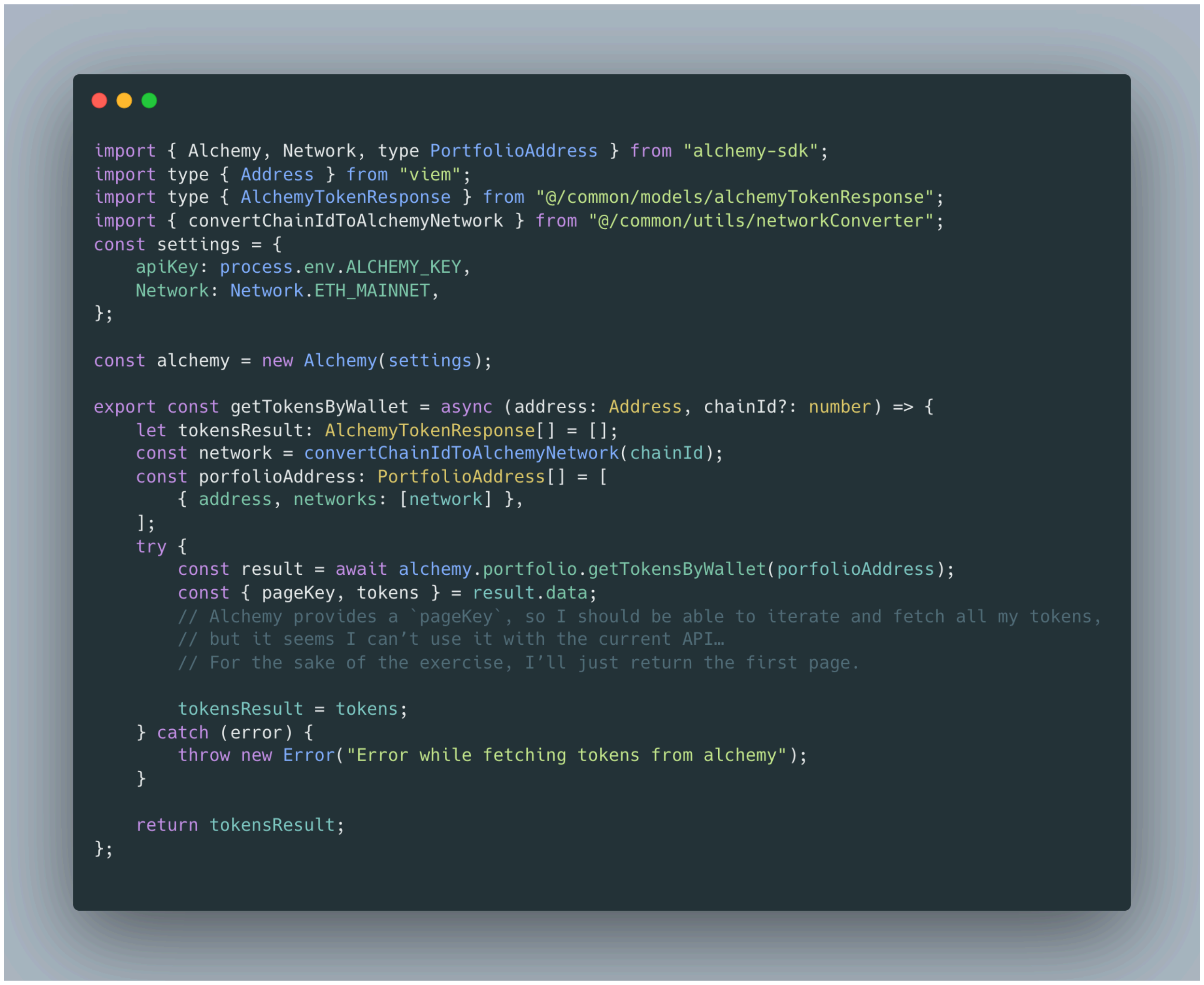
For the /token endpoint I created an Alchemy service which in turn calls the Alchemy client to retrieve tokens from a given account on a given blockchain.

I had a small issue with the Alchemy API. Although I have the pageKey parameter to be able to iterate and retrieve data, I couldn't implement it because the SDK simply doesn't offer the option to pass pageKey. So I considered that we would only retrieve the first page.

In my service I filter the tokens a bit to remove tokens with a balance of 0 and with a somewhat odd format (no decimals, incorrect address, etc.). I also add a cache which prevents the "too many requests" error with the Alchemy SDK, and the key is by address and by blockchain.



```typescript
import { getTokensByWallet as getAlchemyTokensByWallet } from "@/common/clients/alchemyClient";
import { formatUnits, hexToBigInt, isHex, type Address, type Hex } from "viem";
import { type TokenResponse, TokenSchema } from "@/common/models/tokenResponse";
import { z } from "zod";
import { setCache, getCache } from "@/common/cache";

export const getTokensByWallet = async (
    address: Address,
    chainId?: number,
): Promise<TokenResponse[]> => {
    try {
        const cache = getCache<TokenResponse[]>(`${address}-${chainId}`);
        if (cache) {
            return z.array(TokenSchema).parse(cache);
        }
        const alchemyTokens = await getAlchemyTokensByWallet(address, chainId);

        const tokens = alchemyTokens
            .filter(({ tokenBalance, tokenMetadata, tokenAddress }) => {
                const decimals = tokenMetadata?.decimals;
                if (!tokenMetadata || !decimals || !isHex(tokenAddress)) return false;

                try {
                    return hexToBigInt(tokenBalance as Hex) > 0n;
                } catch {
                    return false;
                }
            })
            .map((x) => {
                const decimals = x.tokenMetadata?.decimals;
                return {
                    address: x.tokenAddress,
                    balance:
                        (decimals &&
                            formatUnits(hexToBigInt(x.tokenBalance as Hex), decimals)) ||
                        "0",
                    decimal: x.tokenMetadata?.decimals,
                    logo: x.tokenMetadata?.logo,
                    name: x.tokenMetadata?.name,
                    symbol: x.tokenMetadata?.symbol,
                };
            });

        const parsedTokens = z.array(TokenSchema).parse(tokens);
        setCache(`${address}-${chainId}`, parsedTokens, 1000 * 60 * 60 * 24);

        return parsedTokens;
    } catch (error) {
        console.error("Error while processing tokens", error);
        throw new Error("Error while processing tokens list");
    }
};
```



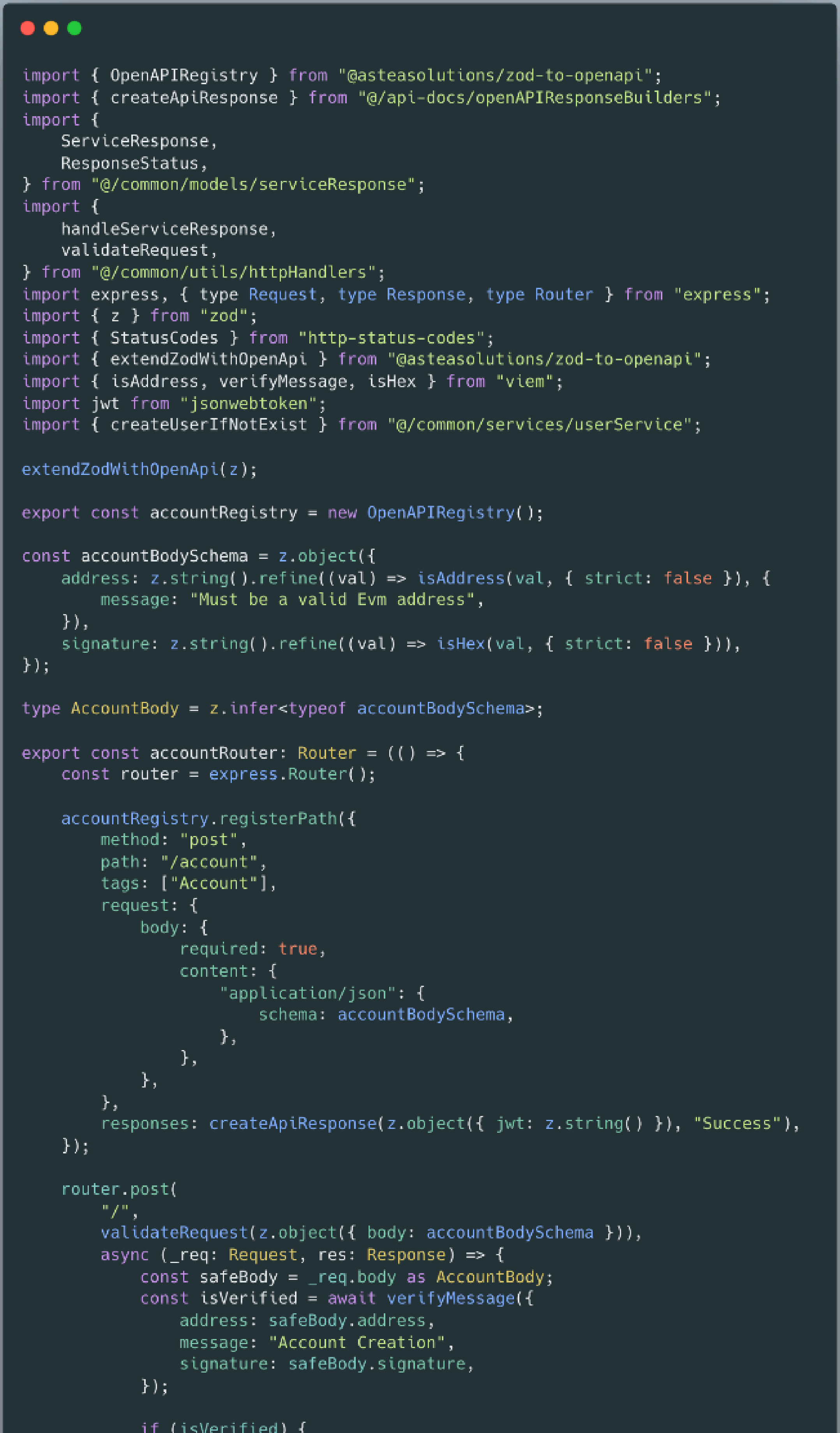
```
import { Alchemy, Network, type PortfolioAddress } from "alchemy-sdk";
import type { Address } from "viem";
import type { AlchemyTokenResponse } from "@/common/models/alchemyTokenResponse";
import { convertChainIdToAlchemyNetwork } from "@/common/utils/networkConverter";
const settings = {
    apiKey: process.env.ALCHEMY_KEY,
    Network: Network.ETH_MAINNET,
};

const alchemy = new Alchemy(settings);

export const getTokensByWallet = async (address: Address, chainId?: number) => {
    let tokensResult: AlchemyTokenResponse[] = [];
    const network = convertChainIdToAlchemyNetwork(chainId);
    const porfolioAddress: PortfolioAddress[] = [
        { address, networks: [network] },
    ];
    try {
        const result = await alchemy.portfolio.getTokensByWallet(porfolioAddress);
        const { pageKey, tokens } = result.data;
        // Alchemy provides a `pageKey`, so I should be able to iterate and fetch all my tokens,
        // but it seems I can't use it with the current API…
        // For the sake of the exercise, I'll just return the first page.

        tokensResult = tokens;
    } catch (error) {
        throw new Error("Error while fetching tokens from alchemy");
    }

    return tokensResult;
};
```

/account

For the /account endpoint I validate the signature sent by the frontend and if the signature is valid I create an account that I store in a database (SQLite for the simplicity of the exercise), the ORM I use is Prisma. Then I create a JWT to handle my authentication on the frontend side. If the account already exists, I don't create it and directly return the JWT (always after signature verification).

/leaderboard

For the /leaderboard endpoint, I assumed that you needed to be authenticated (via JWT) to access this endpoint. So I created an AuthMiddleware.ts that simply verifies the validity of the JWT (which has a lifespan of one week). I return the content of the user table as a leaderboard. (I didn't really know by what criteria to rank users, so I chose to simply retrieve the data from the User database and display it on the frontend.)

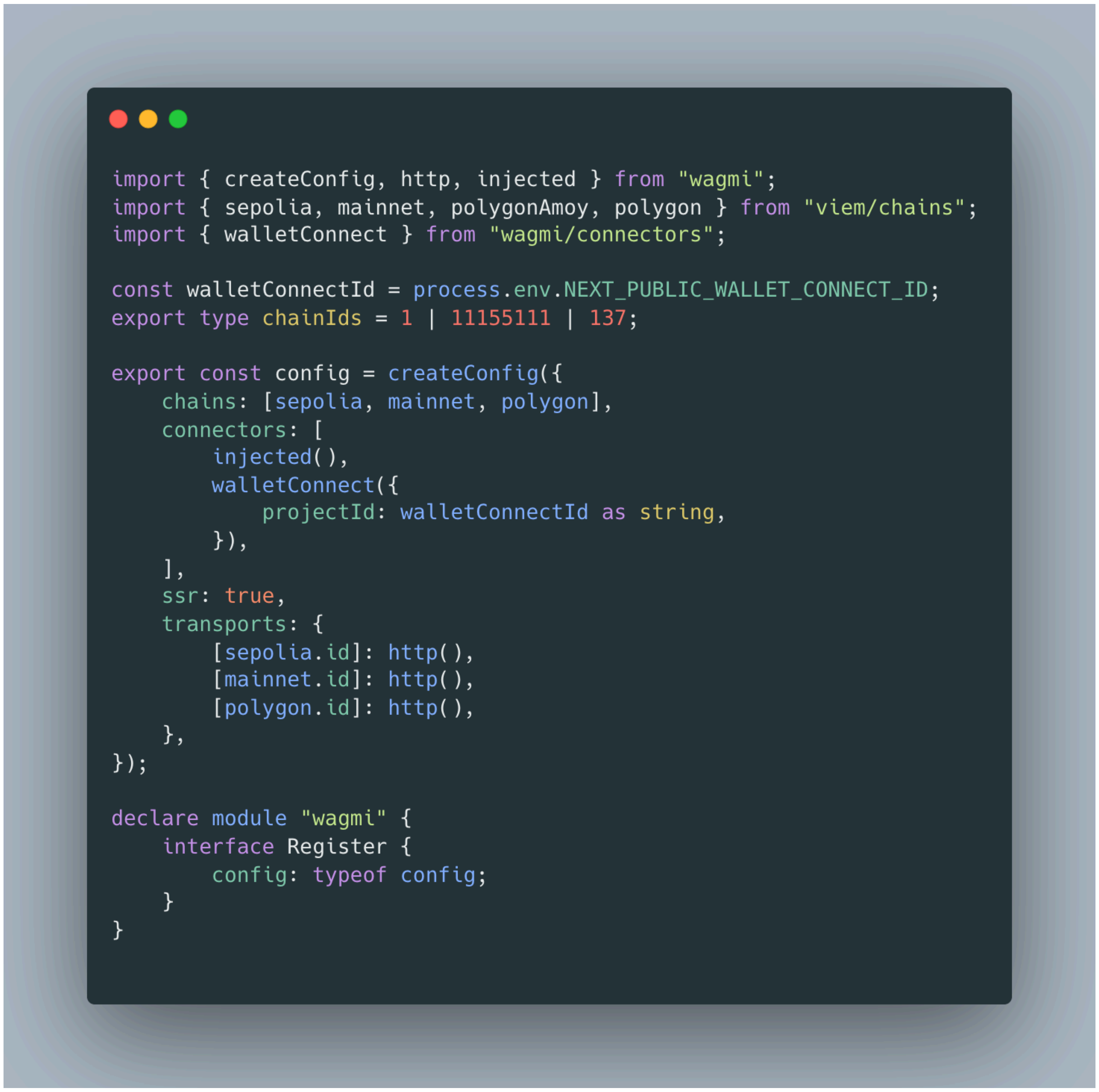


```typescript
import { OpenAPIRegistry } from "@asteasolutions/zod-to-openapi";
import { createApiResponse } from "@/api-docs/openAPIResponseBuilders";
import {
    ServiceResponse,
    ResponseStatus,
} from "@/common/models/serviceResponse";
import {
    handleServiceResponse,
    validateRequest,
} from "@/common/utils/httpHandlers";
import express, { type Request, type Response, type Router } from "express";
import { z } from "zod";
import { StatusCodes } from "http-status-codes";
import { extendZodWithOpenApi } from "@asteasolutions/zod-to-openapi";
import { isAddress, verifyMessage, isHex } from "viem";
import jwt from "jsonwebtoken";
import { createUserIfNotExist } from "@/common/services/userService";

extendZodWithOpenApi(z);

export const accountRegistry = new OpenAPIRegistry();

const accountBodySchema = z.object({
    address: z.string().refine((val) => isAddress(val, { strict: false }), {
        message: "Must be a valid Evm address",
    }),
    signature: z.string().refine((val) => isHex(val, { strict: false })),
});

type AccountBody = z.infer<typeof accountBodySchema>;

export const accountRouter: Router = (() => {
    const router = express.Router();

    accountRegistry.registerPath({
        method: "post",
        path: "/account",
        tags: ["Account"],
        request: {
            body: {
                required: true,
                content: {
                    "application/json": {
                        schema: accountBodySchema,
                    },
                },
            },
        },
        responses: createApiResponse(z.object({ jwt: z.string() }), "Success"),
    });

    router.post(
        "/",
        validateRequest(z.object({ body: accountBodySchema })),
        async (_req: Request, res: Response) => {
            const safeBody = _req.body as AccountBody;
            const isVerified = await verifyMessage({
                address: safeBody.address,
                message: "Account Creation",
                signature: safeBody.signature,
            });
```

Frontend app

For the client, the structure of my application is as follows: The user must first connect using Wallet Connect or the Injected provider. Then they are on the homepage where the tokens from their wallet will load for the given blockchain (you can choose between Ethereum, Sepolia, and Polygon).

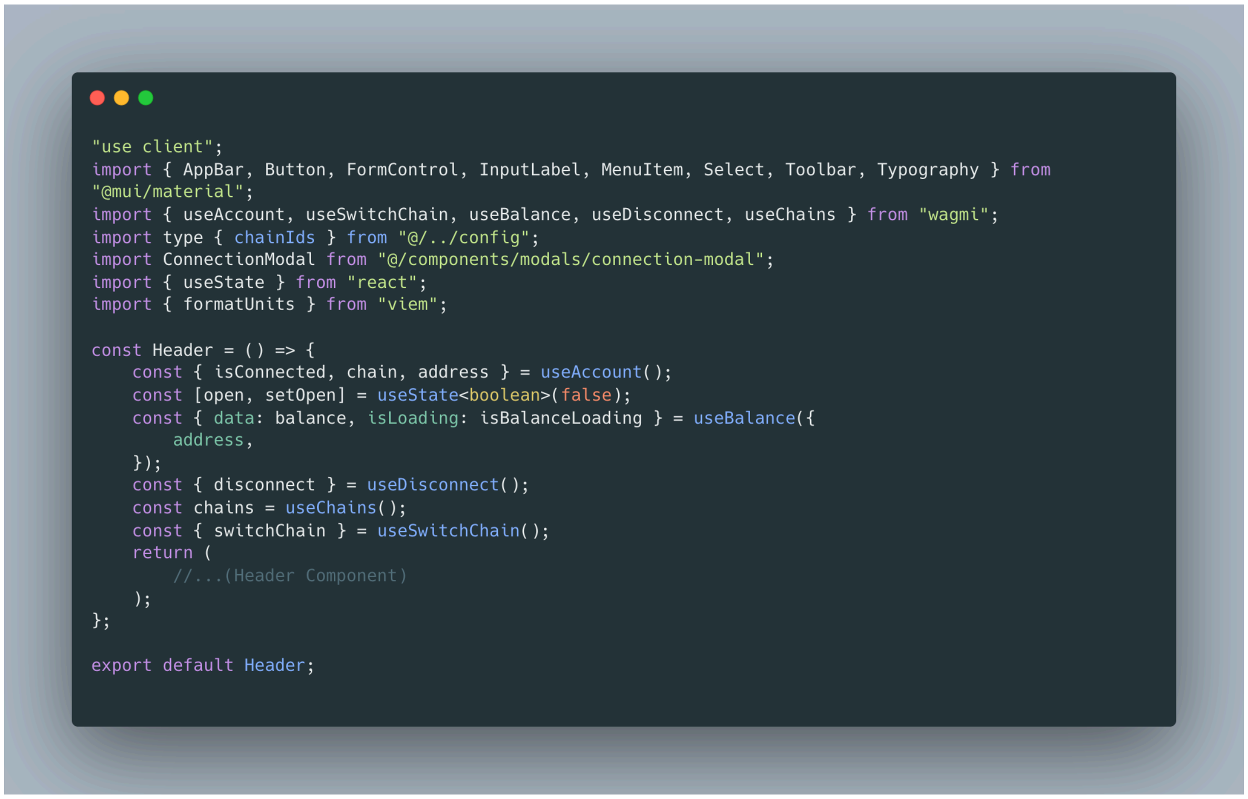
If they go to the leaderboard page, we will ask them to log in or create an account. They will have to sign a message (which will then be verified on the backend side). Once the account is created (thanks to the /account endpoint) a database entry will be created and the user will receive a JWT proving they are properly authenticated. Once they are authenticated they can access the leaderboard (which is simply the content of the User table for the purpose of the exercise).



```
import { createConfig, http, injected } from "wagmi";
import { sepolia, mainnet, polygonAmoy, polygon } from "viem/chains";
import { walletConnect } from "wagmi/connectors";

const walletConnectId = process.env.NEXT_PUBLIC_WALLET_CONNECT_ID;
export type chainIds = 1 | 11155111 | 137;

export const config = createConfig({
    chains: [sepolia, mainnet, polygon],
    connectors: [
        injected(),
        walletConnect({
            projectId: walletConnectId as string,
        }),
    ],
    ssr: true,
    transports: {
        [sepolia.id]: http(),
        [mainnet.id]: http(),
        [polygon.id]: http(),
    },
});

declare module "wagmi" {
    interface Register {
        config: typeof config;
    }
}
```

Blockchain connectivity

I chose to use Wagmi and Viem which allows me to very quickly implement connectors (MetaMask and Wallet Connect) and then use hooks like useAccount, useBalance, useSwitchChains, etc. that allow me to quickly retrieve data and perform basic actions on the blockchain.



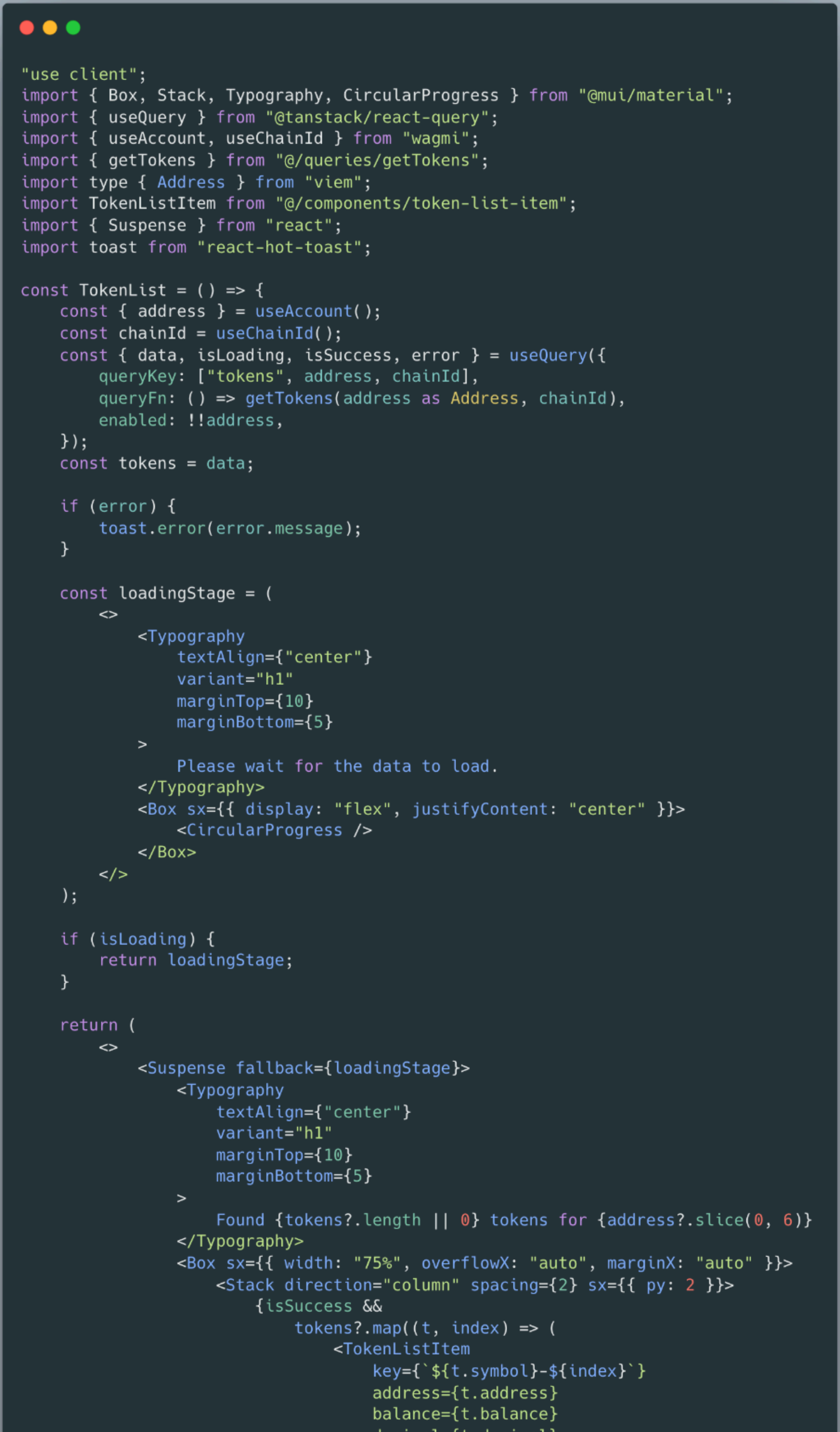
```
"use client";
import { AppBar, Button, FormControl, InputLabel, MenuItem, Select, Toolbar, Typography } from
"@mui/material";
import { useAccount, useSwitchChain, useBalance, useDisconnect, useChains } from "wagmi";
import type { chainIds } from "@/../config";
import ConnectionModal from "@/components/modals/connection-modal";
import { useState } from "react";
import { formatUnits } from "viem";

const Header = () => {
    const { isConnected, chain, address } = useAccount();
    const [open, setOpen] = useState<boolean>(false);
    const { data: balance, isLoading: isBalanceLoading } = useBalance({
        address,
    });
    const { disconnect } = useDisconnect();
    const chains = useChains();
    const { switchChain } = useSwitchChain();
    return (
        //...(Header Component)
    );
};

export default Header;
```

Typing the frontend

Thanks to the Zod typing of the OpenAPI, I managed to find a library that allows me to generate a typed client to call my requests. So I created methods in the query folder that call these other methods generated by this client and that allow me to type my frontend application with the backend types.

Ui state management

Thanks to TanStack Query I can easily call my queries that will request my backend to retrieve the data I'm interested in. I can also manage loading and error states with the isLoading and error parameters. In case of an error in the chain, a popup appears to display a client message about the error.

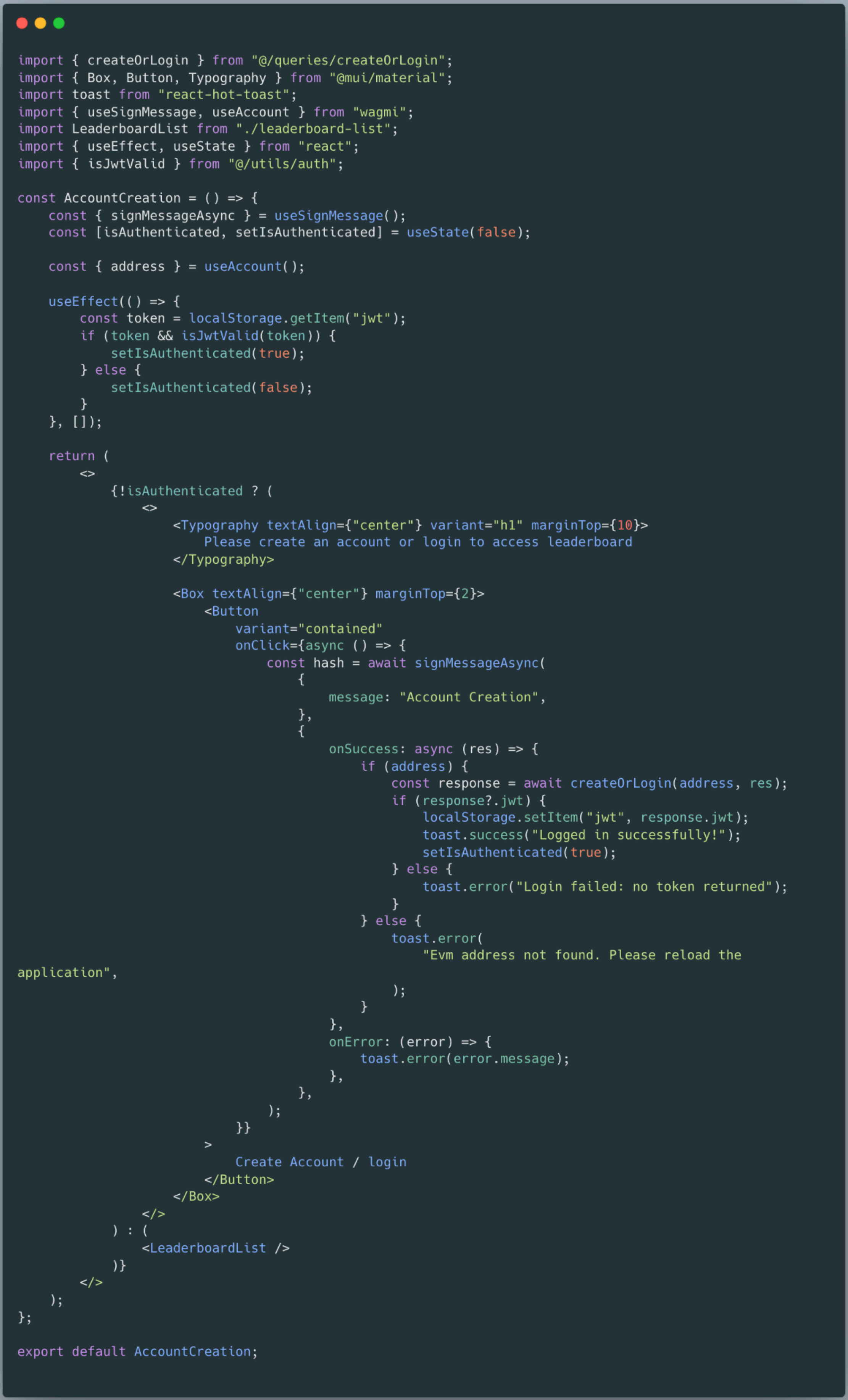


```
"use client";
import { Box, Stack, Typography, CircularProgress } from "@mui/material";
import { useQuery } from "@tanstack/react-query";
import { useAccount, useChainId } from "wagmi";
import { getTokens } from "@/queries/getTokens";
import type { Address } from "viem";
import TokenListItem from "@/components/token-list-item";
import { Suspense } from "react";
import toast from "react-hot-toast";

const TokenList = () => {
    const { address } = useAccount();
    const chainId = useChainId();
    const { data, isLoading, isSuccess, error } = useQuery({
        queryKey: ["tokens", address, chainId],
        queryFn: () => getTokens(address as Address, chainId),
        enabled: !!address,
    });
    const tokens = data;

    if (error) {
        toast.error(error.message);
    }

    const loadingStage = (
        <>
            <Typography
                textAlign={"center"}
                variant="h1"
                marginTop={10}
                marginBottom={5}
            >
                Please wait for the data to load.
            </Typography>
            <Box sx={{ display: "flex", justifyContent: "center" }}>
                <CircularProgress />
            </Box>
        </>
    );

    if (isLoading) {
        return loadingStage;
    }

    return (
        <>
            <Suspense fallback={loadingStage}>
                <Typography
                    textAlign={"center"}
                    variant="h1"
                    marginTop={10}
                    marginBottom={5}
                >
                    Found {tokens?.length || 0} tokens for {address?.slice(0, 6)}
                </Typography>
                <Box sx={{ width: "75%", overflowX: "auto", marginX: "auto" }}>
                    <Stack direction="column" spacing={2} sx={{ py: 2 }}>
                        {isSuccess &&
                            tokens?.map((t, index) => (
                                <TokenListItem
                                    key={`${t.symbol}-${index}`}
                                    address={t.address}
                                    balance={t.balance}
```

Account creation, verification and authentication

And regarding my leaderboard page, I offer the user to create an account or log in by signing a message (which is the same on the backend side). The hash is then sent to the endpoint for verification and if the verification is correct I receive a JWT that I store in storage.

It would probably have been cleaner to create a dedicated hook but for the needs of the exercise and since I only need it here I grouped the logic in this component.



```
import { createOrLogin } from "@/queries/createOrLogin";
import { Box, Button, Typography } from "@mui/material";
import toast from "react-hot-toast";
import { useSignMessage, useAccount } from "wagmi";
import LeaderboardList from "./leaderboard-list";
import { useEffect, useState } from "react";
import { isJwtValid } from "@/utils/auth";

const AccountCreation = () => {
    const { signMessageAsync } = useSignMessage();
    const [isAuthenticated, setIsAuthenticated] = useState(false);

    const { address } = useAccount();

    useEffect(() => {
        const token = localStorage.getItem("jwt");
        if (token && isJwtValid(token)) {
            setIsAuthenticated(true);
        } else {
            setIsAuthenticated(false);
        }
    }, []);

    return (
        <>
            {!isAuthenticated ? (
                <>
                    <Typography textAlign={"center"} variant="h1" marginTop={10}>
                        Please create an account or login to access leaderboard
                    </Typography>

                    <Box textAlign={"center"} marginTop={2}>
                        <Button
                            variant="contained"
                            onClick={async () => {
                                const hash = await signMessageAsync(
                                    {
                                        message: "Account Creation",
                                    },
                                    {
                                        onSuccess: async (res) => {
                                            if (address) {
                                                const response = await createOrLogin(address, res);
                                                if (response?.jwt) {
                                                    localStorage.setItem("jwt", response.jwt);
                                                    toast.success("Logged in successfully!");
                                                    setIsAuthenticated(true);
                                                } else {
                                                    toast.error("Login failed: no token returned");
                                                }
                                            } else {
                                                toast.error(
                                                    "Evm address not found. Please reload the
application",
                                                );
                                            }
                                        },
                                        onError: (error) => {
                                            toast.error(error.message);
                                        },
                                    },
                                );
                            }}
                        >
                            Create Account / login
                        </Button>
                    </Box>
                </>
            ) : (
                <LeaderboardList />
            )}
        </>
    );
};

export default AccountCreation;
```

What could be improved :

- I wrote some tests but I guess I could have written more but it’s not my strength.
- There is probably a way to only use the validateRequest middleware for strong typing.
- The way I authenticated the user in the front end is probably too simplistic but I thought it was enough for the exercise.
- I only did caching for the alchemyService because I think it was the most appropriate but we could probably do it as well in other area.

I hope everything is clear enough. Don’t hesitate to reach me if you have any issue with the code.

Regards,

Sylvain