Projektová dokumentace (DSP)

# Development stack

## Backend

Jelikož tento semestr toho je extrémně moc, vybral jsem cestu SSR oproti konvenčnímu frontent/backend řešení. Nejstrategičtější řešení pro mě byl fullstack Python framework **Django**. Je to takzvaný *batteries included* framework – prakticky vše je do nějaké míry již implementováno, na mě je rozšířit stávající infrastrukturu, navrhnout HTML, apod.

Základem jsou *Views*, každé view reprezentuje jedna třída, která má přiřazený .html souboru a je namapována na určitou url. Pomocí tzv. *mixinů* (python scuffed multiple inheritance) se dá funkcionalita pohledů rozšiřovat. Data do HTML souborů dostávám pomocí kontextu a zobrazuji je pomocí **Django template language**.

## Front end

Na straně front endu využívám CSS framework Bulma, který mi umožňuje snadno vytvářet responzivní design a rychle generovat uživatelské rozhraní. Jelikož tato webová aplikace využívá technologii SSR (Server Side Rendering), není třeba žádného frontendového frameworku jako React, VueJS, Angular apod. Stránka přijde ze serveru již vyrenderována, a jakékoliv dodatečné frontent UI/UX je obstaráno obyčejným klientským Javascriptem.

# Autentizace

Jak jsem již zmínil, Django je „batteries included“ framework, *out of the box* obsahuje velice powerful User/Accounts systém. Na rozdíl od mých kolegů, se tedy nemusí zaobírat implementací zabezpečení aplikace pomocí JWT apod (toho jsem měl až až na maturitním projektu). V mém případě jsem tedy jen rozšířil již existující **User** model o pár specifických atributů a funkcí, a o ostatní se stará Django.

Jednou z podmínek pro zápočet byla také implementace dvoufázového ověření – nehledě na to, jaký typ si student zvolí. Jelikož se mi nechtělo patlat s SMTP a emaily, zvolil jsem populární 2FA metodu – TOTP.

S využitím knihovny pyotp a qrcode můžeme snadno implementovat dvou faktorovou autentizaci v naší aplikaci. PyOTP umožňuje generovat a ověřovat jednorázové kódy pro autentizaci a Qrcode knihovna zase umožňuje generování QR kódů pro zobrazení a snadné skenování v aplikaci Google Authenticator.

Implementace byla nadmíru jednoduchá – určitě jednodušší než posílat ověřovací kódy na email. Pokud bych to dělal znovu, myslím si, že bych to mohl mít do 20 minut hotové, tak jednoduché to je. Pro implementaci stačí pouze natahat již zmíněné knihovny – dodat secret key, uživatelovo jméno a jméno naší aplikace (**issuer name**), a knihovna vyprdne nazpátek OTP URI. Tato OTP URI má určitý formát (více [zde](https://github.com/google/google-authenticator/wiki/Key-Uri-Format)), který lze snadno uložit do QR kódu, který je pro uživatele mnohem jednodušší na použití.

# Databáze

Django defaultně používá vestavěnou SQLite databázi - .sqlite soubrou v rootu projektu – komunikuje se s ní za pomoci ORM. Vývojář definuje takzvané *Modely* – Python třídy, které dědí z *models.Model*, na dané třídě definuje potřebné atributy, parciality, kardinality a ostatní SQL blbosti se také nastavují v naší model třídě. Když máme navrhnutou strukturu databáze, řekneme Djangu, že se má struktura databáze změnit, a Django – za pomoci tzv. migrací, které se vytvářejí po spuštění příkazu *python manage.py makemigrations* – upraví strukturu databáze. Já se tedy nemusím vrtat přímo v SQL, stačí mi pouze nadefinovat třídy a atributy, které korespondují k entitám a jejich atributům. Níže je vidět celá struktura hlavních modelů, díky kterým je chod banky možný.

## Modely

Hlavním modelem naší banky je model **Account**. Ukládá si vlastníka, jako ForeignKey na model User, typ, číslo apod.

Nic, co se týče peněz, se neukládá do tohoto modelu, od toho je model **AccountBalance**. Idea za tímto modelem je taková, že každný Account má na sebe napojené instance tohoto modelu. Každá instance AccountBalance má již zmíněný Account jako ForeignKey, měnu a množství peněz v dané měně. Pokud tedy účet má dvě instance AccountBalance – CZK a USD, může manipulovat s penězi v těchto měnách. Účet musí mít jeden AccountBalance nastavený jako hlavní, ten diktuje, jak se bude celková hodnota účtu zobrazovat, a do jaké měny se převádí přicházející transakce, v případě, že účet nemá požadovanou měnu. Hlavní měna účtu se dá měnit za chodu.

Dalším velice důležitým modelem je model **Transaction**. Instance tohoto modelu je vytvořena pokaždé, když se má uskutečnit jakýkoliv pohyb peněz (převod na účet, výběr, vklad, atd.). Seznam těchto instancí poté na front endu tvoří historii pohybů peněz na účtu.

Bylo také potřeba nějak ukládat data, která přijdou z API České Národní Banky – měnové kurzy. Jelikož *fetch period* je nastaven na 6h, data jsem se rozhodl ukládat do databáze a ne přímo do paměti. Zde přichází na řadu model **CurrencyRate**. Poté co přijdou data z ČNB, pro každou měnu je vytvořena instance tohoto modelu a uložena do databáze. Pokud již záznamy existují, jsou jenom updatovány.

Toto je krátké shrnutí designu databáze této aplikace, pro lepší porozumění jsem do zipu přiložil UML diagram vygenerovaný mým IDE.

# Testování

Musím se přiznat, že jsem za 21 let svého života nenapsal jediný test, takže tato část zadání byla poměrně náročná a sežrala spoustu času. Django má v sobě zabudovaný jednoduchý testovací framework takže setupnout první testy a potřebné mocky nebylo zase tak náročné. Každý aplikace v mém projektu – Django terminologie pro modul – má soubor *tests.py*, kde jsou testovací třídy a funkce. Nainstaloval jsem si také nástroj, který mi dokáže spočítat, kolik kódu mám otestováno, a vyprdne hezký interaktivní HTML bundle, kde je vidět, přesně jaké cally funkcí, konstrukturů apod., je třeba otestovat. Nástroj se nazývá **coverage** (*pip install coverage*). Testování je dále využito pro CI část pipeline.

# CI/CD

Další z věcí, na které jsem nikdy nešáhl. Vždy jsem věděl že existuje něco jako **Github Actions** a co to trochu dělá, nevěděl jsem ovšem jak komplexní a schopný to je systém. Jelikož tento projekt, spolu s prakticky každou aplikací, kterou jsem kdy napsal, ukládám na Githubu, Actions byla jasná volba pro setupnutí CI/CD pipeline pro tuto aplikaci.