FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	GoDeliver - implementace klientské části modulu Administrace
Student:	Matouš Škoda
Vedoucí:	Ing. Filip Glazar
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je implementovat klientskou část modulu Administrace projektu GoDeliver. GoDeliver je projekt v produkční fázi, který slouží ke sledování a správě zásilek. Jedná se o kompletní implementaci doručovacího systému. V rámci této práce autor má za úkol vyvinout v javascriptovém frameworku React frontend pro administraci. Vývoj bude probíhat na základě dodaných návrhů firmou Cognitic s.r.o.

Softwarové řešení GoDeliver umožňuje svým klientům vkládat a sledovat zásilky. V rámci systému jsou evidovány všechny zásilky daného klienta, včetně jejich aktuálního stavu a kurýrů, které je doručují.

1) Analyzujte dodané návrhy a dokumentaci k API serverové části
2) Na základě předchozí analýzy navrhněte případné změny v UX/UI
3) Vyberte vhodné doprovodné technologie implementace
4) Implementujte prototyp klientské části
5) Proveďte uživatelské testování prototypu
6) Na základě výsledků testování navrhněte vhodné úpravy
7) Připravte systém pro produkční nasazení. Zvolte vhodné CI/CD

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 26. ledna 2021 v Praze.

Bakalářská práce

GODELIVER – IMPLEMENTACE KLIENTSKÉ ČÁSTI MODULU ADMINISTRACE

Matouš Škoda

Fakulta informačních technologií ČVUT v Praze
Katedra softwarového inženýrství
Vedoucí: Ing. Filip Glazar
13. května 2021

Odkaz na tuto práci: Matouš Škoda. *GoDeliver – implementace klientské části modulu Administrace.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat především Ing. Filipu Glazarovi za vedení a konzultace při tvorbě této práce. Dále bych chtěl poděkovat zakladatelům a vývojovému týmu firmy Cognitic za to, že mi umožnili podílet se na projektu GoDeliver. Také bych rád poděkoval rodině za podporu během psaní práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č.121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. května 2020 .....

Abstrakt

Bakalářská práce se zabývá implementací klientské části webového administračního rozhraní logistického systému GoDeliver. Aplikace slouží jako kompletní administrační rozhraní pro dispečery společností, umožňující kontrolu nad automatizovaným rozvážkovým systémem. V analýze stávajícího serverového řešení jsou popsány entity vyskytující se v systému, dále jsou analyzovány dodané grafické návrhy. Poté následuje popis nastavení nástrojů pro vývoj a automatickou integraci a nasazení aplikace. V implementační části je nejprve vysvětlen postup vytvoření datové vrstvy aplikace a následně je popsána implementace jednotlivých obrazovek. V poslední části práce je proveden uživatelský průzkum použitelnosti. Výsledkem práce je komplexní administrační rozhraní nasazené v produkčním prostředí, využívané společnostmi k rozvozu zásilek.

Klíčová slova logistika, administrační rozhraní, webová aplikace, front-end, React

Abstract

The bachelor thesis presents the implementation of a web administration interface for the GoDeliver logistics service. The application serves as a complete administration interface for the dispatchers of companies utilizing said service, allowing them control of the automated logistics system. The analysis of the current server-side solution describes basic entities that appear in the system and derives functional requirements from supplied graphic designs. The implementation then gives details of configuration of development tools and continuous integration and delivery, it further describes the development process of the data and presentation layers of the application. Lastly the thesis describes the process of usability testing of the implemented system. The result of the work is a complex web administration interface deployed in a production environment, used by several companies for day-to-day deliveries.

Keywords logistics, administration interface, web application, front-end, React

Seznam zkratek

REST	Representational State Transfer
API	Application Programming Interface
CLI	Command Line Interface
UI	User Interface
UX	User Experience
UXD	User Experience Design
CI/CD	Continuous Integration and Continuous Delivery
XML	Extensible Markup Language
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
SSG	Server Side Generation
SSR	Server Side Rendering

Kapitola 1
Úvod

Logistika je, zejména v dnešní době, kdy se na dovoz zboží přechází z nutnosti, pro některé firmy naprosto vše. Větší společnosti, nebo ty s již zaběhlým logistickým systémem, tak získaly obrovskou konkurenční výhodu oproti firmám menším, které často nemají dostatečné prostředky pro to, aby si nechaly vytvořit systém na míru.

Ve výsledku mnoho firem řeší logistiku takříkajíc „na koleně“, tedy rozvážky plánují jednotlivé pobočky ručně a dochází k obrovské neefektivitě ve využití kurýrů. Zároveň dochází k problémům se zpožděnými dodávkami při neodhadnutí dopravní situace zejména ve městech.

Tento problém se snaží vyřešit start-up Cognitic s.r.o. prostřednictvím služby GoDeliver, která firmám poskytuje kompletní logistické řešení. Služba klientům nabízí zejména mobilní aplikaci určenou pro kurýry a automatické plánování dovozu vytvářených zásilek napojením stávajícího systému společnosti na serverové řešení (dále označované jako backend) pomocí REST rozhraní (dále jen API). Kurýři tak rozváží podle aplikace v mobilním telefonu a jsou jim přidělovány naplánované trasy s využitím aktuálních dat o dopravní situaci po cestě. Jednotlivé pobočky zároveň také dopředu mají informace o tom, kdy očekávat jakého kurýra a jaké další zásilky mají připravit pro rozvoz.

Tímto způsobem funguje služba aktuálně a do velké míry zjednodušuje logistiku firem, nicméně u některých je napojení stávajících systémů obtížné a není tak možné využít všechna poskytovaná data. Proto je dalším plánovaným modulem služby GoDeliver klientská webová aplikace (dále označovaná jako frontend), určená pro dispečery společností, která by měla poskytovat přehledné informace o práci systému, vizualizace dat a možnosti ovládání systému ručně. Vytvoření tohoto modulu je hlavním tématem této práce.

Přínosem této aplikace by bylo kromě poskytnutí možností ručního ovládání a kontroly práce systému zejména zpřístupnění datové analýzy. Společnosti by tak měly informace například o vytížení poboček v průběhu dne, kolik kurýrů je potřeba na kterých pobočkách na další den, a jestli je nutné navýšit počty kurýrů. Zároveň by bylo možné poskytovat individuální funkcionalitu pro jednotlivé klienty a rozhraní upravovat na míru.

Hlavním cílem této práce je tedy implementovat frontend systému GoDeliver pomocí Javascript frameworku React dle dodaných návrhů, ten následně napojit na existující API a celou aplikaci nasadit ve vývojové verzi (pro účely testování a vývoje nových částí) a produkční verzi.

V teoretické části práce představuji technologie a postupy vývoje spojené s frontendovými aplikacemi, také popisuji přístupy uživatelského testování. V praktické části pak analyzuji stávající serverové řešení a strukturu API systému. Dále určuji entity vyskytující se v systému a jejich vztahy. Poté analyzuji dodané návrhy a z nich plynoucí požadavky na funkcionalitu. Následně rozebírám samotnou tvorbu a nasazení aplikace, její strukturu a využité technologie. V poslední řadě popisuji uživatelské testování a z něho vyvozuji závěry pro budoucí vývoj modulu.

Kapitola 2
Cíl práce

Hlavním cílem bakalářské práce je vytvoření klientské části administračního rozhraní softwarového projektu GoDeliver.
Dílčími cíli práce jsou:

- Analyzovat dodané návrhy a z nich vycházející požadavky na implementaci aplikace.
- Analyzovat stávající serverové řešení a API z hlediska využitých technologií a struktury.
- Implementovat prototyp klientské aplikace.
- Provést uživatelské testování implementovaného prototypu.
- Připravit systém pro produkční nasazení.

Výsledná aplikace bude pomáhat společnostem, využívajícím GoDeliver, k administraci rozvážkového systému, k zpřístupnění datové analýzy a ke zvýšení efektivity rozvážek.

Kapitola 3
Teoretická část

3.1 Frontendové technologie a React

V této kapitole popisuji webové technologie využité k implementaci aplikace. Nejprve vylíčím základní webové technologie a pojmy s nimi spojené, dále na nich stavějící knihovny a frameworky.

3.1.1 HTML, CSS a JavaScript

HTML a CSS jsou dvěma technologiemi v jádru všech webových stránek. Spolu se skriptovacím jazykem JavaScript jsou základními stavebními bloky webu a webových aplikací.

HTML

HTML je tzv. *značkovací* jazyk, určený pro strukturování a popis dokumentů, značkovací od toho, že pro zmíněný popis dokumentů se používají značky (anglicky tag), které dávají dokumentům sémantický význam. Dále HTML umožňuje v dokumentech vytvářet *hypertextové* odkazy mířící do jiných částí dokumentu, ale také do dokumentů jiných, z tohoto zkratka HTML – anglicky Hypertext Markup Language. [1]

Blokům označených značkami se říká *elementy* a sestávají z počáteční značky, obsahu a ukončující značky, nebo pouze jedné rovnou ukončené značky (například `<input/> pro vstup`). Značkování pomocí HTML dále v dokumentech udává, jakou funkci má daný blok dokumentu plnit. Tedy například použití značek `<p>Text</p>` udává, že slovo „Text“ uvnitř značek je samostatný odstavec a měl by se tak zobrazit uživateli, těmto elementům se říká *sémantické*, opakem jsou například elementy `<div></div>` a `<span></span>`, které žádný význam nemají, pouze seskupují obsah buď do bloku, nebo do řádku. [1]

Dále HTML využívá tzv. *atributů*, které jsou určené pro identifikaci elementů v dokumentech nebo změnu chování, zejména jde o atributy `id` a `class` pro identifikátor a třídy, existuje řada dalších a specifických pouze pro konkrétní elementy, například u již zmíněného elementu `<input/>` atribut `type` mění typ vstupu, použití atributu vypadá následovně: `<input type="date" />`.

Sémantického významu elementů HTML používají webové prohlížeče k určení, jak by se měl daný element chovat a prezentovat uživateli. Prohlížeče mají pro každý element nějaké výchozí chování a vzhled, je tedy možné napsat webovou stránku pouze za pomoci HTML. Výchozí chování a vzhled ale často není takové, jaké bychom chtěli a tak musíme využít možnosti CSS nebo JavaScript.

CSS

CSS (anglicky Cascading Style Sheets) je jazyk určen pro popis vzhledu dokumentů. Není určen pouze pro HTML, ale je možné ho využít s jakýmkoliv jazykem založeným na XML (metajazyk umožňující tvorbu dalších značkovacích jazyků). CSS dokumenty pomocí *selektorů* umožňují vybrat určité elementy a přiřadit jim požadované vzhledové vlastnosti na základě *pravidel*, selektory jsou zápisem označujícím názvy, vlastnosti nebo atributy, na základě kterých pak algoritmus prohlížeče přiřadí elementům vzhled. [2]

Selektorů existuje celá řada, pro příklad přikládám ukázku. Selektor `p` označí všechny odstavce, dále selektor `#my_id` značí, že element musí mít identifikátor `my_id`, a selektor `.my_class` vybere pouze elementy s touto třídou. Pro takto vybrané elementy (jeden element, ID musí být v celém dokumentu unikátní) je pak použito specifikované pravidlo, tedy je jim přiřazena barva textu `red`.

```
1    p#my_id.my_class {
2        color: red;
3    }
```

Výpis kódu 3.1 Ukázka použití CSS

Elementům takto vybraným může být přiřazeno více pravidel, měnících stejné vlastnosti, i z různých CSS dokumentů, právě z „kaskádování“ pravidel vychází název jazyka CSS. [3]

JavaScript

JavaScript je dynamický skriptovací jazyk původně určený pro spouštění pouze v prohlížečích, jeho primárním účelem bylo umožnění rozšíření funkcionality webových stránek. Dnes není používán jen v prohlížečích, ale i na serverech a v dalších aplikacích (například serverový runtime Node.js [4]).

JavaScript ve webových stránkách a aplikacích umožňuje dynamicky měnit dokumenty. Další z hlavních výhod jeho využití je to, že poskytuje přístup k API prohlížečů a také je pomocí něj možné odesílat požadavky na libovolné servery a služby a získávat a odesílat tak dynamicky data, na základě interakce uživatele s aplikací. [5]

3.1.2 TypeScript

V posledních deseti letech došlo k obrovskému růstu využití a popularity jazyka JavaScript, vznikl Node.js jako serverový runtime a JavaScript se dále rozšířil na servery. Ačkoliv se možnosti javascriptu rozšiřují, původně byl navržený pro malé skripty na webových stránkách a stále obsahuje nedostatky, které zesložiťují vývoj komplexnějších systémů. Z tohoto důvodu byl vytvořen TypeScript. [6]

TypeScript je typovanou nadstavbou jazyka JavaScript, umožňuje tedy vše, co JavaScript, ale rozšiřuje jazyk o další možnosti, zejména definice *typů*, podobně jako v jiných silně typovaných jazycích. TypeScript zároveň není runtime, ale jde o kompilátor, který přetransformuje zdrojové soubory zpět do javascriptu. Tímto způsobem je zachována možnost spustit aplikace napsané v typescriptu všude, kde bychom mohli spustit JavaScript, ale zároveň získáváme výhody spojené se statickou kontrolou a analýzou kódu. [6]

3.1.3 React

React je javascriptovou knihovnou (nebo frameworkem) pro tvorbu uživatelských rozhraní, umožňuje vytvářet komplexní UI za pomoci skládání menších *komponentů*. Při vytváření UI s pomocí knihovny pak vývojář nemusí psát HTML dokumenty, ale knihovna využívá zápisu dokumentu

v tzv. *JSX*. Knihovna pak interně využije javascriptového API prohlížeče k vytvoření HTML elementů ve stránce. [7]

V zápisu JSX je pak možné použít vnořený JavaScript, přístup je tedy podobný jako při použití šablon v jiných frameworcích. JSX jako takové vypadá téměř identicky jako HTML, s menšími rozdíly například v pojmenování atributů elementů, například `class` vs. `className`. Rozdíly jsou z těch důvodů, že třeba zmíněný atribut pro třídu, je klíčovým slovem jazyka JavaScript, a tak nemohl být použit. Velký rozdíl oproti HTML šablonám je ale v tom, že JSX je dále JavaScript *expression*, tedy je s ním možné pracovat, jako jakýmkoliv jiným výrazem javascriptu (vracet JSX z funkcí, předávat jako argument a podobně). [8]

React podporuje různé způsoby vytváření komponentů, popisovat budu dále jen komponenty jako *funkce*, jelikož jsou již de facto standardem a do budoucna je s nimi počítáno jako s primárním způsobem zápisu komponentů. React komponenty psané jako funkce jsou opravdu jen javascriptovou funkcí, která vrací JSX. Ještě nedávno tento způsob tvorby komponentů měl značné nevýhody, jako například to, že takové komponenty nemohly mít žádný interní stav. To se ale změnilo s příchodem React Hooks (dále počeštěně hook, žádný dobrý ekvivalent v češtině neexistuje) ve verzi 16.8, do té doby byla většina komponentů psána jako třídy. [9]

Vytvořený komponent pak může vypadat například takto. Na řádku 2 je použit hook `useState`, který zajistí, že hodnota proměnné `count` se při překreslení komponentu nezmění. Pomocí funkce `setCount` se pak nastavuje hodnota této proměnné. Na řádcích 4–9 je pak vidět použití JSX.

```
01  function Counter() {
02    const [count, setCount] = useState(0);
03    return (
04      <div>
05        <p>You clicked {count} times</p>
06        <button onClick={() => setCount(count + 1)}>
07          Click me
08        </button>
09      </div>
10    );
11  }
```

3.1.4 Next.js

React je často klasifikován jako knihovna, spíše než framework, jelikož sám o sobě neposkytuje řadu funkcionalit, které většina větších frameworků poskytuje, jako například optimalizace, slučování zdrojových kódů do balíků, routing apod. Pokud chceme použít React, pro všechny zmíněné funkcionality je nutné najít nějakou knihovnu, která nám je poskytne. [10]

Výše zmíněné obtíže řeší Next.js. Jde o framework pro vývoj aplikací s pomocí React. Next.js interně používá Node.js pro umožnění použití nástrojů a sestavování aplikace. Dále poskytuje řadu optimalizací, routing, podporu pro CSS knihovny a možnosti vytváření API. Více jsou nástroje poskytované Next.js popsány v další kapitole o vývojových nástrojích. [10]

3.2 Nástroje pro vývoj a principy průběžné integrace a nasazení

Vývoj aplikací ani zdaleka není jen o technologiích používaných aplikací a naprogramování aplikace samotné. Vývoje se dále týká řada nástrojů a technologií usnadňujících programování jako takové, zároveň je zásadním bodem vývoje aplikace průběžné integrace a nasazení výsledného artefaktu. Tyto nástroje a principy popisuji v této kapitole.

3.2.1 Nástroje pro sestavení

Jsou nástroje pro vytvoření spustitelné verze projektu, cílem je automatizování sestavení za účelem opakovatelnosti, minimalizace nutných manuálních kroků a sestavení nezávislé na vývojovém prostředí. Typicky nástroje používají skript (tzv. *build script*) v libovolném formátu (XML, JSON, apod.), který funguje jako předpis pro sestavení projektu.

Obsah tohoto skriptu určuje, jaké závislosti a knihovny projektu se mají stáhnout a jak projekt sestavit, dále je jeho obsahem spuštění testů a případný reporting.

3.2.1.1 Npm

Npm (zkratka Node Package Manager, psáno malými písmeny) je nástroj určený pro sestavování projektů postavených na jazyku JavaScript a systému Node.js. Název se zároveň překrývá s npm Registry, což je veřejný repozitář pro sdílení knihoven, balíčků a nástrojů. Npm jako nástroj je možné použít s jakýmkoliv veřejným repozitářem, výchozím je ale npm Registry. Alternativami k npm jako nástroji jsou například Yarn a pnpm. [11]

Při stažení a instalaci npm je zároveň instalován Node.js, a opačně, obě technologie jsou na sobě závislé, respektive nástroj npm využívá při sestavení projektu na pozadí Node.js. Integrace Node.js následně umožňuje spouštění dalších pomocných nástrojů psaných v jazyce JavaScript. Npm také poskytuje možnost spuštění nástrojů z registru balíčků bez jeho předchozí instalace pomocí příkazu `npx`.

Konfigurační soubor pro npm je pojmenován `package.json` a musí být ve formátu JSON. Konfigurace obsahuje základní informace jako název projektu, verzi projektu, a další jako popis, jméno autora, odkazy na webovou stránku a veřejný repozitář a podobné. Zmíněné položky konfigurace jsou určeny zejména pro projekty, které mají být publikovány ve veřejném repozitáři jako balíčky (ať už v npm, nebo jiném). V tomto bodě je důležité zmínit položku `license` s identifikátorem SPDX (seznam dostupný z https://spdx.org/licenses/), která u balíčků obsahuje licenci k použití a je tak možné kontrolovat licenci, když se vývojář rozhoduje, jestli závislost do svého projektu přidat. [12]

Dále soubor mimo jiné obsahuje dva seznamy závislostí projektu, které se dělí na produkční (položka `packages`) a vývojové (položka `devPackages`). V těchto seznamech jsou závislosti specifikovány jako název balíčku v repozitáři a jeho požadovaná verze. Npm v případě potřeby tyto závislosti následně instaluje do adresáře `node_modules/`. Závislosti nejsou přidávány ručně, ale používají se k tomu CLI příkazy npm, při změně `package.json` tímto způsobem jsou závislosti staženy a nainstalovány, dále je při každé této změně vygenerován nebo případně upraven soubor `package-lock.json`, který specifikuje přesnou verzi každé závislosti (včetně jejích vlastních závislostí), z důvodů zajištění opakovatelnosti sestavení projektu. [13]

Základní příkazy k použití npm pro práci se závislostmi projektu jsou následující (informace z jednotlivých stránek reference [14]):

- `npm init` – inicializuje nový repozitář (vytvoří nový výchozí `package.json`).
- `npm install [-D] "nazev_balicku[@verze]"` – přidá závislost jako produkční do `package.json`, s přepínačem `-D` jako vývojovou, a stáhne ji. Předponou „`@`“ lze určit

pomocí semantického verzování přesnou verzi balíčku, jako výchozí chování je použito `^1.1.1`, kde `1.1.1` je aktuální verze balíčku.

- `npm ci` – nainstaluje všechny závislosti dle `package-lock.json`. Jedná se o bezpečnější a rychlejší alternativu příkazu `npm install` použitého bez argumentů. Rozdíl mezi nimi je ten, že tento příkaz nikdy nemění `package.json`, selže pokud neexistuje `package-lock.json`, a zároveň kontroluje jestli si tyto soubory obsahově odpovídají.
- `npm update` – zkontroluje verze závislostí v repozitáři, stáhne a nainstaluje nejnovější verze balíčků dle `package.json` s přihlédnutím k sémantickému verzování. V případě, že chceme přejít na další verzi, musíme znovu použít `npm install`.
- `npm outdated` – zkontroluje verze balíčků a vypíše seznam kandidátů pro „update“, i těch, které jsou vyšší než sémantická verze, tedy u kterých musíme verzi navýšit ručně.
- `npm audit` – dle repozitáře projde záznamy o známých zranitelnostech závislostí a vypíše je. Tento příkaz má význam zejména pro produkční nasazení projektů.

Dále jsou součástí konfigurace v `package.json` skripty, specifikované jako seznam v položce `scripts`. Zpravidla se tyto skripty spouští příkazem `npm run muj_skript`, výjimkou je několik výchozích skriptů jako například `npm start`. Skripty se po použití příkazu spustí jako klasické příkazy z příkazové řádky. Zajímavou funkcionalitou je životní cyklus skriptů. Pokud máme například skript s názvem `"mujskript"` a vytvoříme dále skripty s názvy `"premujskript"` a `"postmujskript"`, budou se tyto provádět před a po provedení prvního skriptu. [15]

Skripty se používají zejména pro sestavení aplikace a spouštění nástrojů, které popisuji v další kapitole.

3.2.2 Nástroje pro vývoj aplikací

Vývoj aplikací a systémů obecně není jednoduchý a s rostoucím rozsahem projektu se pochopitelně zesložiťuje orientace ve zdrojových kódech a zvyšuje se pravděpodobnost vytváření chyb. U aplikací psaných v kompilovaných a silně typovaných programovacích jazycích chyby, alespoň pokud se jedná o hrubé chyby, odchytí často kompilátor, u jazyků slabě typovaných nebo interpretovaných tento luxus nemáme. Na chyby se tak narazí až při reálném vyhodnocení problematické části kódu a může být velmi obtížné je najít. Funkci kompilátoru jako „síta“ na chyby tak chceme u interpretovaných jazyků suplovat jinými nástroji pro statickou analýzu kódu. [16]

Zmíněný problém odchycení chyb řeší použití samotného TypeScriptu, ale pouze do určité míry. Existuje řada konstruktů, které sice jsou validní TypeScript, ale byli bychom raději, kdyby je vývojáři nepoužívali, protože často vedou k budoucím chybám. Příkladem mohou být například direktivy vypínání kontroly typů TypeScriptu (pravidla příkazy typu `"// @ts-ignore"`), nebo přemíra využívání typu `any`. Pro samotný jazyk JavaScript je typickou chybou například používání dvojitého rovnítka místo trojitého. K prevenci chyb založených na těchto konstruktech nám pomáhají nástroje zvané *lintery* (název od původního nástroje Lint určeného pro jazyk C). [17]

3.2.3 Principy a nástroje CI/CD

Průběžná integrace a nasazení (používanější je z angličtiny zkratka CI/CD pro Continuous Integration a Continuous Delivery) zahrnuje automatizaci všech úkonů a procesů nutných k sestavení a nasazení aplikace, především pro správné vydání její nové verze. Řetězci procesů prováděných během integrace a nasazení se říká CI/CD *pipeline*, ten může zahrnovat kromě sestavení a nasazení aplikace také mimo jiné automatické testování a analýzu kódu za účelem zlepšení kvality softwaru a předcházení chybným verzím aplikací. [18]

Obrovskou výhodou při použití CI/CD pipeline je právě automatizace, díky které se vývojáři mohou starat více o psaní kódu než jeho následné testování a ruční sestavování, zároveň automatizace předchází častým chybám z nepozornosti. Další výhodou je například možnost napojení dalších rozličných nástrojů a webových služeb na vydání nové verze softwaru, příkladem takovéto služby je Snyk [19], služba která umožňuje automatizovat nacházení bezpečnostních zranitelností v závislostech a využitých knihovnách projektu. [18]

Pro vytvoření CI/CD pipeline je typicky využíváno cloudových služeb, jako jsou například GitLab a GitHub [20, 21]. Tyto služby jsou používány jako komplexní DevOps platformy, DevOps je složeninou pro výrazy Development a Operations, jde o výraz pro přístup k vývoji softwaru zdůrazňující spolupráci a integraci při tvorbě projektu, spojuje vývojáře a odborníky na IT provoz. DevOps služby zpřístupňují nejen CI/CD, ale dále poskytují možnosti pro spolupráci na tvorbě kódu, zjednodušují code review, obsahují systémy pro vytváření *ticketů* (z důvodů přerozdělování práce mezi vývojáři) a další funkcionality. I přes tyto výhody existuje řada služeb, které se specializují pouze na CI/CD bez další nadstavby, příkladem jsou Jenkins, CircleCI a Travis CI. [22]

DevOps služby se mezi sebou liší, nicméně CI/CD pipeline je ve většině z nich nastaven pomocí konfiguračního souboru nebo více souborů a má zpravidla formu skriptu, který udává, kdy a v jakém případě se má provést určitá část pipeline. Vykonání sekvence úkonů je nejčastěji napojeno na změnu v určité větvi ve verzovacím systému (zejména Git), tedy třeba v případě, že ve větvi master přibyla nová verze.

CI/CD pipeline se dle [23] většinou dělí na několik fází, fatální chyba v jakémkoliv z těchto kroků zpravidla znamená ukončení. Pipeline se může skládat například z následujících kroků:

1. *Změna zdrojového kódu* – tato fáze spouští celý pipeline, jde třeba o novou verzi v nějaké konkrétní větvi sdíleného repozitáře.

2. *Sestavení* – v této fázi jsou provedeny úkony k sestavení projektu nebo aplikace. Ne u všech projektů je tato fáze nutná, například při použití jazyků jako Ruby a Python není potřeba kompilace programu.

   Dále v této fázi mohou být použity zpravidla před sestavením aplikace nástroje pro statickou analýzu kódu, například *lintery*, jejichž účelem je nejen odchycení případných chyb v kódu, ale i zabránění následně zbytečnému sestavování programu, které může být často velmi zdlouhavé (a v neposlední řadě i drahé, u některých služeb se proběhlé minuty CI/CD nad určitý limit platí).

3. *Testování* – po sestavení aplikace je potřeba ověřit korektnost vzniklého artefaktu. Tato fáze funguje jako síto, které by mělo odchytnout chybné verze softwaru a nepovolit následné nasazení. Z jednotlivých úkonů sem spadá například spuštění *unit testů* (česky jednotkové testy, málo používané) a dalších typů testování aplikací.

4. *Nasazení* – poté co byla aplikace sestavena a otestována lze očekávat, že je ve funkčním stavu, tudíž logickým dalším krokem je její nasazení. Pro tyto účely a pro menší a střední projekty jsou v dnešní době už zpravidla využívány cloudové služby, jakými jsou například Google App Engine, Amazon Web Services, Heroku a další.

   Ačkoliv jsou mezi jednotlivými platformami rozdíly ve způsobech nasazení, typicky jde o konfigurační soubor a nahrání projektu na hosting pomocí nástroje k tomu určenému, který dodává poskytovatel cloudových služeb.

Tímto procesem ale využití CI/CD platforem nekončí, poskytují navíc nástroje pro analýzu výsledků pipeline, monitoring a podobné. Další výhodou DevOps služeb jsou možnosti napojení na jiné služby, jako je například Slack [24] (aplikace pro týmovou komunikaci), díky čemuž se dále zjednodušuje spolupráce na projektech – členové týmu nemusí kontrolovat nespočet informačních kanálů, ale mají informace centralizované.

Výše popsané procesy jsou v zásadě vše nutné pro menší projekty, u větších jsou principy sice podobné, ale integrace a nasazení se komplikuje například o častější využití Docker kontejnerů a nutnost jejich následné orchestrace. CI/CD pipeline se tak s rostoucím rozsahem projektu rozšiřuje o řadu dalších konfiguračních souborů a procesů nutných k sestavení a nasazení částí projektu, v tomto ohledu je automatizované CI/CD pro udržení rozumné časové náročnosti nenahraditelné. [22]

3.2.3.1 GitHub Actions

Kapitola čerpá z [25]. Pro pochopení nastavení CI/CD v praktické části práce zde krátce popíšu způsob, jakým se vytváří pipeline ve službě GitHub. GitHub Actions je komponenta služby GitHub týkající se CI/CD (ekvivalent například GitLab CI/CD ve službě GitLab).

CI/CD se pro GitHub Actions tvoří pomocí tzv. *workflow* konfiguračních souborů, tyto soubory se musí nacházet v repozitáři v adresáři `.github/workflows/`, aby je GitHub rozpoznal. Konfigurační soubory musí být ve formátu YAML.

Základní jednotkou práce GitHub Actions je tedy workflow, který je spouštěn na základě nějaké události v repozitáři. Touto událostí může být typický *push* nebo *pull request* v nějaké větvi repozitáře, ale může jít o řadu dalších událostí, například události týkající se *issues* nebo *milestones*, možností je obrovské množství. Dále je možné spouštět workflow externě pomocí *webhooks*. [26]

Příklad základní konfigurace workflow který by se spustil při každém `git push` do větve `master`:

```
1  name: Workflow 1
2  on:
3      push:
4          branches:
5              - master
6  jobs:
7      ...
```

Výpis kódu 3.2 Ukázka konfigurace GitHub Workflow – 1

Další koncept, který je nutné popsat, je tzv. *runner*, jedná se o servery, na kterých jsou dále spouštěny části pipeline. Workflow je zjednodušeně řečeno sada úkonů, které se provedou na jednom nebo více runners.

Jednotlivé workflows se dále skládají z tzv. *jobs* (dále úlohy), kterých může mít jeden workflow více. Výchozí chování je takové, že se všechny úlohy ve workflow spustí paralelně, sekvenční spouštění je nastaveno popsáním závislostí mezi jednotlivými úlohami, tedy například pokud úloha 2 závisí na úloze 1, spustí se za sebou v pořadí nejdříve úloha 1 a následně při úspěchu úloha 2. [27]

Za zmínku dále stojí to, že se jedna úloha zaručeně provede v jednom runner prostředí, a tak se lze v rámci jedné úlohy spolehnout na sdílená data (například filesystem). Není tomu tak ale mezi jednotlivými úlohami, tedy v rámci jednoho workflow se jednotlivé úlohy spustí v odlišných prostředích. Při potřebě předat výsledné artefakty z jedné dokončené úlohy do další je možné tyto artefakty vyspecifikovat pomocí konfiguračního pole `outputs`, takto uvedené artefakty jsou dále přístupné všem úlohám, které jsou na této závislé. Konfigurace, v jakém prostředí runner serveru je úloha spuštěna, se provádí pro každou úlohu zvlášť položkou `runs-on`. [27]

Ukázka konfigurace sekvenčního spuštění úloh, `job2` se spustí až po dokončení `job1` kvůli řádku 14, zároveň díky řádkům 8–9 bude mít úloha `job2` přístup k `output1`, dále je vidět konfigurace prostředí na řádcích 7 a 13.

```
 6      job1:
 7          runs-on: ubuntu-latest
 8          outputs:
 9            output1: ...
10          steps:
11              ...
12      job2:
13          runs-on: ubuntu-latest
14          needs: job1
15          steps:
16              - run: echo ${{needs.job1.outputs.output1}}
```

Výpis kódu 3.3 Ukázka konfigurace GitHub Workflow – 2

Úlohy jsou složeny z tzv. *steps* (dále jen kroky), jde o nejmenší jednotky práce prováděné v celém workflow. Jedním krokem může být buď terminálový příkaz (např. řádek 16 v ukázce výše), nebo tzv. *action* (snadno zaměnitelný termín s GitHub Actions), což jsou předem připravené samostatné úlohy, které je možné vkládat jako kroky. Actions mohou být veřejně zpřístupněné na GitHub Marketplace [28], je tak možné jednoduše vkládat předem připravené kroky, například pro nahrání artefaktů na hosting, bez nutnosti znát všechny kroky k tomu potřebné.

Jako kroky pro úlohy se nejčastěji používají příkazy pro nástroje sestavení aplikace, například pro JavaScript aplikace bude většina kroků sestávat z příkazů pro *NPM* nebo *Yarn*.

3.3 Testování uživatelských rozhraní

V této kapitole se zabývám testováním uživatelských rozhraní. Nejprve rozebírám základní pojmy spojené s uživatelskými rozhraními, dále představuji přístupy k jejich tvorbě a z nich vycházející principy testování těchto rozhraní. Kapitola čerpá obecně z [29].

3.3.1 UI/UX

Tato podkapitola čerpá z [30]. Uživatelské rozhraní neboli User interface je soubor technologií a prostředků umožňující uživateli interagovat s aplikací. V oblasti webových aplikací jde zejména o grafické či textové prvky (GUI – Graphical User Interface), které umožňují získávání dat a požadavků od uživatele a dále zobrazení výsledků jejich vyhodnocení.

Moderní webové aplikace k tvorbě uživatelských rozhraní využívají zejména technologií poskytovaných webovými prohlížeči, jako jsou například HTML5, CSS3, JavaScript a dalších. Vzhledem k tomu že jde o základní technologie, často jsou využívány různé frameworky, které zjednodušují a zrychlují tvorbu aplikací, zároveň poskytují určitou jistotu přenositelnosti aplikace mezi jednotlivými prohlížeči a jejich verzemi. Větší část uživatelů používá neaktuální verze prohlížečů a ty tak zdaleka nemusí podporovat veškerou funkcionalitu, dále existuje řada rozdílů mezi jednotlivými prohlížeči (a jejich verzemi) v interpretaci dokumentů (například CSS), kvůli kterým se zobrazení aplikace pro různé uživatele může lišit.

UX (User Experience) je obecný termín používaný pro způsob interakce uživatele s UI, zahrnuje například použitelnost, přístupnost a efektivitu užívání grafického rozhraní. Tvorba uživatelských rozhraní cílená na uživatele je označována jako UXD (User Experience Design), jejímž cílem je co nejlepší použitelnost pro uživatele, ale i grafický design aplikace. [30, s. 1]

3.3.2 Nielsenova heuristika

Nielsenova heuristika je soubor deseti obecných pravidel (principů použitelnosti) pro tvorbu grafických rozhraní, vytvořený Jakobem Nielsenem [31]. Obecně lze říct, že jde o jedny z nejčastěji doporučovaných základních pravidel pro tvorbu GUI. Vychází z dlouhodobého testování chování uživatelů při používání aplikací a jde o velmi obecné principy, které se mohou zdát založené na intuici (proto heuristiky). Využitím těchto pravidel bychom se měli vyvarovat zásadním chybám vznikajícím při tvorbě rozhraní.

Pravidla jsou seřazeny popořadě od nejdůležitějšího, a zní následovně:

1. *Visibility of system status* – uživateli by vždy mělo v co nejkratší době být jasné, v jakém stavu se systém nachází. Pravidlo uživatelům zjednodušuje naučení se práci se systémem, systém je předvídatelnější.
2. *Match between system and the real world* – měly by být použity termíny a koncepty, s kterými se uživatel již pravděpodobně setkal, nebo existují v reálném světě. Pravidlo dále urychluje schopnost uživatele naučit se pracovat se systémem, systém je pro uživatele „intuitivnější“.
3. *User control and freedom* – uživatelé často dělají chyby, vždy musí existovat jasný a jednoduchý způsob jak se vrátit do předchozího stavu systému bez zdlouhavých procesů. Díky tomuto pravidlu mají uživatelé více pocit jistoty při práci se systémem.
4. *Consistency and standards* – v systému by měly být využity standardní prvky a termíny. Porušováním pravidla zbytečně znejisťujeme uživatele, který má určité očekávání z práce s obdobnými systémy.
5. *Error prevention* – kvalitní chybové hlášky jsou důležité, předcházení chybám je lepší. Pokud nelze možnosti chyby předejít, indikujeme možnost uživateli před jejím vznikem (například nutností potvrdit akce).

6. *Recognition rather than recall* – pokud je to možné, ukazovat podstatné informace nebo popisky spíše, než nutit uživatele vzpomenout si.
7. *Flexibility and efficiency of use* – povolit uživatelům zkratky použití systému tak, aby zkušenější uživatelé měli možnost pracovat efektivněji a nezkušení uživatelé nebyli zahlceni možnostmi.
8. *Aesthetic and minimalist design* – grafické rozhraní by mělo obsahovat jen podstatné prvky a informace. Zahlcení uživatele nepodstatnými informacemi zesložiťuje orientaci v systému.
9. *Help users recognize, diagnose, and recover from errors* – chybové hlášky by měly být snadno čitelné, jasně identifikovat problém a ideálně navrhnout uživateli možnost řešení.
10. *Help and documentation* – pokud je nutné poskytnout uživateli dokumentaci k pomoci s řešením problémů, měla by být snadno přístupná a čitelná.

Pravidla Nielsenovy heuristiky jsou velmi důležitá pro tvorbu uživatelských rozhraní a v zásadě jsou základním stavebním kamenem při testování aplikací, zejména pro heuristický průchod aplikací.

3.3.3 Heuristický průchod

Heuristický průchod je způsob testování aplikace (obecně nějakého *artefaktu*), při kterém se *expert* nebo experti snaží dosáhnout předem stanovených uživatelských *cílů* pomocí akcí, a během průchodu aplikací se snaží identifikovat její nedostatky na základě sady heuristik (často například právě Nielsenovy heuristiky).

Při identifikaci problému v systému (negativní odpověď na to, jestli systém splňuje danou heuristiku) se expert snaží zjistit důvod porušení dané heuristiky. Výsledkem heuristického průchodu aplikací by měl být seznam problematických míst aplikace, porušujících nějaké z daných heuristik a případný důvod jejich porušení.

Zásadními slabinami testování aplikací heuristickým průchodem jsou nutnost, aby expert byl schopen „vžít se“ do role uživatele a aby se oprostil od subjektivního hodnocení systému. Z těchto důvodů testování nezjistí chování reálných uživatelů, a zdaleka ne uživatelů v podmínkách, ve kterých bude systém opravdu používán, jde spíše o identifikování zásadních problémů použitelnosti systému. [32]

3.3.4 Uživatelské testování použitelnosti

Použitelnost je termínem pro míru jednoduchosti použití a naučitelnosti práce se systémem, použitelností lze označit kvalitu interakce uživatele se systémem.

Cílem testování použitelnosti je tedy identifikace problémových míst systému, kvalifikace do jaké míry systém splňuje svůj účel a případně návrhy úprav pro zlepšení použitelnosti systému. Testy mohou být prováděny na prototypech, rozpracovaných aplikacích ale i produkčních nasazeních aplikací.

Zjednodušeným postupem uživatelského testování použitelnosti systému je:

1. Uživateli předložíme nějaký cíl, kterého má v aplikaci dosáhnout.
2. Sledujeme uživatele při práci se systémem a snažíme se identifikovat překážky.
3. Identifikujeme problémová místa a navrhneme z nich vycházející změny systému.

Cíl zmíněný v prvním bodu postupu je často reprezentován *scénářem*. Tento scénář je tvořen seznamem realistických úkolů, které mají za úkol plně otestovat požadovanou funkčnost systému. Scénář je většinou ve formě jednoduchého jazyka, úkoly jako například „Najdi informace o konkrétním produktu“. Výstupem uživatelského testování by nemělo být pouze naše pozorování, ale i data převeditelná do statistik, například z vyplňovaných dotazníků. [29]

Kapitola 4
Praktická část

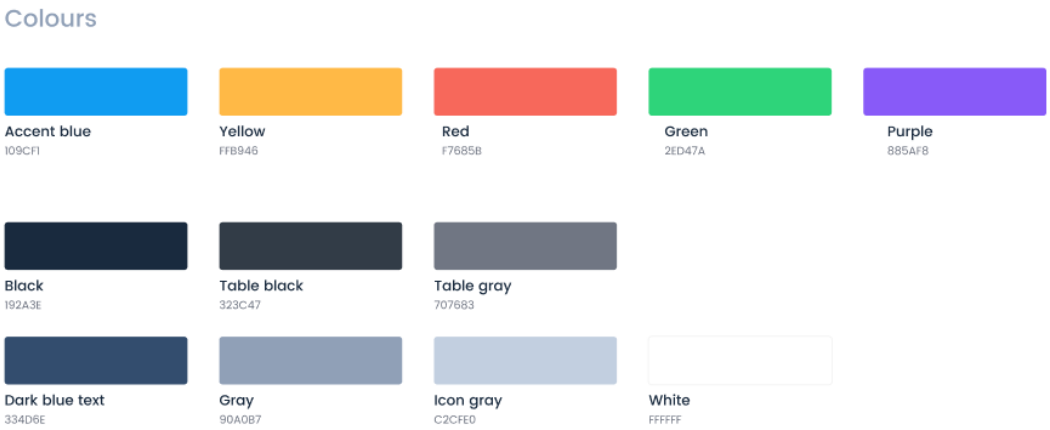
4.1 Analýza dodaných návrhů, serverové části aplikace a API

V této kapitole provádím analýzu dodaných návrhů a identifikuji požadavky na funkcionalitu aplikace a jednotlivé obrazovky. Dále pak popisuji stávající serverovou část systému a zejména API určené pro administrační rozhraní.

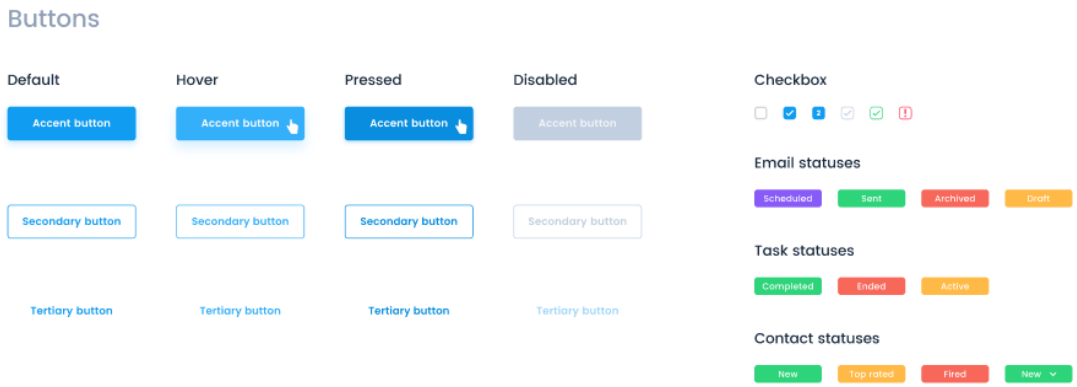
4.1.1 Analýza dodaných návrhů a požadavků na funkcionalitu

Všechny grafické návrhy byly v průběhu vývoje dodány ve webovém grafickém nástroji Figma [33]. Takto jsem získal design pro hlavní z obrazovek, některé, jako například obrazovka nové objednávky, ale byly vytvářeny iterativně během jejich implementace na základě zpětné vazby od zadavatelů.

Dále byl dodán částečný designový manuál pro některé z prvků vyskytujících se na více místech v aplikaci. Například šlo o vzhled tlačítek, definice barevnosti aplikace, typografie, odznaky stavů apod., dále přikládám ukázku designového manuálu.



Obrázek 4.1 Dodaný návrh barevnosti aplikace



Obrázek 4.2 Dodaný návrh tlačítek a odznaků k použití v aplikaci

Obecné návrhy z designového manuálu, jako jsou barevnost a typografie, jsem zakomponoval do projektu pomocí konfigurace Tailwind CSS. Tedy v konfiguračním souboru `tailwind.config.js` jsem nastavil například následující. Nastavení `fontFamily` přepne v celém Tailwind font na specifikovaný, dále jsem pomocí `extend` a `colors` nastavil pro použití všechny požadované barvy.

```
24    fontFamily: {
25      sans: "'Inter', sans-serif",
26    },
27    extend: {
28      colors: {
29        primary: {
30          default: '#109CF1',
...
33        }
34        yellow: '#FFB946',
...
```

Výpis kódu 4.1 Nastavení typografie a barevnosti v Tailwind CSS

Díky konfiguraci výše pak při sestavení aplikace Tailwind CSS vygeneruje pomocné CSS třídy s požadovanými barvami, přesně dle konfigurace. Takže dále jsem mohl v React komponentech používat CSS třídy pro stylování barevnosti jako například `text-primary`, nebo `bg-yellow`, které použijí právě barvu z konfigurace.

Následně jsem pro prvky z manuálu, jako tlačítka a odznaky stavu, vytvořil vlastní React komponenty. Většina z těchto se nachází v adresáři `src/components/ui/`, příkladem může být třeba právě zelený odznak pro úspěšný nebo validní stav. Jde pouze o obalový blok nějakého textu, z ukázky zdrojového kódu dále je zřetelné použití tříd generovaných Tailwind CSS na řádku 17.

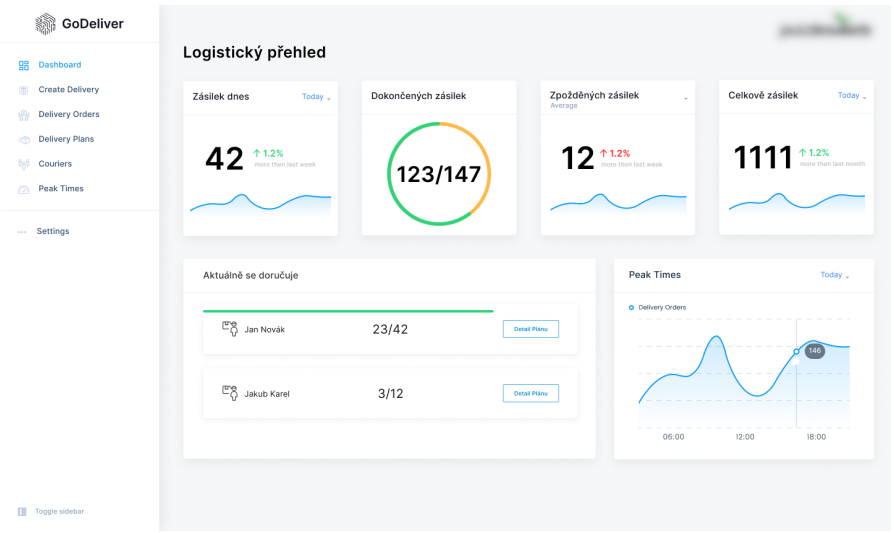
```
13    const StatusSuccess: React.FC<{
14      text: string
15    }> = ({ text }) => {
16      return (
17        <div className="text-center w-24 inline-block font-semibold
                          text-white text-xs bg-green rounded py-1 px-2
                          text-md">
18          {text}
19        </div>
20      )
21    }
```

Výpis kódu 4.2 Použití tříd Tailwind CSS

Obrazovky aplikace a funkcionalita

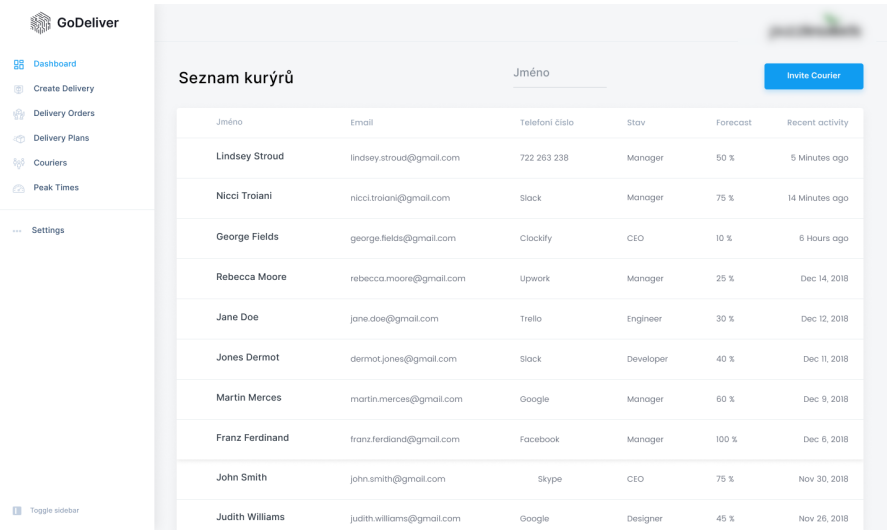
Dále jsem obdržel návrhy obrazovek. Některé z návrhů byly ve finálním stavu, jiné byly pouze rámcové nebo rozpracované.

Návrh úvodní obrazovky byl následovný a je z něj dobře identifikovatelné, k čemu má obrazovka sloužit. Horní řada bloků a blok vpravo dole poskytuje informace a statistiky o stavu a počtu zásilek doručovaných v aktuální den. Blok v levém dolním rohu pak ukazuje momentálně probíhající plány, s odkazem na jejich detail. V levé části obrazovky je potom menu s odkazy na všechny stránky aplikace, toto menu se opakuje na všech obrazovkách.



Obrázek 4.3 Návrh úvodní obrazovky

Dalším dodaným návrhem je návrh obrazovky přehledu kurýrů 4.4. Hlavním prvkem této obrazovky je tabulka kurýrů, zobrazující zejména jejich kontaktní informace. Dále je na obrazovce vpravo nahoře možnost přizvat nového kurýra zadáním jeho e-mailu.

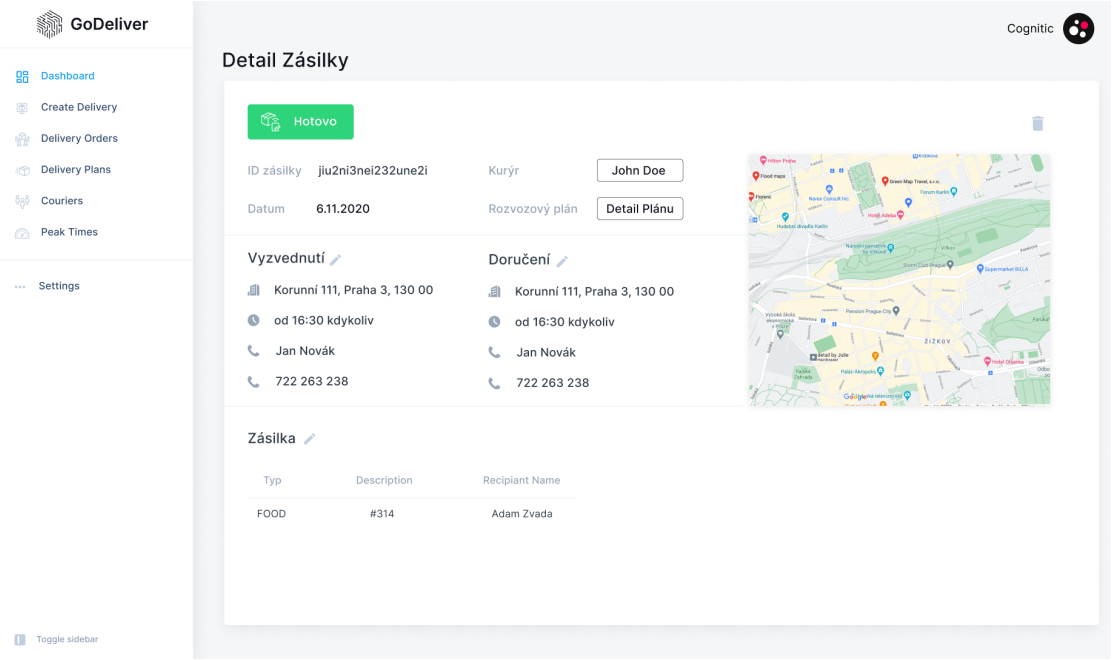


Obrázek 4.4 Návrh přehledu kurýrů

Tento návrh nesloužil pouze pro obrazovku přehledu kurýrů, ale také pro obrazovky přehledů zásilek a rozvozových plánů. Zmíněné obrazovky jsou totiž prakticky stejné, dále toto popíšu v implementaci obrazovek.

Dalším dodaným materiálem byly návrhy detailu zásilky a její editace. První z těchto návrhů slouží také pro detail rozvozového plánu, dále popsané v implementaci stránek.

Obrazovka tedy ukazuje základní údaje o zásilce, možnost přejít na přidělený rozvozový plán a mapu s označenými body vyzvednutí a doručení. Dále je pak v pravém horním rohu vyznačena možnost zásilky vymazat a u každé informační sekce je tlačítko pro editaci této části zásilky.



Obrázek 4.5 Návrh detailu zásilky

Další snímky návrhů přidávám pouze do přílohy práce, ze zbylých návrhů a požadavků jsem dále identifikoval následující požadavky.

- Přehledy zásilek a plánů, které jsem již zmiňoval.
- Obrazovka pro vytvoření nové zásilky – pro tuto nebyl vytvořen detailní návrh kvůli její komplexitě, dále ji popisuji v kapitole implementace obrazovek.
- Obrazovka nastavení funkce plánovače – tuto také dále popisuji v implementaci a její návrh je na snímku A.7.
- Obrazovka pro přihlášení a registraci – pro tuto obrazovku také nebyl dodán žádný návrh, je poměrně přímočará a nepopisuji ji ani v implementační části, snímky výsledné obrazovky se nachází v příloze A.9.
- Obrazovka diagramu plánů – tato obrazovka zobrazuje diagram časových údajů rozvozových plánů, konkrétně požadované časy vyzvednutí a doručení, a k nim korespondující časy odhadované plánovačem. Tedy na této obrazovce je možné ověřit funkčnost plánovače a zjistit dopředu případné zpoždění zásilek. Návrh obrazovky se nachází v příloze A.8 a snímek implementované obrazovky také A.19.

4.1.2 Analýza API

V této kapitole popisuji entity vyskytující se v systému. API GoDeliver je veřejné a poskytuje specifikaci vygenerovanou ze zdrojových kódů. Rozhraní je klasické REST API používající JSON jako datový formát.

V systému jsem identifikoval následující základní entity. Všechny entity přímo reprezentují reálné společnosti, kurýry a dále.

- Základní entitou v systému je *společnost*. Tato entita reprezentuje reálnou společnost používající systém GoDeliver. Obsahuje základní informace o společnosti jako název apod., dále pak nastavení fungování plánovače. Společnost je pevně spojena s uživatelským účtem vytvořeným ve službě Firebase Authentication.
- Společnost dále obsahuje *kurýry*, tito vždy pracují pouze pro jednu společnost a rozváží pro ni zásilky. Dále tato entita obsahuje základní kontaktní údaje. Společnost muže kurýry přizvat a odstranit. Zároveň platí, že kurýr je ekvivalentní s účtem kurýra v mobilní aplikaci GoDeliver pro kurýry.
- Další entitou vyskytující se v systému je *zásilka*. Reprezentuje reálnou zásilku, která má být někde vyzvednuta a někam doručena. Pro vyzvednutí a doručení obsahuje časové údaje a údaje o kontaktních osobách, dále pak obsah zásilky.
- Poslední zásadní entitou je *rozvozový plán*. Tato entita v zásadě spojuje zásilky a kurýry. Plán má vždy přiřazeného jednoho nebo žádného kurýra, dále obsahuje seznam bodů, ke kterým pak patří buď vyzvednutí nebo předání zásilky, případně několika zásilek. Kurýr se tímto plánem pak řídí při rozvážení zásilek.

Implementace datové vrstvy dále v práci popisuje implementaci datové vrstvy pracující s API a vytvoření TypeScript typů pro tyto entity. Datové typy byly vytvořeny zejména za konzultace zmíněné specifikace API.

4.2 Nástroje pro vývoj a CI/CD pipeline

V této kapitole popisuji konfiguraci a výběr nástrojů používaných pro vývoj aplikace, dále pak nastavení CI/CD a nasazení.

Pro zjednodušení vývoje jsem nastavení všech nástrojů a CI/CD pipeline provedl ihned ze začátku vývoje, dále bylo vše pouze upravováno podle potřeb.

4.2.1 Nástroje pro vývoj

Nastavení npm pro vývoj spočívalo pouze v přidání několika skriptů do `package.json`. Popořadě to jsou tyto skripty:

```
05  "dev": "next dev",
06  "build": "rm -rf ./build && next build",
07  "start": "next start -p $PORT",
08  "type-check": "tsc --pretty --noEmit",
09  "lint": "eslint . --ext ts --ext tsx --ext js",
10  "test": "jest --passWithNoTests",
11  "test-all": "npm run lint && npm run type-check && npm run test",
```

Výpis kódu 4.3 Konfigurace skriptů npm

Řádek 5 spustí vývojový server Next.js, který aplikaci nesestavuje celou, ale pouze aktuálně vyžadované komponenty. Zároveň poskytuje tzv. *Fast Refresh*, díky se změny v kódu okamžitě propisují do běžící aplikace a otevřeného prohlížeče.

Řádky 6 a 7 sestaví produkční verzi a spustí ji na požadovaném portu. Proměnná `$PORT` je zde použita kvůli Google App Engine, který ji používá při vyvažování zátěže nasazené aplikace.

Skript na řádku 8 použije kompilátor TypeScriptu pro statickou kontrolu typů a celkově kódu, přepínač `noEmit` instruuje kompilátor aby nevytvářel žádné kompilované soubory, ale pouze vypsal chybové hlášky. 9. řádek použije ESLint k další statické analýze zdrojových souborů a 10. řádek dále spustí testy pomocí knihovny Jest. Všechny tyto nástroje musely dále být nakonfigurovány pro správnou funkčnost.

Dále jsem ještě v rámci konfigurace npm přidal nástroj Husky, který využívá funkcionalitu Git repozitáře *Git Hooks*. Po nainstalování závislostí pomocí npm se před každým `git commit` spustí ESLint na každý upravený soubor v commitu s příponou ts nebo tsx. Zároveň se před každým `git push` spustí ještě TypeScript kompilátor na celý repozitář. Tímto nástrojem se mi podařilo odstranit vytvoření mnoha zbytečně chybných commitů. Konfigurace vypadá takto:

```
14  "husky": {
15    "hooks": {
16      "pre-commit": "lint-staged",
17      "pre-push": "npm run type-check"
18    }
19  },
20  "lint-staged": {
21    "*.@(ts|tsx)": [
22      "npm run lint"
23    ]
24  }
```

Výpis kódu 4.4 Konfigurace npm pro nástroj Husky

Implementovat aplikaci jsem se rozhodl v jazyku TypeScript, z důvodů zmíněných v teoretické části práce, a obecně pro zjednodušení vývoje a udržitelnosti kódu. Kompilátor TypeScriptu je nutné také nastavit, soubor k tomu určený se jmenuje `tsconfig.json` a použil jsem následující konfiguraci (ukázka pouze zajímavějších nastavení):

```
...
05    "allowJs": true,
06    "checkJs": true,
08    "strict": true,
10    "noEmit": true,
...
```

Výpis kódu 4.5 Ukázka konfigurace kompilátoru TypeScript

Z důvodu využití frameworku Next.js, který při sestavení, a tedy i pro kompilaci zdrojových souborů v TypeScriptu, používá nástroj Babel, je nutné v konfiguraci použít řádek 10. Dále jsem na řádcích 5 a 6 povolil použití souborů v samotném JavaScriptu a zapnul kontrolu i v těchto souborech. Tato nastavení jsou pro Next.js také nutná z toho důvodu, že některé soubory využívané Next.js nemohou být napsané v TypeScriptu (např. `_app.js` a `_document.js`), nicméně kontrolovat je stále chceme.

Nastavení konfigurace samotného frameworku Next.js tu nepopisuji, není nijak zajímavé, až na přepnutí adresáře pro výstup sestavení na `build/`. Next.js totiž ve výchozím stavu aplikaci sestaví do skrytého adresáře, a ten by nebyl přístupný při následném nasazení na Google Cloud (Google Cloud CLI pro účely nasazení skryté soubory ignoruje), nasazení je popsáno dále v kapitole.

Next.js dále při sestavení interně používá nástroje Babel, pro transpilaci kódu a zaručení funkčnosti mezi různými prohlížeči, a Webpack, kvůli slučování kódu do „balíků“ a možnostem například importovat SVG soubory do zdrojových souborů jako kód, nebo importovat CSS soubory.

Dále bych ještě chtěl popsat nastavení linteru ESLint, ve výchozí konfiguraci je totiž tento nástroj určen pouze pro jazyk JavaScript. Obrovská výhoda ESLint je ale právě v tom, že je dobře konfigurovatelný a podporuje použití rozšíření, které dále přidávají funkcionalitu. Z použité konfigurace jsou důležité právě využité rozšíření, a to:

```
...
07  "parser": "@typescript-eslint/parser",
08  "plugins": ["@typescript-eslint"],
09  "extends": [
10    "eslint:recommended",
11    "plugin:react/recommended",
12    "plugin:react-hooks/recommended",
13    "plugin:@typescript-eslint/recommended"
14  ],
...
```

Výpis kódu 4.6 Ukázka konfigurace ESLint

V konfiguraci výše je na řádku 8 přidáno rozšíření pro TypeScript a na řádku 7 přepnut parser na ten, který toto rozšíření poskytuje. Dále jsou pak v položce `extends` přidány nastavení určené pro TypeScript a React. Například rozšíření z řádku 12 přidává různé kontroly týkající se konkrétně React Hooks (popsané v teoretické části práce), jako je třeba kontrola, jestli je při použití hooku `useEffect` v poli závislostí přidáno vše, co je uvnitř funkce využito.

4.2.2 Integrace a nasazení

Jelikož aplikace byla vyvíjena pro firmu, sdílený repozitář mi byl předem vytvořen zadavateli ve službě GitHub. Hosting, na kterém aplikace má být nasazena, byl předem určený (Google App Engine) z toho důvodu, že na něm již byla nasazena serverová část systému a nebyl žádný podstatný důvod pro to, aby vyvíjená aplikace byla hostována jinde.

K vytvoření CI/CD pipeline jsem tedy použil GitHub Actions a vytvořil jsem proto dva workflow soubory, jeden pro vývojovou větev `dev` a druhý pro produkční větev `master`. Oba soubory jsou prakticky stejné, pouze se použije jiný konfigurační soubor pro nasazení na App Engine. Dále popíši konfiguraci pro vývojovou větev.

4.2.2.1 Konfigurace CI/CD

Konfigurační soubor je umístěný v `.github/workflows/main.yml` a začíná poměrně jednoduše pojmenováním a nastavením události pro kterou se spustí, tedy `push` do větve `dev`.

```
01    name: Test and deploy dev version
02
03    on:
04      push:
05        branches:
06          - dev
...
```

Výpis kódu 4.7 Nastavení CI/CD ve službe GitHub – 1

Ve workflow je pouze jedna úloha a to `build`, spouští se v prostředí `ubuntu-latest`.

```
...
08    jobs:
09      build:
10        runs-on: ubuntu-latest
11        steps:
...
```

Výpis kódu 4.8 Nastavení CI/CD ve službe GitHub – 2

Dále pak pokračují jednotlivé kroky úlohy. Následující použije akci `checkout@v2`, která použije doslova `git checkout` k zpřístupnění obsahu repozitáře do prostředí běžící úlohy.

```
...
13          - uses: actions/checkout@v2
...
```

Výpis kódu 4.9 Nastavení CI/CD ve službe GitHub – 3

Krok na další stránce jsem použil pro zrychlení celého workflow a sestavení aplikace. Krok použije akci `cache@v2`, která zpřístupňuje cache. Dále krok vytvoří klíč se zahešovaným obsahem `package-lock.json`. Ve výsledku to znamená regeneraci cache npm v případě, že nebyly změněny závislosti.

```
...
15      - name: Cache Node modules
16        uses: actions/cache@v2
17        env:
18          cache-name: cache-node-modules
19        with:
20          path: ~/.npm
21          key: ${{ runner.os }}-build-${{ env.cache-name }}-\\
22               ${{ hashFiles('**/package-lock.json') }}
23          restore-keys:
...
```

Výpis kódu 4.10 Nastavení CI/CD ve službe GitHub – 4

Poté jsou nainstalovány závislosti a spuštěny všechny testy (ESLint, TypeScript, Jest).

```
...
27      - name: Install dependencies
28        run: npm ci
29
30      - name: Run all tests
31        run: npm run test-all
...
```

Výpis kódu 4.11 Nastavení CI/CD ve službě GitHub – 5

Následně je překopírován soubor s proměnnými prostředí pro nasazení.

```
...
33      - name: Using dev environment
34        run: cp environments/.env.development .env.local
...
```

Výpis kódu 4.12 Nastavení CI/CD ve službe GitHub – 6

Dále je vytvořena cache pro cache sestavení Next.js (adresář `.next/cache`) a je sestavena aplikace. Aplikace je zde sestavena kvůli tomu, že sestavení může selhat i přes předchozí testování a následné nasazení na App Engine se sestavením trvá poměrně dlouho. Výsledek sestavení níže je pak odeslán na hosting.

```
...
42      - name: Try building app
43        run: npm run build
...
```

Výpis kódu 4.13 Nastavení CI/CD ve službe GitHub – 7

Nakonec je v další ukázce provedeno nasazení. Spočívá v nastavení CLI nástroje pro Google Cloud a následně jeho použití pro nasazení s konfiguračním souborem pro požadovaný *service*.

```
...
45      - name: Set up gcloud CLI
46        uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
47        with:
48          project_id: ${{ secrets.GCP_PROJECT_ID }}
49          service_account_key: ${{ secrets.GCP_SA_KEY }}
50
51      - name: Deploy to GCP
52        run: gcloud app deploy --quiet --stop-previous-version app.dev.yml
...
```

Výpis kódu 4.14 Nastavení CI/CD ve službe GitHub – 8

4.2.2.2 Konfigurace nasazení na Google App Engine

Konfigurační soubor pro nasazení vývojové verze na Google App Engine je pojmenován `app.dev.yml`. Použil jsem standardní prostředí, Node.js verze 12, a zpracování požadavků je ponecháno na aplikaci. Soubor vypadá následovně, pro produkční verzi je soubor stejný, kromě jména služby a pojmenování souboru:

```
01      env: standard
02      runtime: nodejs12
03      service: web-client-dev
04
05      handlers:
06        - url: /.*
07          secure: always
08          script: auto
```

Výpis kódu 4.15 Nastavení nasazení Google App Engine pomocí app.dev.yml

Další důležitou konfigurací pro nasazení v Google Cloud je soubor `.gcloudignore`, jelikož Google Cloud CLI ve výchozím nastavení nahraje do služby celý repozitář kromě skrytých souborů, tedy včetně celého `node_modules/` obsahující veškeré soubory knihoven a závislostí ze sestavení výše. Jelikož je aplikace sestavena v CI/CD, není nutné ji dále sestavovat v App Engine, a tak se výsledek sestavení nahraje na hosting. Není tedy poté potřebné nahrávat zdrojové soubory, soubory knihoven a testy.

```
01    /node_modules/
02    /src/
03    /tests/
```

Výpis kódu 4.16 Použití .gcloudignore

4.3 Implementace aplikace

V této kapitole popisuji postupnou implementaci aplikace a všech jejích komponent. V průběhu implementace došlo k přepsání některých už vytvořených částí projektu, zejména například celého způsobu, jakým jsou do aplikace zavedena data z API. Z celého procesu popíšu pouze finální verzi aplikace.

4.3.1 Adresářová struktura projektu

Adresářová struktura celého projektu je v zásadě jednoduchá, do určité míry opisuje strukturu aplikace jako takové. Je následovná:

```
├── … ..... veškeré další konfigurační soubory nástrojů
├── package.json ..... konfigurace projektu
├── .github/workflows/ ..... konfigurace CI/CD
├── environments/ ..... soubory s proměnnými prostředí
├── public/ ..... statické soubory (konkrétně obrázky a překlady)
└── src ..... zdrojové kódy aplikace
    ├── @types/ ..... deklarace typů pro TypeScript kompilátor
    ├── assets/ ..... soubory importované do kódu (zejména SVG ikony)
    ├── css/ ..... CSS dokumenty rozšiřující Tailwind.css
    ├── pages/ ..... adresář pro komponenty obrazovky
    ├── components/ ..... podkomponenty obrazovek
    └── utils/
        ├── api/ ..... asynchronní funkce získávání dat z API, definice typů entit
        └── helpers.ts ..... pomocné funkce
```

V této kapitole budu rozepisovat pouze obsah adresáře `src` a to konkrétně podadresáře `pages`, `components` a `utils`.

4.3.2 Implementace datové vrstvy aplikace

Datovou vrstvu jsem implementoval ve dvou místech projektu, prvním z nich je adresář `src/utils/api/`, který dále obsahuje soubory pro jednotlivé kategorie API. Tyto soubory obsahují asynchronní funkce pro získávání dat z endpointů API, dále soubory obsahují definice TypeScript typů dat získávaných pomocí těchto funkcí.

```
api/
├── business.ts ..... data související se společnostmi
├── couriers.ts ..... data související s kurýry
├── dashboard.ts ..... data pro úvodní obrazovku
├── deliveryPlans.ts ..... data související s rozvozovými plány
└── order.ts ..... data související se zásilkami
```

Pro příklad popíšu implementaci funkce pro získání detailu konkrétní zásilky. Nejprve jsem vytvořil datový typ zásilky následovně:

```
02    export interface IOrder {
03      id: string
04      external_id: string | null
05      delivery: {
06        pickup_address: IAddress
07        destination_address: IAddress
08        pickup_contact: IContact
09        destination_contact: IContact
10        packages: IPackage[]
11        delivery_time: {
12        ...
13        }
14      }
15      business_id: string
16      status: EOrderStatus
17      priority: number
18      courier_id: string
19      delivery_plan_id: string
20      date: string
21    }
```

Výpis kódu 4.17 Vytvoření datového typu zásilky

Datový typ popisuje zásilku tak, jak je získána z API, ale zároveň je ve tvaru požadovaném pro odeslání na API, například při vytváření nové zásilky. Zásilka je dále složena z jiných typů, které se v datové vrstvě opakují i jinde (`IAddress`, `IContact`, `IPackage` a další).

Tvar všech takto vytvářených datových typů jsem získával a ověřoval přímo z API, na kterém je použit nástroj Swagger, který automaticky vytváří dokumentaci API ve formě webové stránky (dostupné z https://api.godeliver.co/swagger).

Do určité míry je škoda, že nebyly možnosti tohoto nástroje využity naplno, protože je s jeho pomocí možné z dobře dokumentované specifikace API automaticky generovat knihovnu pro klienty v různých programovacích jazycích. Tedy při využití této funkcionality pro TypeScript je vygenerována knihovna obsahující definice všech typů vyskytujících se v API a zároveň jsou vytvořeny funkce pro provolávání endpointů. Jelikož tato funkcionalita použita nebyla, bylo nutné vše vytvářet a udržovat ručně.

Dále jsem tedy vytvořil asynchronní funkci pro získání dat z endpointu API takto:

```
01    export async function getOrderDetails(
02      props: Record<string, unknown>
03    ): Promise<IGetOrderDetailsResponse> {
04      const { businessId, orderId } = props
05      const url = `${baseUrl}/order/${businessId}/${orderId}`
06
07      return await axios
08        .get(url)
09        .then((res) => res.data)
10        .catch((_e) => {
11          return { status: 'error' }
12        })
13    }
```

Výpis kódu 4.18 Asynchronní funkce datové vrstvy

Typ argumentů funkce je z mého pohledu nehezký, ale bohužel tímto způsobem asynchronním funkcím předává argumenty knihovna React Async, která v době implementace postupně přecházela na TypeScript. Funkce tedy rozloží argumenty, poskládá z nich URL endpointu a pomocí knihovny Axios odešle GET požadavek a vrátí data, respektive *promise*, tedy. Typ vrácených dat `IGetOrderDetailsResponse` je pouze obalením typu `IOrder` s přidaným polem `status`, které značí ne/úspěch požadavku a případně typ chyby.

Kromě těchto asynchronních funkcí dále soubory obsahují ještě případné funkce pro transformaci dat z formulářů na DTO, ve výsledku má například samotný soubor `order.ts` téměř šest set řádků.

Jak už jsem psal v úvodu kapitoly, jde pouze o první část datové vrstvy aplikace, další část spočívá v datových *hooks*, ze kterých jsou zpřístupněny data komponentům. Asynchronní funkce zde vytvořené nejsou v komponentech používány až na výjimky explicitně, i z tohoto důvodu není až takový problém, že rozhraní funkcí nespecifikuje konkrétní typy argumentů.

4.3.2.1 Datové React hooks

Na začátku kapitoly implementace jsem zmínil, že jsem tuto část projektu v průběhu vývoje musel přepsat. Původně, dokud měl projekt menší rozsah, jsem pro získávání dat v komponentech používal jednoduché volání výše popsaných asynchronních funkcí, pomocí knihovny React Async.

Tento přístup byl zpočátku vývoje v pořádku, data byla získána v komponentu obrazovky a dále předávána vnořeným komponentům. S přibývajícími úrovněmi vnořených komponentů ale jakákoliv změna v přístupu k datům znamenala nutnost měnit kód ve více komponentech na různých úrovních aplikace. Další obtíží při tomto přístupu je například obnovení dat z komponentu na nižší úrovni – je nutné kromě dat předávat i funkci která nám toto umožní, tedy dále se zesložiťuje interface komponentů.

Kvůli všem zmíněným důvodům jsem tuto část aplikace přepsal takovým způsobem, aby byl využit React Context (použití vysvětleno v teoretické části), a vytvořil jsem pro data několik React hooks. Soubory s nimi se nacházejí v adresáři `src/components/hooks/`.

```
hooks/
├── … ..... další hooks nesouvisející s datovou vrstvou
├── useBusiness.ts ..... hook pro data společnosti
└── useData.ts ..... hook pro všechna ostatní data
```

Získávání dat o společnosti jsem oddělil od ostatních dat z toho důvodu, že musí být získány dříve, aby ostatní data mohly být vůbec získány, a zároveň jsou globální pro celou aplikaci, po přihlášení do aplikace vlastně nikdy není důvod tato data obnovovat kromě odhlášení.

Dále popíši po částech pouze hook `useData`, hook pro data o společnosti je prakticky stejný. Nejdříve bylo nutné vytvořit datový typ pro data poskytovaná přes hook:

```
24    interface IDataContext {
...
42      ordersData: GetOrdersResponse | undefined
43      ordersError: Error | undefined
44      ordersLoading: boolean
45      ordersReload: () => void
46      ordersDate: string
47      setOrdersDate: Dispatch<SetStateAction<string>>
...
55    }
```

Výpis kódu 4.19 Definice typu interface datového hooku

Na řádcích 42–45 jsou položky získané použitím knihovny React Async, řádky 46 a 47 jsou pro udržení informace o filtru, který byl naposledy použit na obrazovce přehledu zásilek. Interface

také obsahuje obdobné položky pro data KPI, kurýrů, plánů a diagramu plánů. Poté jsem vytvořil kontext:

```
57    const dataContext = createContext<IDataContext>({} as IDataContext)
```

Výpis kódu 4.20 Vytvoření React Context datového hooku

Následně jsem vytvořil komponent pro provider, který pak poskytuje ostatním komponentům data, nejdříve však využije hook `useBusiness` pro získání dat společnosti.

```
59    const DataProvider: React.FC = ({ children }) => {
60      const { businessData } = useBusiness()
...
```

Výpis kódu 4.21 Vytvoření komponentu provideru datového hooku – 1

Dále je v komponentu použit hook `useState` pro filtrování dat o zásilkách a dále jsou získány požadovaná data, identicky pro ostatní data. Argument `watch` na řádku 114 zajišťuje, že se obalená asynchronní funkce (řádek 110) znovu zavolá při jeho změně. Hook `useAsync` pak vrátí samotná data, případnou chybu, příznak jestli aktuálně probíhá získání dat a funkci pro explicitní spuštění.

```
102   const [ordersDate, setOrdersDate] = useState<string>(getTodaysDate())
103
104   const {
105     data: ordersData,
106     error: ordersError,
107     isLoading: ordersLoading,
108     reload: ordersReload,
109   } = useAsync<GetOrdersResponse>({
110     promiseFn: getOrders,
111     businessId: businessData?.business?.id,
112     status: null,
113     date: ordersDate,
114     watch: ordersDate,
115   })
...
```

Výpis kódu 4.22 Vytvoření komponentu provideru datového hooku – 2

Komponent končí vrácením provideru vytvořeného kontextu s vloženými daty, zároveň obaluje do něj vnořené komponenty (řádek 167). Takto obalené komponenty pak mohou využít interní hook Reactu `useContext` pro získání poskytovaných dat vytvořeného kontextu.

```
132    return (
133        <dataContext.Provider
134          value={{
...
152            ordersData,
153            ordersError,
154            ordersLoading,
155            ordersReload,
156            ordersDate,
157            setOrdersDate,
...
165          }}
166        >
167          {children}
168        </dataContext.Provider>
169      )
```

Výpis kódu 4.23 Vytvoření komponentu provideru datového hooku – 3

Pro další práci s kontextem je ještě z modulu exportován hook `useData`, jde čistě o vytvoření příjemnějšího rozhraní.

```
172    const useData = (): IDataContext => {
173      return useContext<IDataContext>(dataContext)
174    }
...
176    export { DataProvider }
177    export default useData
```

Výpis kódu 4.24 Vytvoření komponentu provideru datového hooku – 4

Hook je takto dokončen, nicméně je nutné někde ve stromě komponentů použít `DataProvider`, aby byly zpřístupněny data. Toto je ideální provést v souboru `src/pages/_app.js`, kterým Next.js obaluje všechny obrazovky. Využil jsem ho pro tento účel u všech vytvořených hooks.

```
19    const App = ({ Component, pageProps }) => {
23      return (
32                  <BusinessProvider>
33                    <DataProvider>
...                     ...
37                    </DataProvider>
38                  </BusinessProvider>
44      )
45    }
```

Výpis kódu 4.25 Vytvoření komponentu provideru datového hooku – 5

Hlavní výhodou tohoto přístupu k poskytování dat v aplikaci je, že následně můžeme data získat v jakémkoliv komponentu tímto způsobem (příklad z komponentu obrazovky přehledu zásilek).

```
52      const {
53        ordersData: data,
54        ordersError: error,
55        ordersLoading: isLoading,
56        ordersReload: reload,
57        ordersDate,
58        setOrdersDate,
59      } = useData()
```

Výpis kódu 4.26 Vytvoření datového hooku – použití

4.3.3 Implementace obrazovek

V této kapitole se věnuji implementaci jednotlivých obrazovek aplikace a jejich funkcionalitě. Zdrojové soubory obrazovek jsem rozdělil vždy na základní komponent obrazovky v adresáři `src/pages/` a dále na vnořené komponenty v podadresáři `src/components/` se stejným názvem jako konkrétní obrazovka. Zdrojové soubory jsou takto logicky rozděleny dle struktury aplikace a orientace v nich se tím poměrně dost zjednodušila.

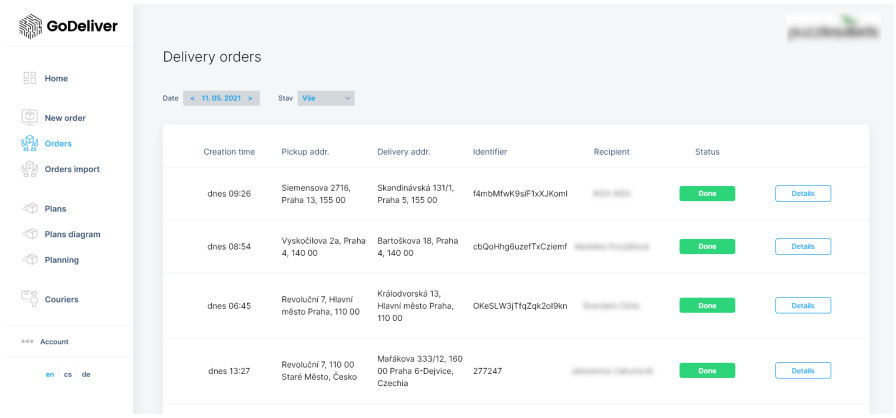
4.3.3.1 Obrazovky přehledů zásilek, plánů a kurýrů

Tyto tři obrazovky jsou velmi podobné a liší se pouze v drobných detailech a datech která zobrazují. Na každé z těchto obrazovek je hlavní částí stránky tabulka se zobrazením primárních informací o daných položkách, například pro zásilky jde zejména o čas vytvoření, adresy vyzvednutí a doručení, stav zásilky a jméno příjemce.

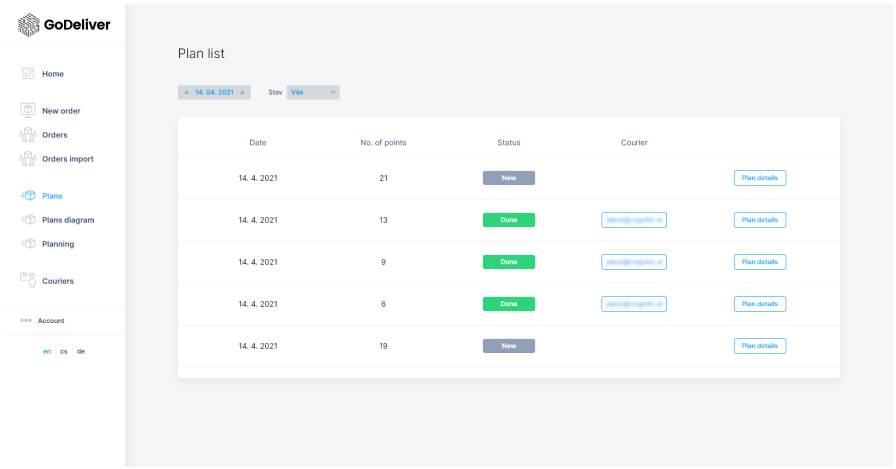
U tabulek jsou přidány možnosti filtrování obsahu, filtr pro datum využívá funkcionalitu popsanou výše v implementaci `useData`, tedy při zvolení dne jsou stahovány položky pouze přímo odpovídající tomuto datu, zároveň při přechodu na jinou stránku si aplikace toto datum pamatuje.

Dále je u zásilek a plánů u každé položky tabulky tlačítko jako odkaz na stránku detailu konkrétní položky. Na obrazovce kurýrů jsem pak ještě přidal možnost odeslání pozvánky pro nového kurýra.

Pro vytvoření tabulek jsem použil knihovnu React Table, a vytvořil jsem generickou tabulku jako komponent v souboru `src/components/table/Table.tsx`. Tento komponent je poté ve stránce vykreslen s předanou definicí sloupců a daty.



Obrázek 4.6 Výsledná obrazovka zásilek



Obrázek 4.7 Výsledná obrazovka plánů

4.3.3.2 Obrazovka vytvoření nové objednávky

Tato obrazovka je v podstatě jedním velkým formulářem. Ačkoliv neměla být používána nijak často a slouží prakticky pouze jako záloha, jelikož objednávky jsou typicky vytvářené v systému společnosti a vkládány přes API, zabrala na implementaci téměř nejvíce času, zároveň jde také o největší komponent celého projektu.

Velkou porci rozsahu implementace této obrazovky tvořila validace formuláře, pro tento účel byla použita knihovna Yup. Konfiguraci validace jsem provedl vytvořením následně vypadajícího schématu. Pro ukázku jsem vybral část která kontroluje správnost dat balíků zásilky. Stručně řečeno popořadě schéma zaručí, že ve výsledku formuláře je položka `packages`, která je polem objektů s položkami `deliveryType` a `deliveryDescription`, a že tato položka je alespoň jedna.

```
01  const schema = useMemo(
02    () =>
03      yup.object().shape({
04        packages: yup
05          .array()
06          .of(
07            yup.object().shape({
08              deliveryType: yup
09                .string()
10                .typeError(t('validation:text'))
11                .required(t('validation:required')),
12              deliveryDescription: yup
13                .string()
14                .typeError(t('validation:text'))
15                .required(t('validation:required')),
16            })
17          )
18          .min(1, t('validation:at_least_one_package')),
...
```

Výpis kódu 4.27 Vytvoření schématu pro validaci knihovnou Yup

Formulář jako takový jsem vytvořil za pomoci knihovny React Hook Form, která umožňuje sjednotit chování jednoduchých HTML Input elementů a zároveň již předpřipravených kompo-

nentů, jako je například React Datepicker, použitý pro volbu data a času vyzvednutí a doručení.

Použití knihovny spočívá ve využití hooku `useForm`, kterému jsou jako argumenty předány výše popsané schéma validace a další nastavení požadovaného chování. Tímto jsou získány funkce (`register` a `control`) které při předání komponentům pro vstup umožňují knihovně odchytávat změny hodnot vstupů.

Pro všechny jednoduché vstupy jsem dále vytvořil vlastní obalující komponenty kvůli nastylování, zobrazení chybových hlášek a sjednocení zobrazení, nachází se v adresáři `src/components/form/`. Bez rozdělování komponentů tímto způsobem by už tak dlouhý zdrojový soubor obrazovky narostl na několikanásobek aktuální velikosti. Tyto předem vytvořené komponenty jsem pak ve formuláři použil následovně:

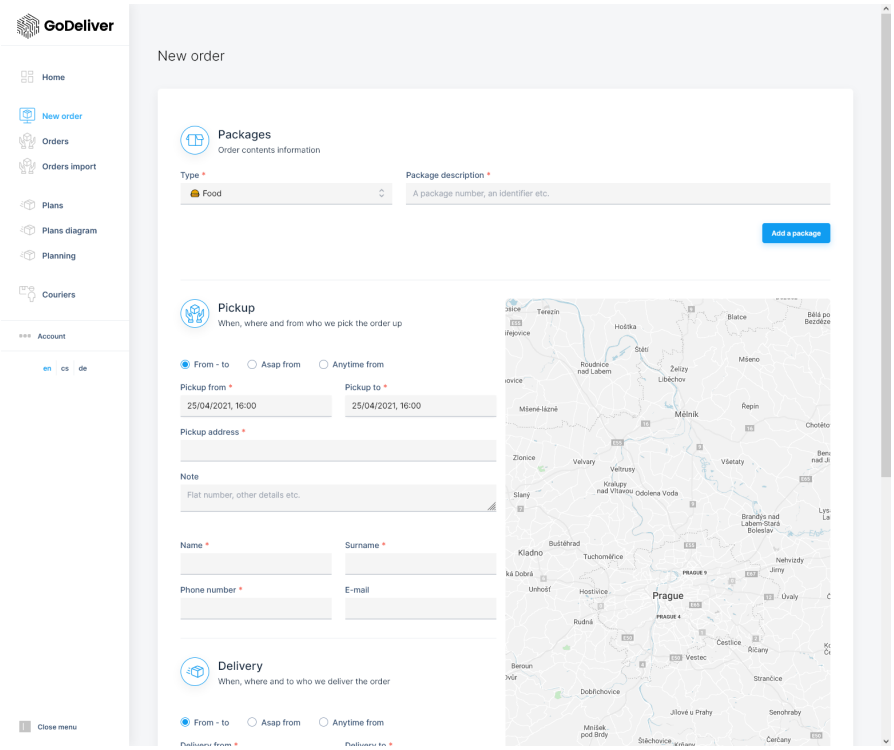
```
01    <Input
02      name="pickupName"
03      label={t('name')}
04      required
05      register={register}
06      error={errors.pickupName}
07      defaultValue=""
08    />
```

Výpis kódu 4.28 Použití knihovny React Hook Form

Dále bych u této obrazovky ještě chtěl zmínit validaci adres. Pro tento účel jsem využil Google Geocoding API, uživatel tak při vypisování adresy průběžně dostává nápovědu validních adres (dle Google Maps). Následně je při validaci formuláře odeslána adresa na API, zpět se vrátí odhadovaná adresa, rozložená na komponenty, společně s GPS souřadnicemi. Zkontrolováním komponent adresy pak odhaduji, jestli je adresa dostatečně přesná. K této kontrole a získání adres jsem napsal funkci `getFormattedAddress`, nachází se v souboru `src/utils/helpers.ts`.

Na obrazovce nové objednávky se ještě nachází mapka (Google Maps), na které se při zadání adres vykreslí podle GPS souřadnic zadané body vyzvednutí a doručení, pro ověření, že zadané údaje jsou správné. Nejde pouze o zkrášlení jinak nudného formuláře, ale zejména o kontrolu, jelikož výše zmíněná validace adres je dosti nepřesná.

Google Geocoding API totiž nijak nekontroluje, jestli jsou adresy opravdu validní, ale pouze vrátí odhadované místo dle textové adresy. Komponenty vracené z API zároveň nejsou konzistentní pro různé země, a tak je jejich kontrola ručně velmi obtížná. Pro tento účel bych Google Geocoding API dále nepoužíval a silně bych doporučil v případě potřeby najít pro validaci adres jinou službu.



Obrázek 4.8 Výsledná obrazovka nové objednávky

4.3.3.3 Obrazovka detailu plánu

Tato obrazovka je určena pro zobrazení detailních informací o konkrétním plánu, vytvořeným systémem. Slouží primárně pro průběžnou kontrolu plánu, zejména v případě, že došlo k nějakému problému, jako nedoručení některé ze zásilek kurýrem.

Z informací na obrazovce jsou zásadní zejména přiřazený kurýr a kontakt na něj, jednotlivé body plánu a mapka s vykreslenými body plánu, sloužící dále pro ověření, že vytvořený plán dává smysl.

Zdrojový soubor komponentu obrazovky jsem pojmenoval `src/pages/plan/[id].tsx`, díky frameworku Next.js a jeho automatickému routingu jsou, na obrazovku takto vytvořenou, dále předávány požadavky s URL `/plan/{id}`, kde `id` je parametr dále předaný komponentu obrazovky.

Tímto způsobem jsem pak mohl v komponentu obrazovky odeslat požadavek na API pro získání dat. Dále ale bylo nutné získat i data o kurýrovi přiřazenému k tomuto plánu, tedy bylo nutné na základě získání dat o plánu dále odeslat další požadavek. Použil jsem možnost `onResolve` hooku `useAsync` na řádku 6, která je zavolána při proběhnutí asynchronní funkce z řádku 3. K udržení dat o kurýrovi a probíhajícím požadavku o ně jsem použil hook `useState`, odtud funkce `setCourier` a `setCourierLoading` na řádcích 12, 13 a dalších.

```
01  const { data, isLoading, reload } =
    useAsync<IGetDeliveryPlanDetailsResponse>(
02    {
03      promiseFn: getDeliveryPlanDetails,
04      businessId: businessData?.business.id,
05      planId: id,
06      onResolve: (newData) => {
07        if (newData && newData.delivery_plan.courier_id && businessData) {
08          setCourierLoading(true)
09          getCourierIdentity({
10            courierId: newData.delivery_plan.courier_id,
11          }).then((res) => {
12            setCourierLoading(false)
13            setCourier(res.courier)
14          })
15        } else if (newData && !newData.delivery_plan.courier_id) {
16          setCourierLoading(false)
17          setCourier(null)
18        }
19      },
20    }
21  )
```

Výpis kódu 4.29 Získání dat závislých na jiných datech

Obrazovka také umožňuje kurýra přiřazeného k plánu odstranit a případně přiřadit kurýra jiného. V případě, že již nějaký kurýr je přiřazen, se vyrenderují pouze informace o kurýrovi a tlačítko pro jeho odstranění, které zavolá funkci `postUnassignPlan` z datové vrstvy s předanými ID kurýra a plánu. Pokud ale plán žádného kurýra nemá přiřazeného, je vyrenderován malý „formulář“ s výběrem kurýrů získaných z globálních dat pomocí hooku `useData`.

Dále jsem vytvořil tabulku bodů. V tomto případě jsem nepoužil způsob popsaný výše u přehledů zásilek a plánů, ale využil jsem jen možností CSS Flexbox. Každý řádek tabulky obsahuje základní informace, zajímavější je první sloupec, ve kterém je značka s pořadím bodu plánu, která pak odpovídá zobrazené značce na mapě, zároveň jsou barevně rozlišeny dokončené body plánu (zelené, nedokončené jsou modře), a to jak v tabulce tak v mapě. Další „skrytou“ funkčností je možnost kliknout na značku plánu, což automaticky posune a přiblíží mapu na požadovaný bod v mapě. Tohoto jsem docílil následující funkcí:

```
01      const onMarkerClick = useCallback(
02        (e) => {
03          if (map && mapPoints) {
04            map.panTo(mapPoints[e.currentTarget.attributes['data-idx'].value])
05          }
06        },
07        [map, mapPoints]
08      )
```

Výpis kódu 4.30 Použití Google map – přesun mapy

Zároveň je při načtení obrazovky mapa vycentrována na první bod plánu a je oddálena tak, aby její viewport obsahoval všechny body. Tuto funkcionalitu není úplně jednoduché implementovat korektně, protože použitá mapa Google poskytuje pro tento účel pouze funkci `fitBounds`, která přesune a oddálí mapu tak, aby pokryla jen dva body. Pro pokrytí více bodů je nutné

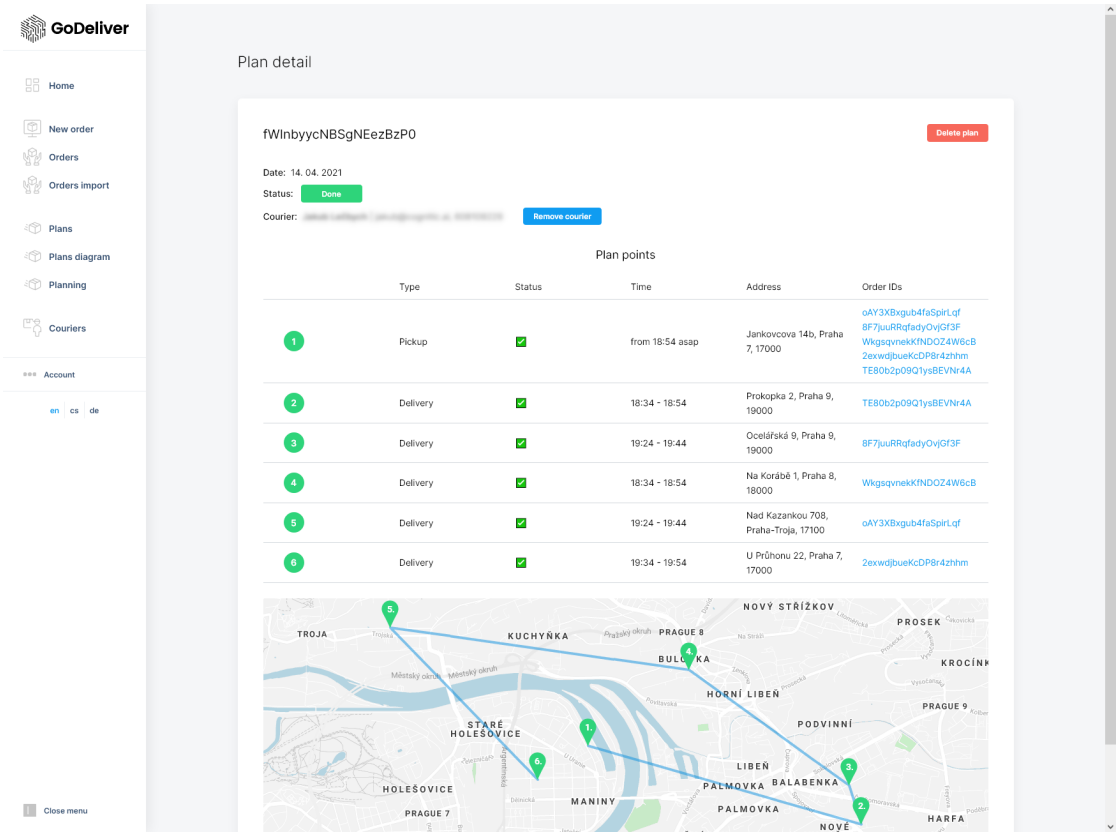
dopočítat ohraničení všech bodů, tedy například levý dolní a pravý horní roh požadovaného viewportu, respektive jeho souřadnice.

Protože v době implementace měla být aplikace používána pouze pro Českou republiku, rozhodl jsem se pro časový kompromis a jednoduše ze získaných GPS souřadnic bodů vybírám aritmeticky největší a nejmenší šířku a délku. Tento přístup pochopitelně rozbije funkcionalitu ve chvíli, kdy body přesáhnou nultý poledník a nebo rovník.

Mapu samotnou jsem pak vykreslil následovně, `Polyline` na řádku 9 vykreslí vzdušnou čáru mezi jednotlivými body pro přehlednost pořadí:

```
01        <GoogleMap ... >
02          {data.delivery_plan.delivery_points.map((point, idx) => {
03            return (
04              <Marker
05                icon={...} label={{ text: `${idx + 1}.` }}
06                position={mapPoints[idx]}
07              /> )
08          })}
09          <Polyline path={mapPoints} options={polylineOptions} />
10        </GoogleMap>
```

Výpis kódu 4.31 Použití Google map – vykreslení mapy a prvků v ní



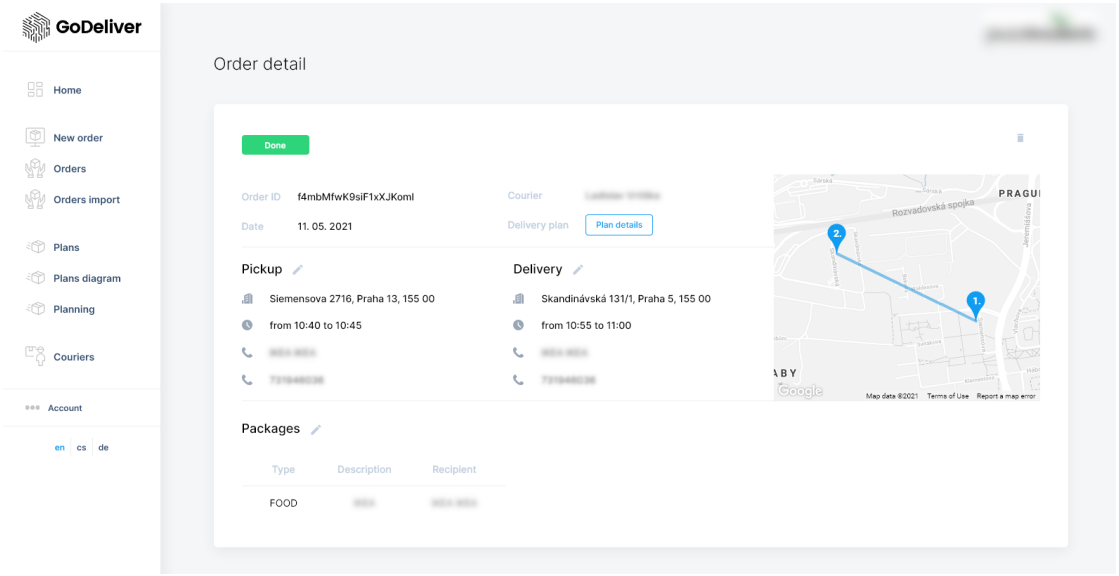
Obrázek 4.9 Výsledná obrazovka detailu plánu

4.3.3.4 Obrazovka detailu zásilky

Tato obrazovka, podobně jako obrazovka detailu plánu, slouží ke kontrole údajů vytvořených zásilek. Dále kromě údajů zadaných při vytváření obsahuje odkaz na detail plánu, do kterého je přiřazena, tedy je přes plán možné zjistit například který kurýr objednávku aktuálně doručuje.

Rozdíl oproti obrazovce detailu plánu je ten, že detail zásilky umožňuje editovat jednotlivé části zásilky. Tato funkcionalita je zpřístupněna pomocí kliknutí na ikonu editace u nadpisu části zásilky, tedy je možné editovat zvlášť vyzvednutí, doručení a balíky zásilky. Následně se zobrazí modální okno pro úpravu požadované části.

Nejdříve přikládám snímky implementované obrazovky detailu a okna pro úpravu údajů vyzvednutí objednávky, a poté popíšu detaily implementace editace.



Obrázek 4.10 Výsledná obrazovka detailu objednávky

Pro vytvoření funkcionality editace jsem nejdříve potřeboval vytvořit generický způsob, jakým zobrazovat modální okna, z toho důvodu, že jich bylo potřeba ve více místech aplikace. Pravděpodobně by pro tento účel bylo možné vytvořit jednoduchý komponent modálního okna jako obalový blok, a to pak zobrazovat, kde je potřeba.

Problém s tímto přístupem by byl ten, že bychom tento komponent dále museli vkládat do každého místa použití a museli bychom někde udržovat informaci o tom, kdy ho renderovat a kdy ne. Toto by bylo nutné opakovat při každém vložení komponentu. Také by nám tento přístup nijak nepomohl v případě, že bychom chtěli zobrazit modální okno po přechodu na jinou stránku aplikace. Po delším rozmýšlení mě napadlo využít k tomuto účelu React kontext, z mého pohledu jde o další ze zajímavějších způsobů jeho využití.

Nejdříve jsem vytvořil hook `useModal` s dále ukázaným interface. Položka `modal` obsahuje React element pro vyrenderování (pozn.: v případě že je element `null`, tak React nic nerenderuje). Na řádku 3 je funkce pro otevření modálního okna a na řádku 4 pro jeho zavření, kromě výchozího chování okna tak je možné ho zavřít třeba po vyhodnocení asynchronního volání API.

```
01    interface IModalContext {
02      modal: ReactElement | null
03      openModal: (children: ReactElement) => void
04      closeModal: () => void
05    }
```

Výpis kódu 4.32 Vytvoření hooku modálních oken – definice interface

Provider pro tento hook je také vlastně jednoduchý, v komponentu jsem přidal následující stav a funkce pro jeho změnu (vytvořené také pouze pro zjednodušení interface), hook `useCallback` jsem použil kvůli efektivitě renderování komponentu.

```
01    const [modal, setModal] = useState<ReactElement | null>(null)
02
03    const closeModal = useCallback(() => setModal(null), [setModal])
04    const openModal = useCallback(
05      (children: ReactElement) => setModal(<Modal>{children}</Modal>),
06      [setModal, Modal]
07    )
```

Výpis kódu 4.33 Vytvoření hooku modálních oken – funkcionalita

Dále jsem v provideru vytvořil samotný obalový komponent `Modal`. Komponent jsem vytvořil uvnitř provideru záměrně, potřeboval jsem mu předat interní funkci `closeModal`. `Modal` tedy obalí jakýkoliv obsah který do něj vložíme a zobrazí se jako fixní element uprostřed obrazovky se zatmaveným pozadím. Díky řádkům 6 a 9 se okno zavře, pokud klikneme právě na toto pozadí.

```
01      const Modal: React.FC = useCallback(
02        ({ children }) => (
03          <div
04            className="fixed inset-0 flex justify-center items-center z-50"
06            onClick={(_e) => closeModal()}
07          >
08            <div className="bg-white p-12 rounded-lg"
09                onClick={(e: React.MouseEvent) => e.stopPropagation()}>
10              {children}
11            </div>
12          </div>
13        ),
14        [closeModal]
15      )
```

Výpis kódu 4.34 Vytvoření hooku modálních oken – obalový blok pro okna

Následně už jsem jen přidal provider do obalového komponentu `_app.js` a dále jsem vyrenderoval obsah `modal` v komponentu `App.tsx`:

```
17    const { modal } = useModal()
...
19    return (
20      <div className="h-screen flex overflow-hidden bg-white">
21        {modal}
```

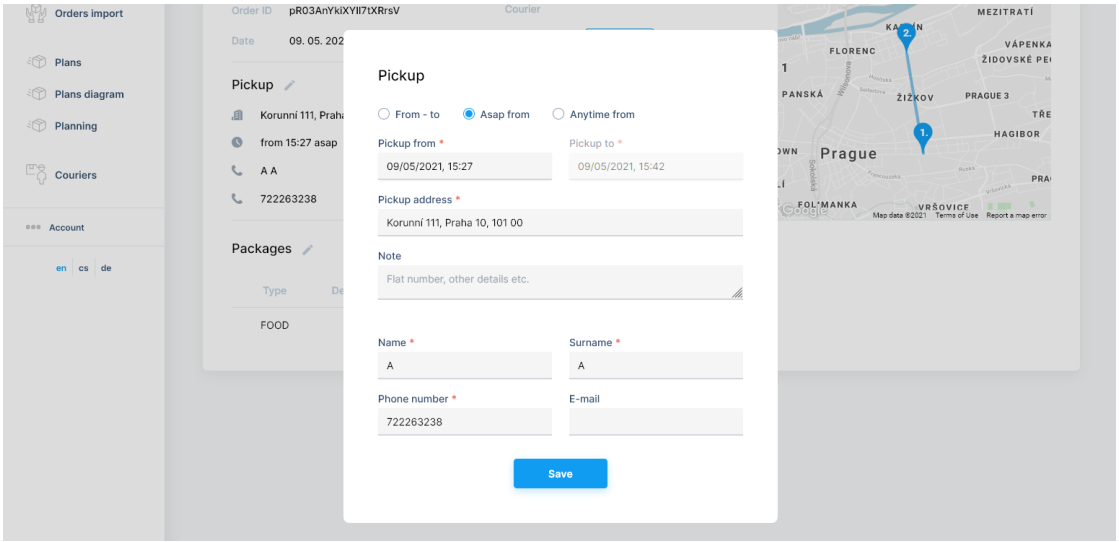
Výpis kódu 4.35 Vytvoření hooku modálních oken – vykreslení

Poté už bylo možné hook využít kdekoliv v aplikaci. Tedy v komponentu obrazovky jsem ho použil následovně. Při kliknutí na ikonu `IconEdit` se jako modální okno zobrazí komponent `PackageModalContent`, který obsahuje část formuláře z obrazovky nové objednávky. V tomto případě nebylo nutné předávat funkci pro zavření okna, jelikož se po úspěchu překreslí celá obrazovka kvůli získání nových dat, aby se propsaly změny do aplikace.

```
56    const { openModal } = useModal()
...
387   <IconEdit
388     className="..."
389     onClick={(_e: React.MouseEvent) => {
390       openModal(
391         <PackageModalContent
392           order={data.delivery_order}
393           onSuccess={() => reload()}
394         />
395       )
396     }}
397   />
```

Výpis kódu 4.36 Vytvoření hooku modálních oken – použití

Pochopitelně tento hook není použit pouze pro editaci objednávek, ale například také pro zobrazení hlášky o úspěchu a neúspěchu vytvoření objednávky, přenastavení plánovače a podobně. Implementovaná funkcionalita editace pak vypadá takto:



Obrázek 4.11 Funkcionalita editace na obrazovce detailu zásilky

4.3.3.5 Obrazovka plánování

Na této obrazovce se nachází nastavení funkce plánovače, respektive přepínání mezi automatickým a manuálním plánováním. V případě, že je plánovač nastaven na manuální provoz, je dále vykreslen seznam dnešních zásilek, které ještě nemají přiřazený plán, a možnost vytvořit z nich nový plán. Pokud je nastavení přepnuto na automatické plánování, zobrazuje se komponent umožňující nastavit počty kurýrů, kteří jsou k dispozici v různých úsecích dne.

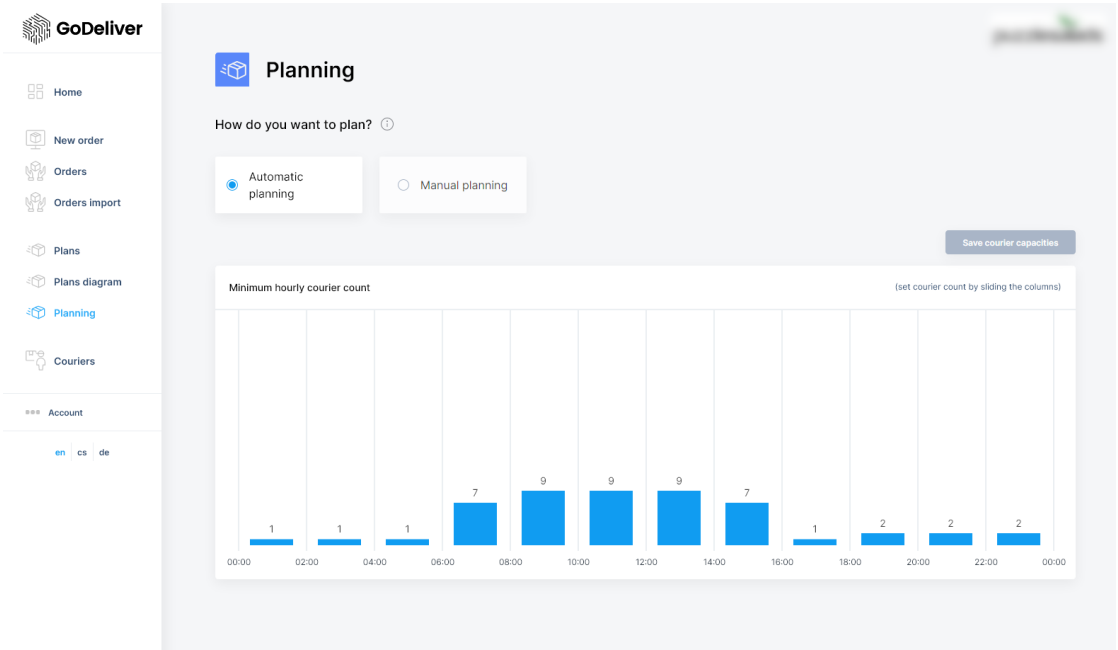
Část určena pro manuální plánování není až tak zajímavá, v zásadě jde o použití komponentu pro tabulky, podobně jako na obrazovce přehledu zásilek, s tím, že v posledním sloupci tabulky jsou pro každou zásilku přepínače pro přidání zásilky do seznamu, z kterého je následně vytvořen plán. V komponentu stránky jsem použil hook `useState` pro udržení stavu o označených zásilek, odtud `selected` a funkce `setSelected`. Přepínače jsou přidány takto ve specifikaci sloupců:

```
01    Cell: (cell: any) => {
02      return (
03        <ButtonCheckbox checked={selected.includes(cell.value)}
04          onChange={(e) => {
05            if (e.currentTarget.checked) {
06              setSelected([...selected, cell.value])
07            } else {
08              setSelected(selected.filter((val) => val !== cell.value))
09            }
10          }} /> ) },
```

Výpis kódu 4.37 Implementace označení zásilek v obrazovce plánování

Seznam označených zásilek je pak při vytvoření předán API pomocí asynchronní funkce `postCreatePlan` z datové vrstvy aplikace.

Zajímavější bylo vytvoření komponentu pro nastavení počtů kurýrů. Pro jednoduší vysvětlování postupu přidám snímek obrazovky už zde a implementaci popíšu dále.



Obrázek 4.12 Výsledná obrazovka plánování

Komponent nastavení kapacit se skládá ze sloupců reprezentujících počet kurýrů, kteří budou dostupní v dané době, po dvou hodinách. Jedná se tedy vlastně o sloupcový graf s časem na horizontální ose. Sloupci se vertikálně posunuje pro nastavení kapacity.

Zprvu jsem vůbec netušil, jak takovou funkcionalitu implementovat, žádnou knihovnu s požadovanými možnostmi se mi nepodařilo najít. Použití čistě JavaScriptu by bylo poměrně obtížné, jelikož by bylo nutné odchytávat události kliknutí, nějakým způsobem nastavovat, jak se sloupec pak zobrazí, a ze souřadnic kliknutí dopočítávat hodnotu, to by ale stále neřešilo posouvání sloupci.

Nakonec jsem zjistil, že HTML element pro vstup typu `range` v podstatě podporuje tuto funkcionalitu pokud by byl otočen vertikálně. Problémem je, že podpora pro stylování tohoto elementu přes CSS je při lepším „experimentální". Každý prohlížeč podporuje jinou část specifikace, o starších a mobilních verzích prohlížečů nemluvě.

Naštěstí je aplikace určena pouze pro desktop, což podstatně zjednodušilo otestování, jestli se komponent v prohlížečích zobrazuje správně, takže nakonec jsem zvolil cestu použití experimentálních API prohlížečů. Každý `input` element má přidanou CSS třídu `shift-slider`, která obsahuje následující nastavení:

```
01    .shift-slider {
02      appearance: none;
03      -webkit-appearance: none;
04      outline: none;
05      transform: rotate(270deg) translate(125px, -125px);
06    }
```

Výpis kódu 4.38 Implementace posuvných sloupců v plánování – 1

Takto jsem odebral výchozí stylování pomocí `apearance` na řádku 2, prefix `-webkit-` na řádku 3 je nutný pro použití v prohlížeči Safari. Dále jsem element otočil vertikálně a posunul. Poté bylo nutné použít dva nestandardní selektory CSS `::-moz-range-thumb` a `::-webkit-scrollbar-thumb` pro stylování tlačítka, kterým se pro nastavení hodnoty v rozsahu hýbe.

Díky omezení na desktop by měla být podpora pro tato API dostatečná pro drtivou většinu uživatelů, dle služby Can I use…(dostupné z https://caniuse.com) je kombinovaná podpora zhruba pro 95 % uživatelů, ze zbylých 5 % používají 4 % mobilní prohlížeče.

Tato CSS pravidla vypadají následovně. Podobně jako pro samotný element pro vstup jsem vypnul výchozí vzhled a pak jsem tlačítku přidal `box-shadow` v požadované barvě.

```
01    .shift-slider::-webkit-slider-thumb {
02      -webkit-appearance: none;
03      width: 0;
04      border: none;
05      background: var(--shift-col);
06      box-shadow: -200px 0 0 200px var(--shift-col);
07    }
```

Výpis kódu 4.39 Implementace posuvných sloupců v plánování – 2

Dále jsem takto nastylované posuvníky vložil do komponentu a přidal jim absolutní pozici a šířku, které se dopočítávají při změně šířky viewportu prohlížeče. Hodnoty sloupců jsou pak při potvrzení odeslány jako pole hodnot v požadavku na API GoDeliver.

Implementaci zbylých obrazovek dále popisovat nebudu, ve většině případů byla přímočará. Snímky výsledných implementací těchto obrazovek jsou dále v příloze na konci práce.

4.4 Uživatelské testování použitelnosti

V této kapitole nejdříve analyzuji cílovou skupinu uživatelů, dále identifikuji jejich cíle, z těchto vytvořím scénář pro uživatelské testování a následně popíši výsledky testování.

4.4.1 Analýza cílové skupiny uživatelů

Systém GoDeliver jako takový je cílen na poměrně úzkou skupinu uživatelů. Jelikož jeho administrační rozhraní bude používáno pouze dispečery společností, vytvářená aplikace tak zahrnuje ještě užší cílovou skupinu. Pro účely této práce je aplikace cílena pouze na Českou republiku, dostupná je tedy jen v češtině. Odhadovaný věk uživatel se pohybuje mezi 25–55 lety, půjde o muže i ženy.

Vzhledem k tomu, že uživatelé aplikace budou primárně dispečeři a dispečerky nebo manažeři a manažerky poboček společností, lze od nich očekávat nadprůměrné vzdělání, manažerské a organizační schopnosti. Ačkoliv tedy půjde o profesionály, je pravděpodobné, že někteří z uživatelů se nikdy s podobným logistickým systémem jako GoDeliver nesetkali. Na jednotlivých pobočkách například restaurací podobnou funkcionalitu jako systém často vykonává pouze mobilní telefon, dispečer obvolává kurýry a zjišťuje stav rozvážek.

4.4.2 Persony

Jako typické uživatele jsem tedy zvolil dvě persony, jednu v roli zkušeného uživatele a druhou v roli uživatele, který se zatím s podobným systémem nesetkal. Díky personám by mělo být dále snazší zaměřit se na požadavky a cíle uživatelů. Vzniklé klíčové persony jsou:

- *Marek* – Marek je pětatřicetiletý manažerem pobočky většího řetězce obchodů, s obdobnými logistickými systémy má dlouholetou zkušenost. Rád pracuje efektivně, ale potřebuje mít v práci dobrý přehled.
- *Marie* – Marie je čtyřicetiletou manažerkou menší pobočky řetězce restaurací, s žádným logistickým systémem se zatím nikdy nesetkala. Vždy řešila veškeré problémy s rozvážkami telefonicky, díky tomu je zvyklá na krizovější situace. Kvůli aktuální situaci společnost přešla na logistický systém za cílem zvýšení objemu dovážek.

4.4.3 Uživatelské cíle

Uživatelské cíle pro potřeby uživatelského testování vychází z funkčnosti aplikace, ale také z potřeb identifikované cílové skupiny uživatel a vytvořených person. Tyto uživatelské cíle jsem vybral zvlášť pro jednotlivé obrazovky aplikace a vypadají následovně:

- *Přihlašovací obrazovka* – registrace a přihlášení uživatele.
- *Úvodní obrazovka (přehled informací)* – rychlé zjištění stavu systému, počtu zpožděných rozvážek a probíhajících plánů apod.
- *Obrazovka vytvoření zásilky* – zadání nové zásilky do systému.
- *Přehled zásilek* – zobrazení základních informací o zásilkách pro zadaný den, najití konkrétní zásilky a zobrazení jejího detailu.
- *Detail zásilky* – zobrazení detailních informací o zásilce jako např. obsah zásilky, adresy a časy vyzvednutí a doručení, možnost zrušení zásilky, přechod na detail přiděleného rozvozového plánu apod.

- *Přehled rozvozových plánů* – zobrazení základních informací o plánech pro zadaný den, najití konkrétního plánu a zobrazení detailu.
- *Detail rozvozového plánu* – zobrazení detailních informací o plánu jako např. informace o jednotlivých bodech plánu, přiřazených zásilkách, možnost zrušit plán, možnost přiřazení konkrétního kurýra.
- *Diagram plánů* – zjištění informací o předpokládaných časech vyzvednutí a doručení zásilek, a případných zpožděních.
- *Obrazovka plánování* – přepnutí mezi ručním a automatickým plánovačem, v případě automatického nastavení kapacity kurýrů v průběhu dne, v opačném případě vytvoření rozvozového plánu výběrem konkrétních zásilek.
- *Přehled kurýrů* – zobrazení přehledu kurýrů, zjištění stavu určitého kurýra, přizvání nového kurýra.
- *Obrazovka účtu* – zobrazení informací o uživatelském účtu, odhlášení se.

4.4.4 Scénář pro uživatelské testování

Z výše identifikovaných uživatelských cílů jsem vytvořil scénář, podle kterého by měl uživatel při testování projít aplikací. Poznámky v závorkách jsou pouze moje komentáře pro účely této práce, uživatel samotný scénář dostal bez těchto informací, naznačuji v nich potenciální problémy, na které jsem se snažil zaměřit při pozorování testovaného uživatele.

1. Otevřete aplikaci ve webovém prohlížeči na adrese https://dev.admin.godeliver.co.
2. Přepněte jazyk aplikace na češtinu. (U tohoto bodu pravděpodobně uživatel bude mít problém najít na obrazovce tuto možnost, nachází se v pravém horním rohu obrazovky.)
3. Zaregistrujte se s následujícími uživatelskými údaji (Tento bod by měl být bezproblémový.):
   - *Jméno, příjmení a e-mail* – vaše osobní informace
   - *Telefonní číslo* – 123 456 789
   - *Název firmy* – UI-test-X, kde `X` je vaše přidělené číslo
   - *Adresa firmy* – Pernerova 697/35, Karlín, 186 00
   - *Heslo, heslo znovu* – libovolné heslo
4. Pozvěte do své společnosti nového kurýra, pozvání odešlete na e-mail `matous+kuryr@cognitic.ai`. (Tady by také neměl být problém, pokud uživatel pochopí, že musí přejít na obrazovku kurýrů.)
5. Přidejte do systému novou zásilku, měla by vypadat takto (U tohoto bodu by mohly být problémy se zadáváním zejména časových údajů.):
   - Zásilka má dva balíky, jeden hamburger pro Marka a jeden nákup pro Kateřinu
   - Vyzvednuta by měla být za hodinu a co nejdříve, předávat ji bude na adrese Skuteckého 1395/1 Jan Vomáčka, měl by být případně dostupný na čísle `987654321`
   - Doručena by měla být co nejdříve, nejlépe deset minut po vyzvednutí
   - Zásilka je určena pro Marka Ostrého, telefonní číslo na něj je `456789123`, adresu udal pouze jako Pernerova 12 v Karlíně.

6. Zkontrolujte údaje nově vytvořené zásilky a jestli se do systému propsaly všechny její položky. (Zde uživatel musí zásilku najít v přehledu zásilek a přejít do jejího detailu, neměl by tu být problém.)
7. Vytvořte ručně nový plán, do kterého přidáte vámi vytvořenou zásilku. (V tomto bodě musí uživatel hledat funkcionalitu, která se nachází na obrazovce plánování.)
8. Ověřte vytvořený plán a přiřaďte mu vámi pozvaného kurýra. (Uživatel musí přejít na detail plánu.)
9. Zjistěte, jestli se změnil stav kurýra. (Uživatel tuto informaci nalezne na obrazovce kurýrů.)
10. Ověřte, jestli kurýr již vyzvedl zásilku. (Během předchozího bodu v aplikaci pro kurýry označím zásilku za vyzvednutou.)
11. Přepněte plánovač na automatické plánování a nastavte kapacity kurýrů na jednoho během celého dne. (Této možnosti si uživatel pravděpodobně všiml při vytváření plánu.)
12. Ověřte stav plánu. (Tady by už měl být plán dokončený.)
13. Zkontrolujte si informace o vašem účtu.
14. Odhlaste se z aplikace.

4.4.5 Provedení a výsledky uživatelského testování

Testování použitelnosti jsem provedl s pěti uživateli, kteří neměli s obdobnými administračními systémy žádné zkušenosti. Věkově účastníci testování odpovídali výše vytyčené cílové skupině uživatelů, jednalo se o dvě ženy a tři muže. Těmto uživatelům byl předložen dotazník obsahující vytvořený scénář, společně s bodovým ohodnocením obtížnosti každého úkonu s možnostmi od jedné do pěti a polem pro případný textový popis problému, který uživatelé měli s dosažením cíle.

Testovaného jsem navedl k tomu, aby si představil, že je zaměstnancem společnosti, která provádí rozvážku jídla, a že dostal za úkol provést popořadě jednotlivé body v dotazníku. Tedy aby se k úkolům postavil jako k pracovní činnosti a aby se snažil provádět úkony tak, jak by je dělal v práci (tedy i pod časovým tlakem). Zároveň jsem poté uživatele sledoval při plnění zadaných bodů a nezasahoval jsem do jejich postupu, ani jsem jim žádným způsobem neradil v případě, že se dostali do úzkých. V tomto ohledu by samozřejmě bylo lepší využít kameru a pozorovat uživatele vzdáleně kvůli tomu, aby při vyplňování dotazníku nebyl rušen.

Výsledky testování byly poměrně zajímavé. Průměrná doba dosažení všech cílů dotazníku byla zhruba 24 minut, nejrychlejší účastník všechny otázky zpracoval za 19 minut, rozptyl tedy nebyl nijak velký. Žádný z dotazovaných neměl při plnění úkolů nepřekonatelný problém, vše probíhalo vcelku hladce.

K mému překvapení měla nejméně problémů s dosažením všech cílů uživatelka, která byla nejméně technicky zaměřená, a tak měla nejméně zkušeností s webovými aplikacemi obecně. Naopak k tomu bylo vyplnění dotazníku nejsložitější pro testovaného, který se přímo zabývá technickými obory a musí interagovat s webovými aplikacemi v běžném životě prakticky neustále. Do určité míry si myslím, že toto bylo způsobeno tím, že se nebyl dostatečně schopen vžít do požadované role a snažil se pochopit, jak systém funguje, zároveň tohoto uživatele mátly používané termíny (zásilka, rozvozový plán apod.), kterým nerozuměl.

Popořadě byly obtížnosti (hodnoceny od jedné do pěti – bez obtíží až velmi obtížné) jednotlivých bodů scénáře testovanými vyplněny takto (ke každému bodu přikládám rovnou i detailnější popis problémů uživatel, svá pozorování a případné poznámky pro další vylepšení UX aplikace):

1. *1.–3. bod* – všichni testovaní vyplnili obtížnost 1, tedy vykonání bodů proběhlo hladce.

2. *4. bod (přizvání kurýra)* – všichni vyplnili obtížnost 1. V tomto bodě nebyl prakticky žádný problém s plněním cíle, nicméně jsem si všiml poměrně zajímavého fenoménu. V seznamu kurýrů je u každého kurýra, pokud je online, jako poslední prvek řádky červené tlačítko se symbolem pro vypnutí a zapnutí (stejný symbol jako tlačítko pro zapnutí a vypnutí počítače). Tato tlačítka nemají žádný slovní popis a jsou na celé obrazovce barevně (ne velikostí) nejvýraznějším prvkem. U všech uživatelů jsem zaznamenal to, že se okamžitě snažili zjistit, k čemu tlačítko slouží (vynutí vypnutí kurýra v mobilní aplikaci), a někteří na něj bez zamyšlení kliknuli.

3. *5. bod (přidání zásilky do systému)* – dva uživatelé vyplnili obtížnost 1, dva obtížnost 2, jeden obtížnost 4. U tohoto bodu jsem záměrně v zadání použil u časových údajů spojení „co nejdříve od“, z důvodu zjednodušení úkonu. Termíny použité v systému pro nastavení formátu časových údajů („od-do“, „co nejdříve od“ a „kdykoliv od“) jsou obtížně pochopitelné bez složitějšího vysvětlení fungování systému.

   Obecně z pozorování uživatelů bylo zřetelné, že byli prakticky všichni zmateni možnostmi časových rozsahů u vyzvednutí a zároveň i předání zásilky. Z mého pohledu jde ale zejména o zmatení z důvodů nezkušenosti s rozvážkami, při diskuzi po odevzdání dotazníku a po vysvětlení, jak zadání zásilky funguje, pak dotazovaní uvedli, že jim způsob vytváření zásilky začal dávat smysl.

4. *6. bod (ověření údajů zásilky)* – čtyři dotazovaní vyplnili obtížnost 1 a jeden 2. U účastníka, který měl drobné obtíže došlo k chybě při zadání údajů nové zásilky, následně si při ověřování údajů chyby všiml, ale činilo mu problémy najít způsob úpravy již vytvořené zásilky. Tlačítka pro editaci jsou na obrazovce detailu zásilky málo výrazná a nový uživatel si jich pravděpodobně napoprvé nevšimne.

5. *7. bod (vytvoření plánu)* – jeden dotazovaný označil obtížnost 1, tři 2, jeden 3. U tohoto bodu jsem obtíže očekával, rozvrhnutí obrazovky pro manuální plánování je dle mého názoru nešikovné. Tlačítko, které potvrdí výběr zásilek, se nachází nad seznamem, ze kterého uživatel musí vybírat, a tak umístěním vlastně neodpovídá postupu, který musí uživatel vykonat. Zároveň by bylo vhodné oddělit nastavování plánovače a samotné vytváření plánů, uživatelé byli do určité míry zmateni přebytečnými informacemi.

6. *8.–10. bod* – tyto body byly bez problémů.

7. *11. bod (přepnutí na automatický plánovač a nastavení kapacit)* – jeden dotazovaný 1, tři 3 a jeden 4. V tomto bodě jsem očekával u dotazovaných více potíží se splněním zadání. Přepnutí na automatizaci plánování nedělalo potíže nikomu, ale nastavení kapacit bylo kamenem úrazu.

   Zprvu textový popisek „posouváním sloupců nastavíte počet kurýrů“ pochopili všichni uživatelé jako pohyb sloupci zleva doprava nebo zprava doleva, ale ne vertikálně. Uživatele také na první pohled zmátlo to, že při nulových kapacitách nejsou žádné sloupce vidět, ani nejsou naznačené. Na druhou stranu ale bylo zajímavé sledovat, jak uživatel při nepochopení grafického rozhraní okamžitě začal hledat textové informace, které by ho k cíli dovedly.

   Základním problémem celého bodu bylo nepochopení uživatel, čeho se vlastně snaží dosáhnout, dále dělalo obtíže pochopit, že jednotlivé sloupce nastavují kapacitu kurýrů pro časové úseky během dne (časová osa pod sloupci je pravděpodobně málo výrazná). Díky tomu si pak nebyli jisti, jestli nastavili kapacitu na jednoho kurýra v průběhu dne, nebo dvanáct kurýrů na den (dvanáct sloupců).

8. *12.–14. bod* – dále nebyly žádné obtíže.

Výsledky uživatelského testování pro mě byly zajímavé, a při práci na dalších projektech bych se jím chtěl, pokud bude možnost, více zabývat. Testování aplikace bylo provedeno až v pozdní fázi vývoje, a to do té míry, že jeho výsledky nebudou využity pro úpravy této aplikace, ale při zakomponování stávající funkcionality do nové aplikace, která je momentálně vyvíjena.

Kapitola 5
Závěr

Bakalářská práce popisuje tvorbu webové aplikace pro firmu Cognitic s.r.o., sloužící jako klientská část administračního rozhraní pro logistický systém GoDeliver. Hlavním cílem práce bylo implementovat frontendové řešení rozhraní využívající stávající serverovou část systému za použití javascriptového frameworku React. Dle dodaných materiálů jsem vytvořil za pomoci dodatečných technologií aplikaci splňující všechny funkční požadavky zadání a odpovídající dodaným grafickým návrhům.

V první části práce jsem provedl analýzu dodaných materiálů a vyvodil z nich požadavky na funkčnost aplikace. Dále jsem analyzoval stávající serverové řešení a jeho veřejné rozhraní, identifikoval a popsal entity vyskytující se v systému. Poté jsem pro účely vývoje projektu nakonfiguroval pomocné nástroje a vytvořil pipeline pro automatickou integraci a sestavení ve službě GitHub, nasazení aplikace jsem provedl ve službě Google App Engine. Tento pipeline jsem vytvořil tak, aby probíhalo sestavení, testování a nasazení aplikace ve vývojové a produkční verzi. Následně jsem implementoval datovou vrstvu a postupně jsem po jednotlivých obrazovkách a jejich komponentech vytvořil celou aplikaci. Nakonec jsem provedl uživatelské testování systému, ze kterého jsem vyvodil závěry pro další možné zlepšení použitelnosti.

Výsledkem celé práce je implementované kompletní uživatelské rozhraní pro logistickou službu. Aplikace umožňuje dispečerům společností vytvářet zásilky v systému, kontrolovat průběh vytvářených rozvozových plánů a stav zásilek v nich obsažených, zobrazovat přehledy a data o doručovaných zásilkách, přizvat do společnosti nové kurýry, měnit chování plánovače systému a celkově kontrolovat veškerou jeho práci.

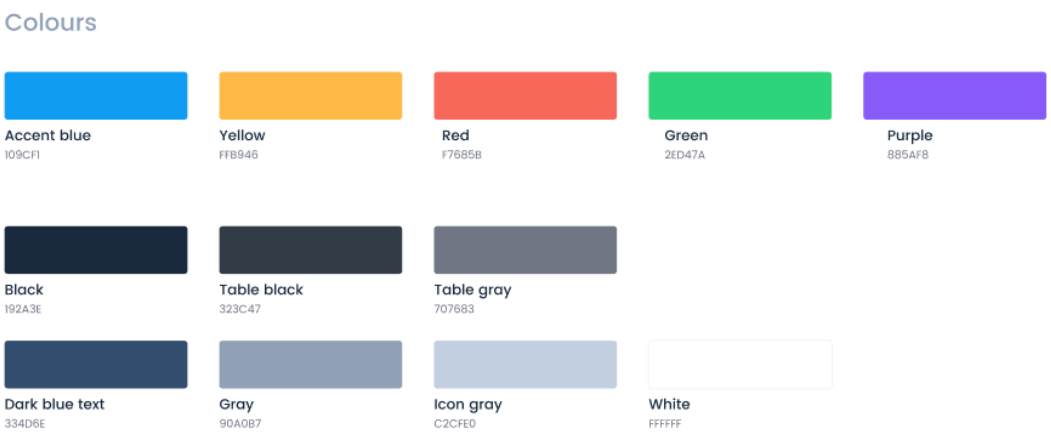
Aplikace je nasazena na produkčním prostředí a k reálnému provozu ji využívá pět společností při rozvozu zásilek. Dispečerům a manažerům těchto společností systém pomáhá mít kontrolu a přehled nad prací automatizovaného rozvážkového systému. Denně je tímto způsobem do systému zadáno ručně okolo desíti zásilek, běžně pak společnosti rozvezou za použití systému GoDeliver za den dohromady téměř dvě stě zásilek.

Momentálně je aplikace považována firmou Cognitic za dokončenou, probíhají pouze opravy a úpravy na základě zpětné vazby společností využívajících tento systém. Zároveň s tím je vyvíjena nová frontendová aplikace, z důvodů přechodu na získávání a zobrazování dat o zásilkách a kurýrech společností v reálném čase, pro lepší kontrolu společností nad prací kurýrů, dále by nová verze měla poskytovat více možností zobrazení dat. Poznatky vzniklé z této práce a zejména provedeného uživatelského testování použitelností dále budou využity pro vylepšení UX nové verze administračního rozhraní.

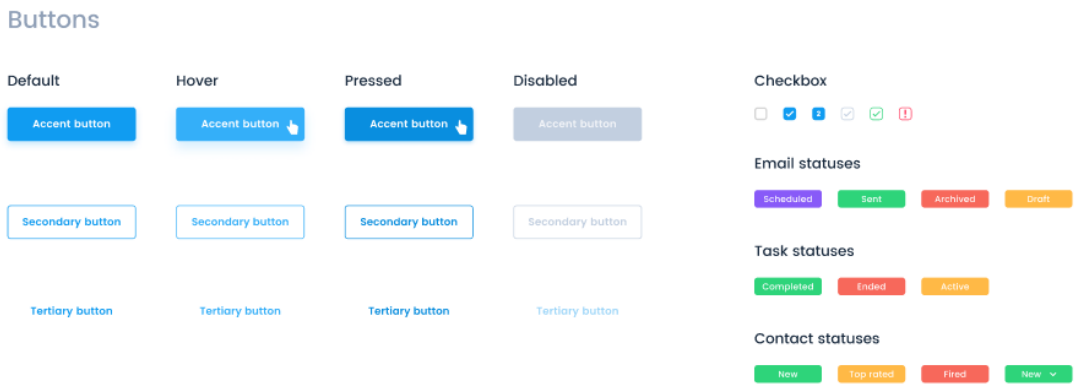
Příloha A

Příloha

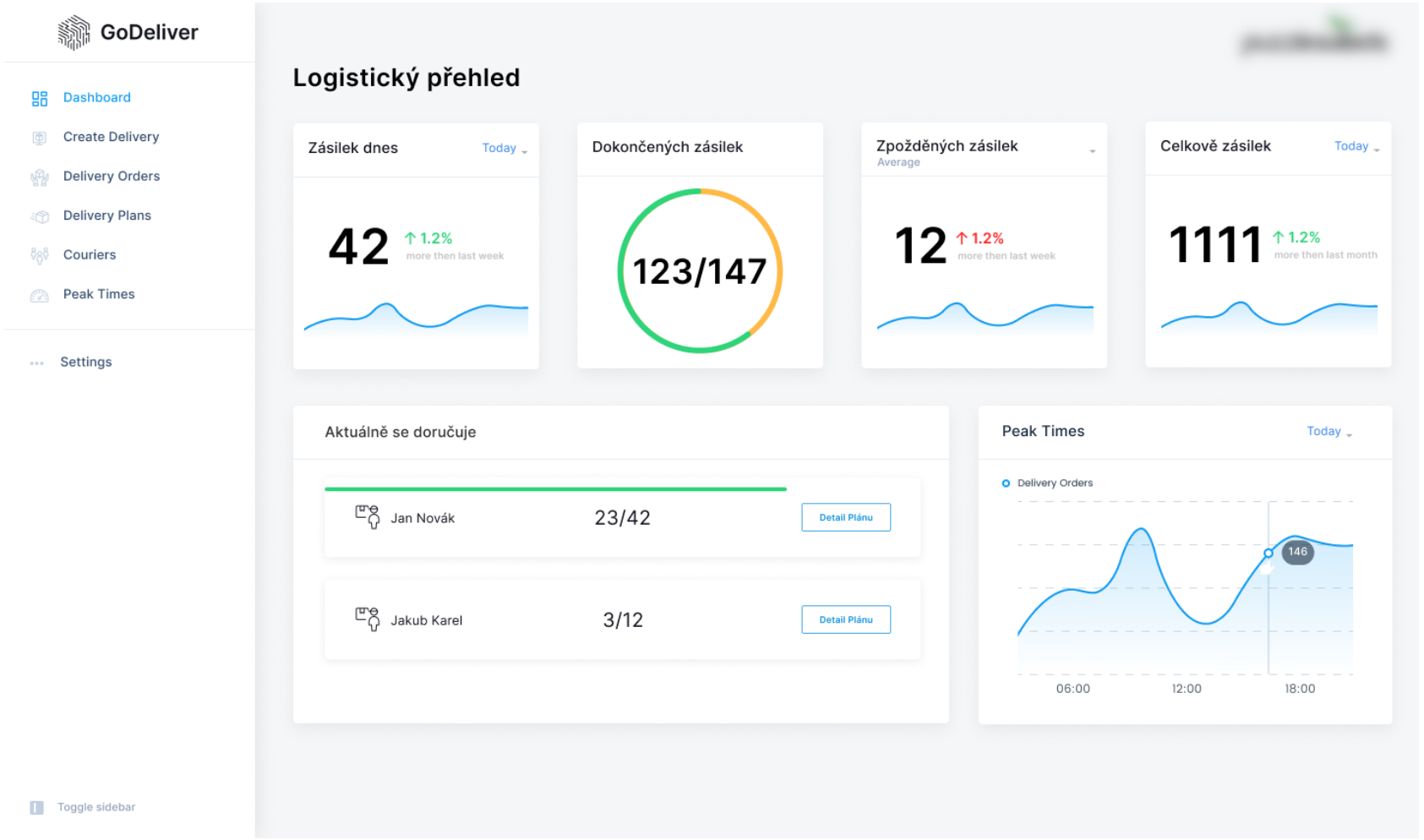
V této kapitole se nachází přílohy práce, zejména snímky grafických návrhů a dále snímky implementovaných obrazovek.



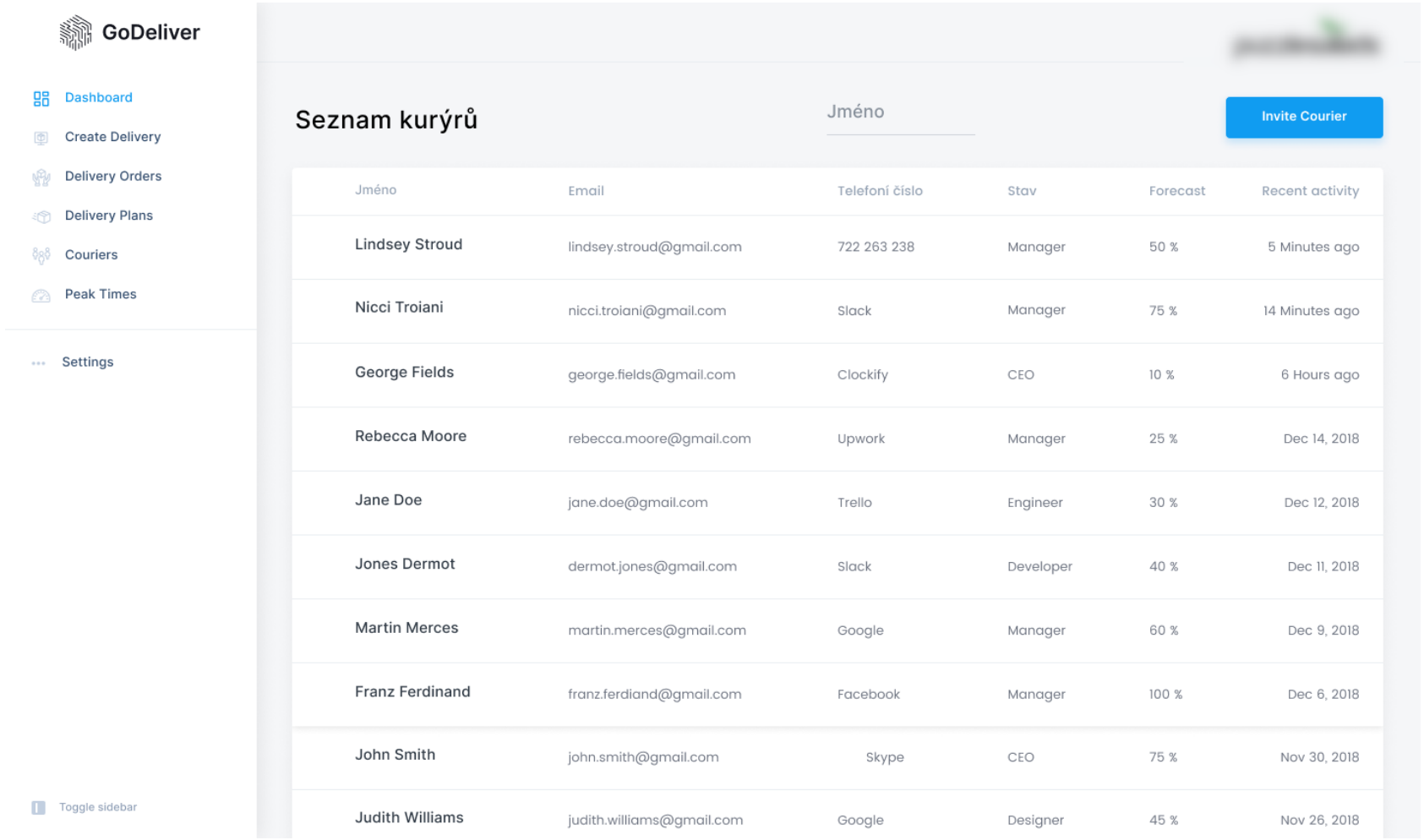
Obrázek A.1 Návrh barevnosti aplikace



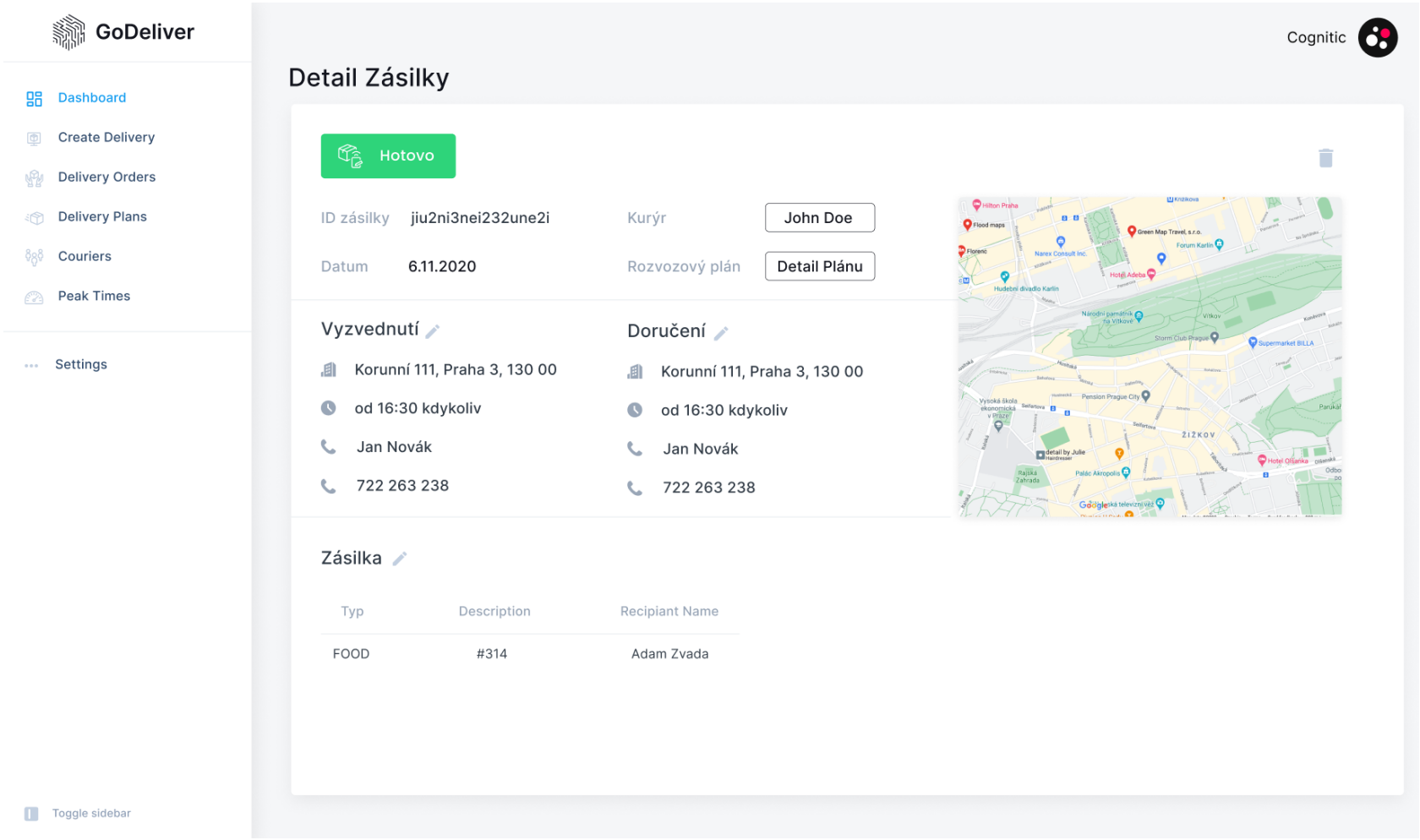
Obrázek A.2 Návrh tlačítek a odznaků k použití v aplikaci



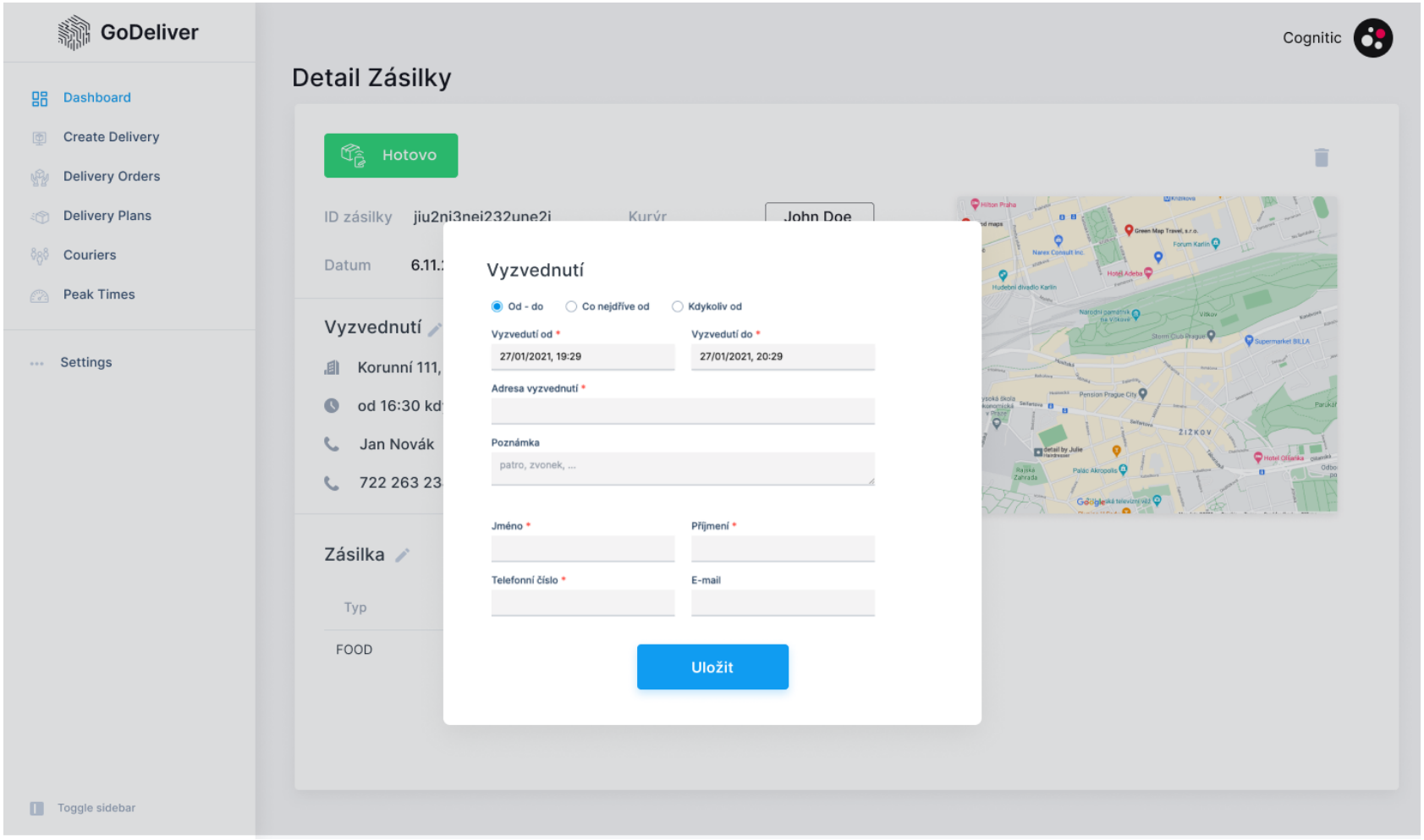
Obrázek A.3 Návrh úvodní obrazovky



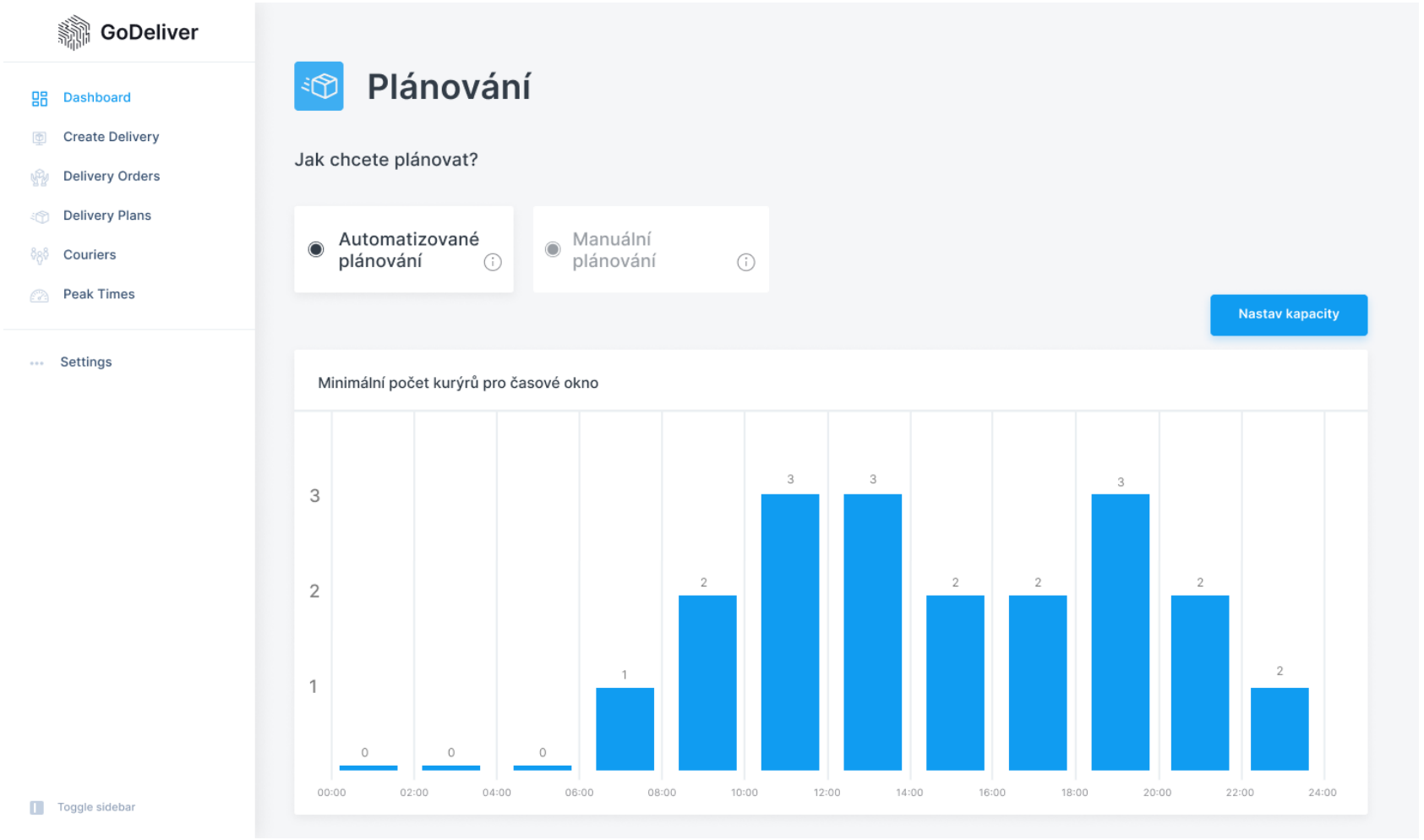
Obrázek A.4 Návrh obrazovky přehledu kurýrů



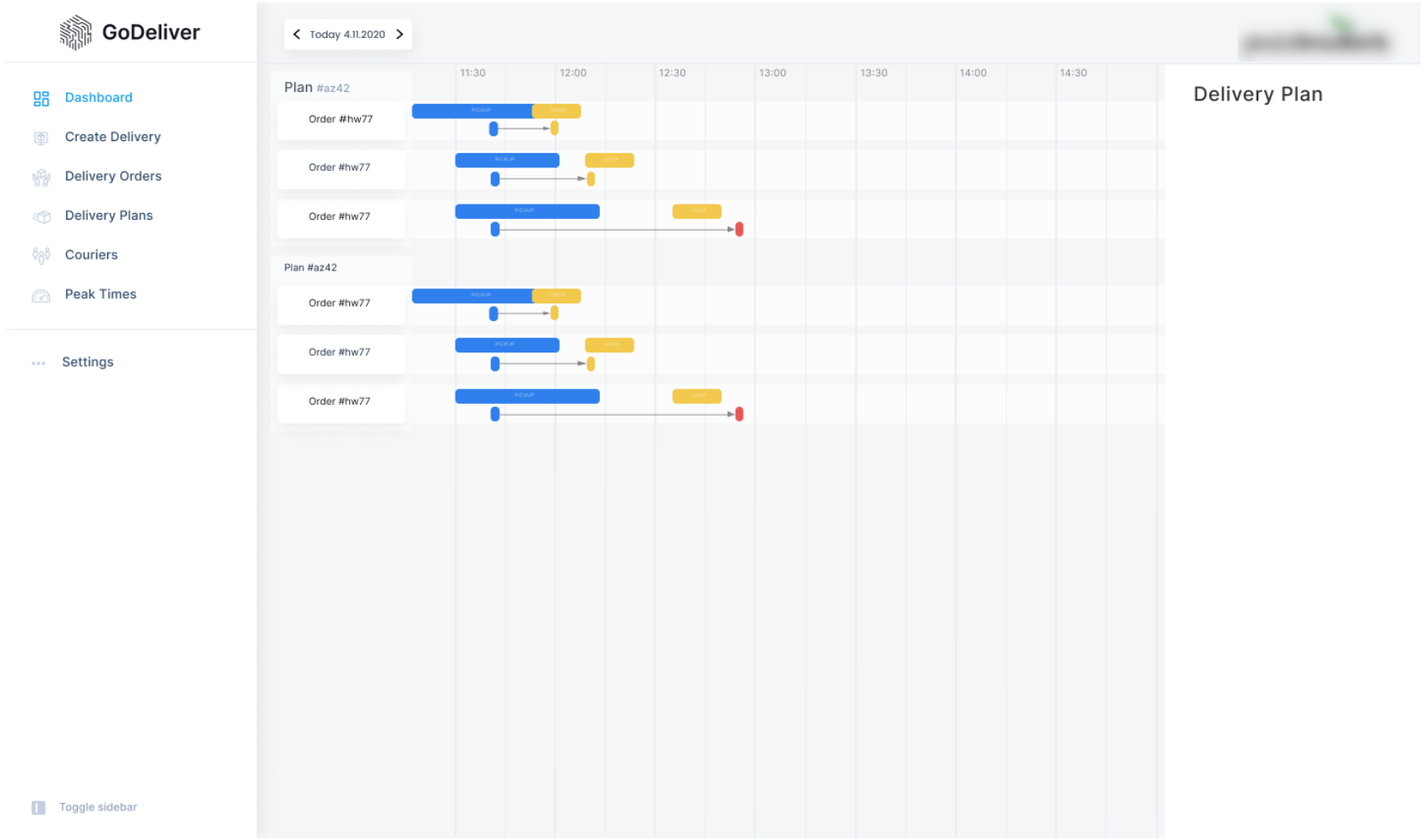
Obrázek A.5 Návrh obrazovky detailu zásilky



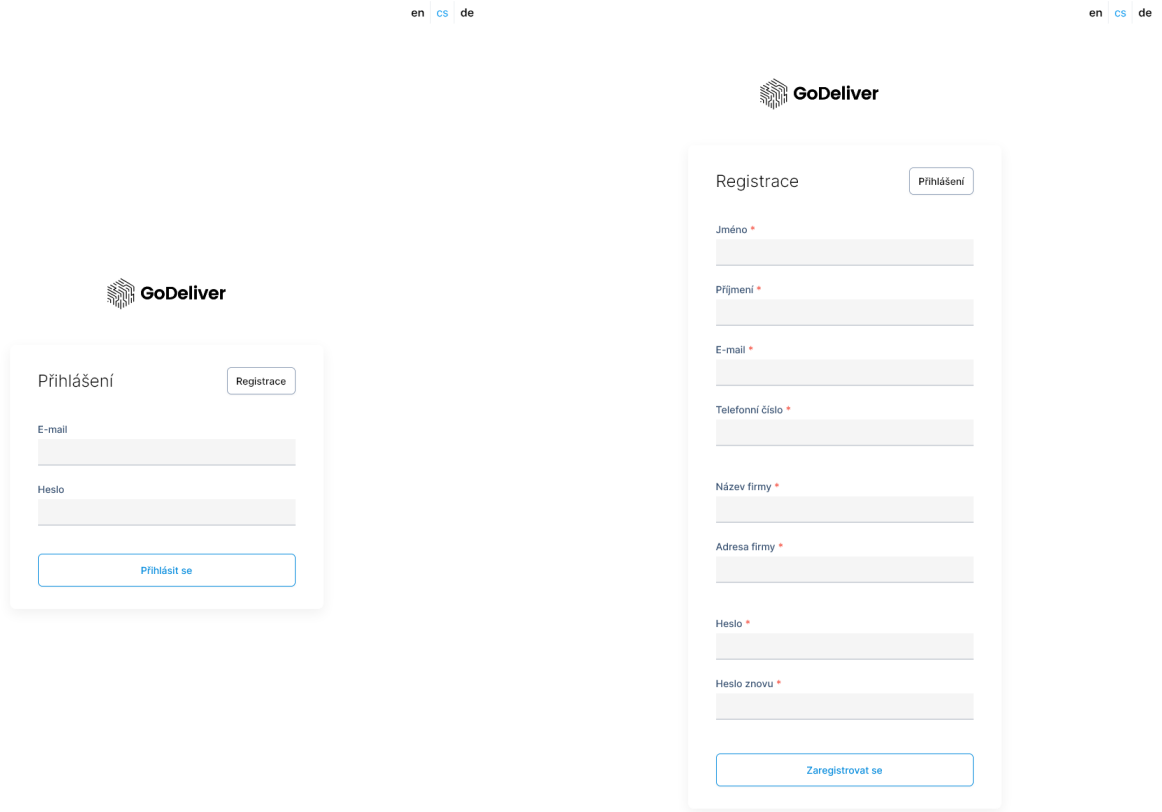
Obrázek A.6 Návrh obrazovky editace zásilky



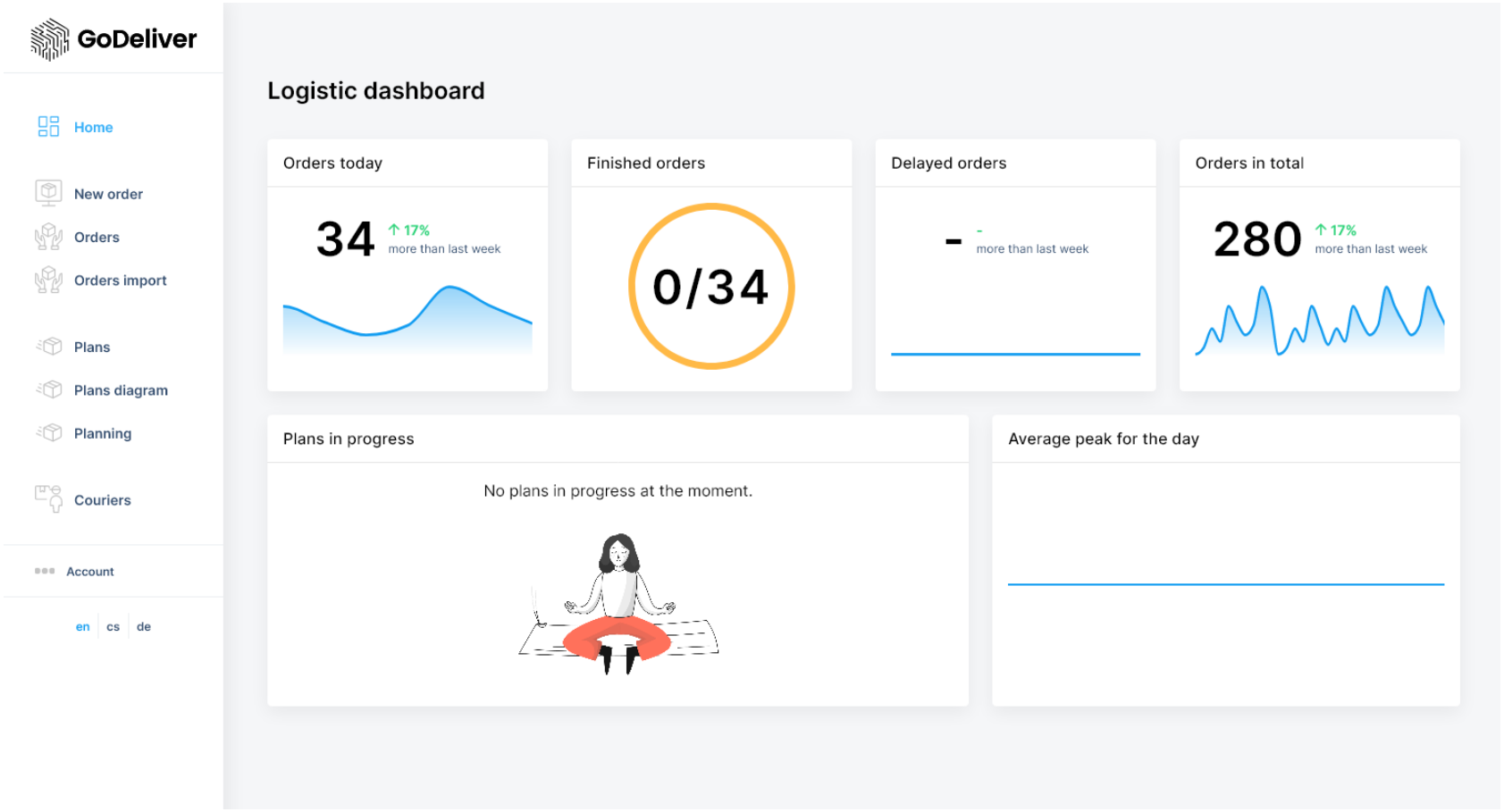
Obrázek A.7 Návrh obrazovky nastavení plánovače



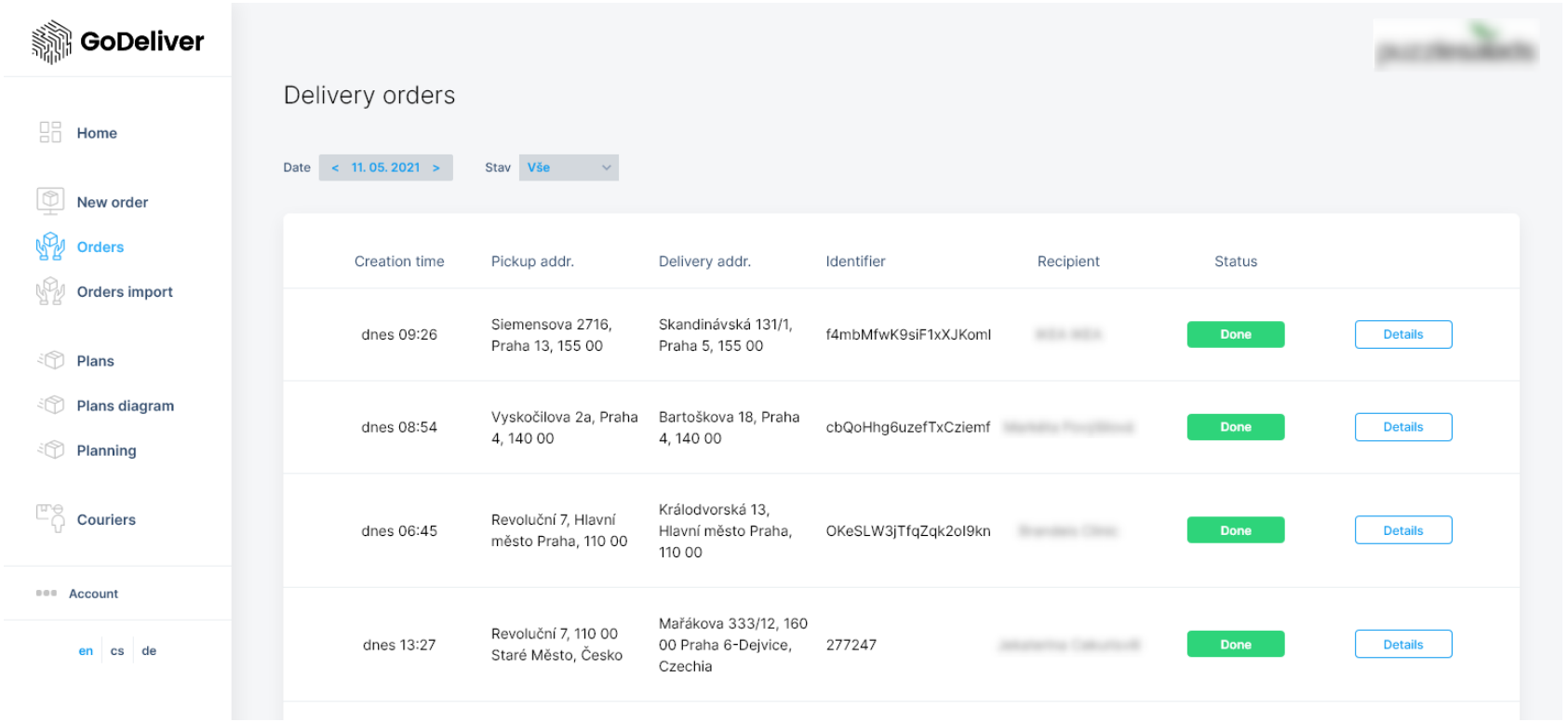
Obrázek A.8 Návrh obrazovky diagramu plánů



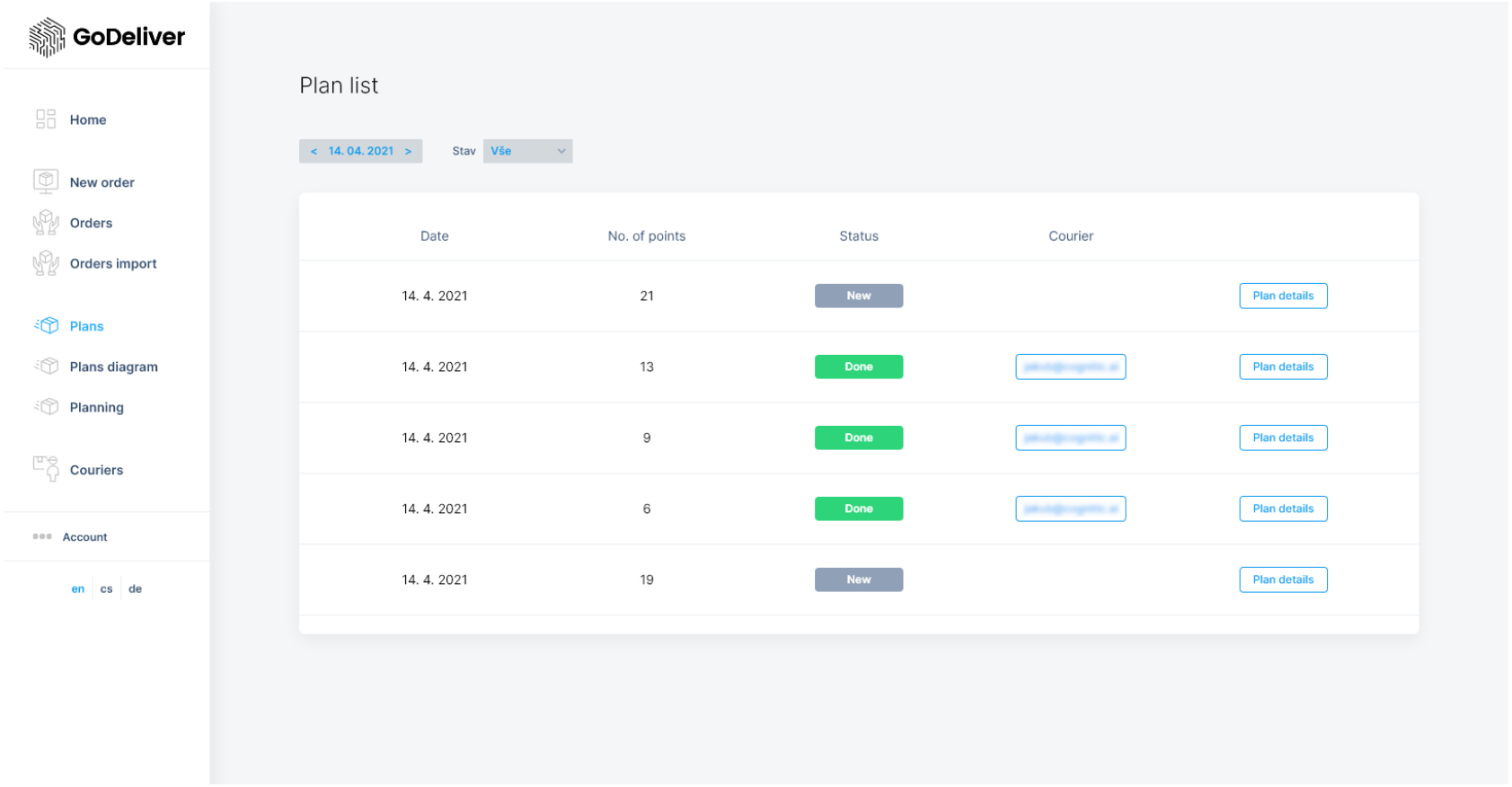
Obrázek A.9 Implementace přihlášení a registrace



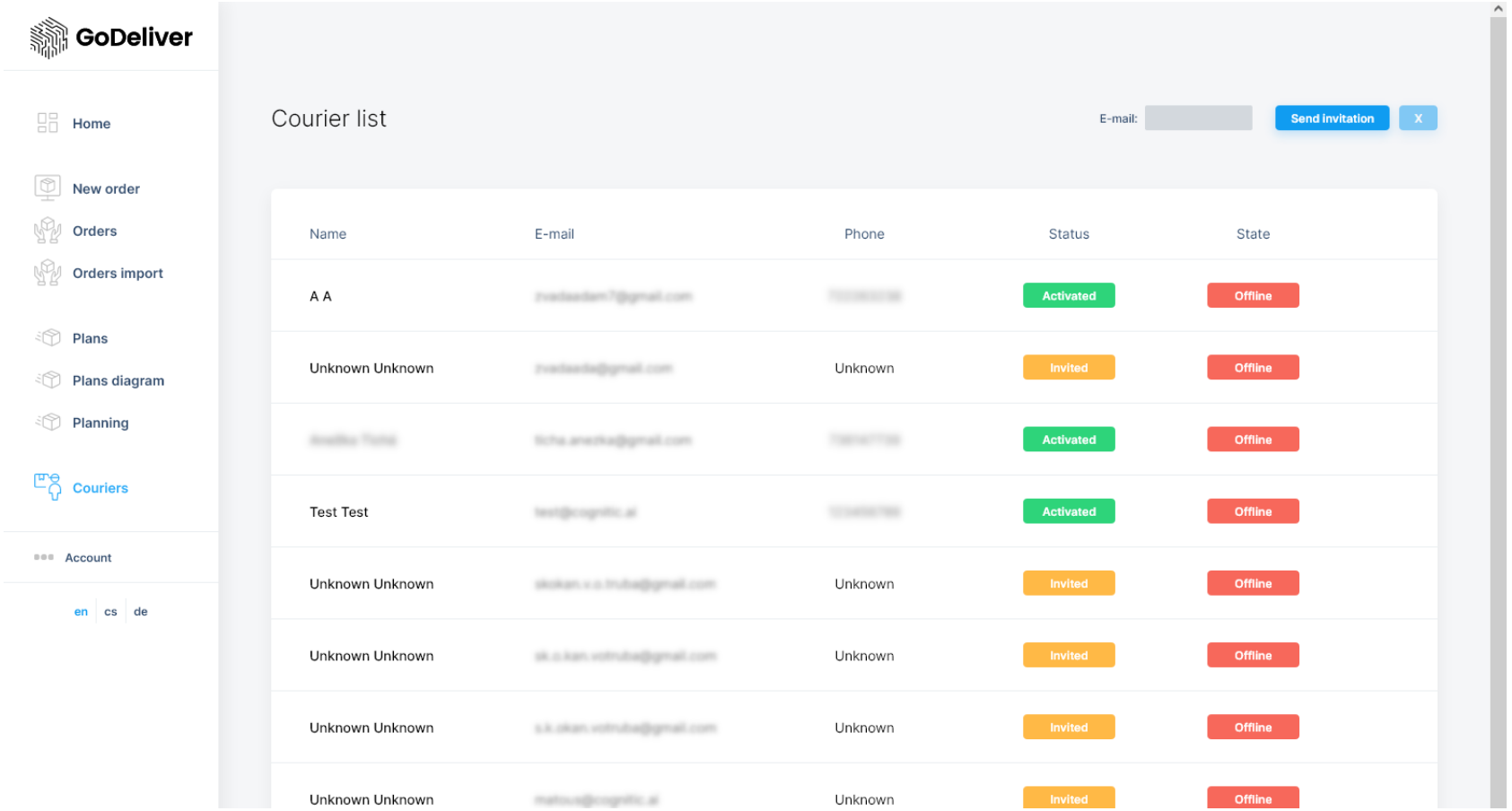
Obrázek A.10 Implementace úvodní obrazovky



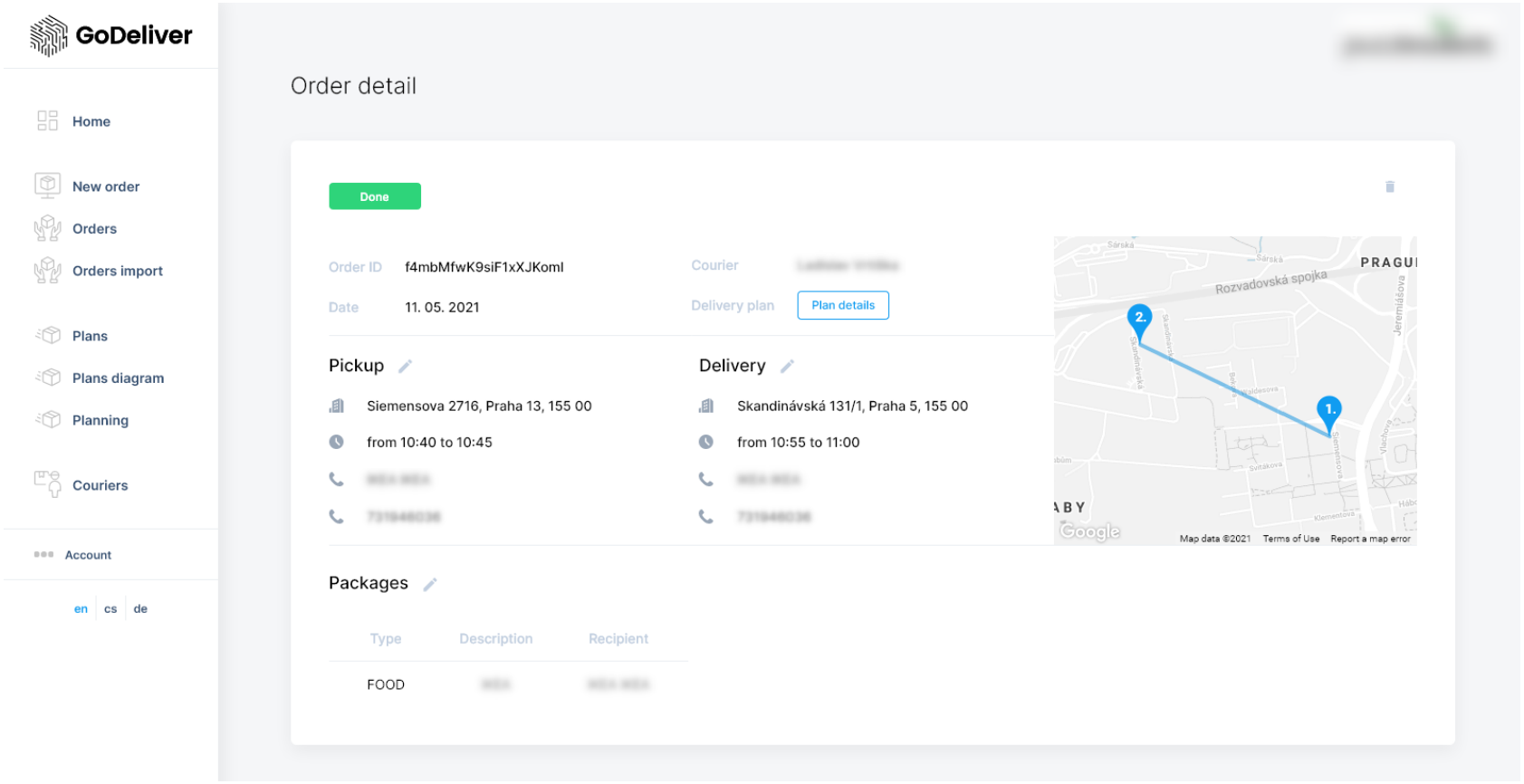
Obrázek A.11 Implementace přehledu zásilek



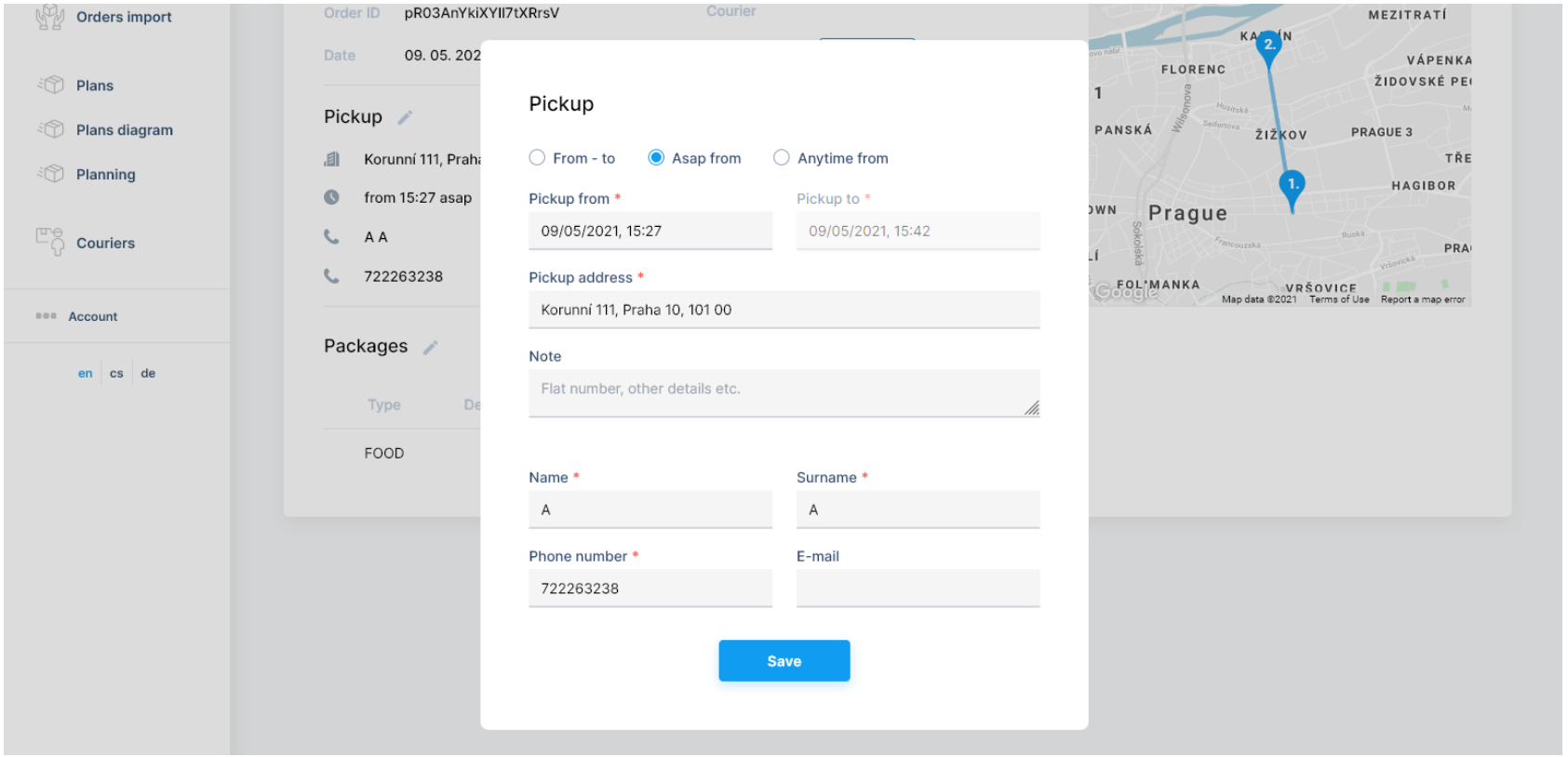
Obrázek A.12 Implementace přehledu plánů



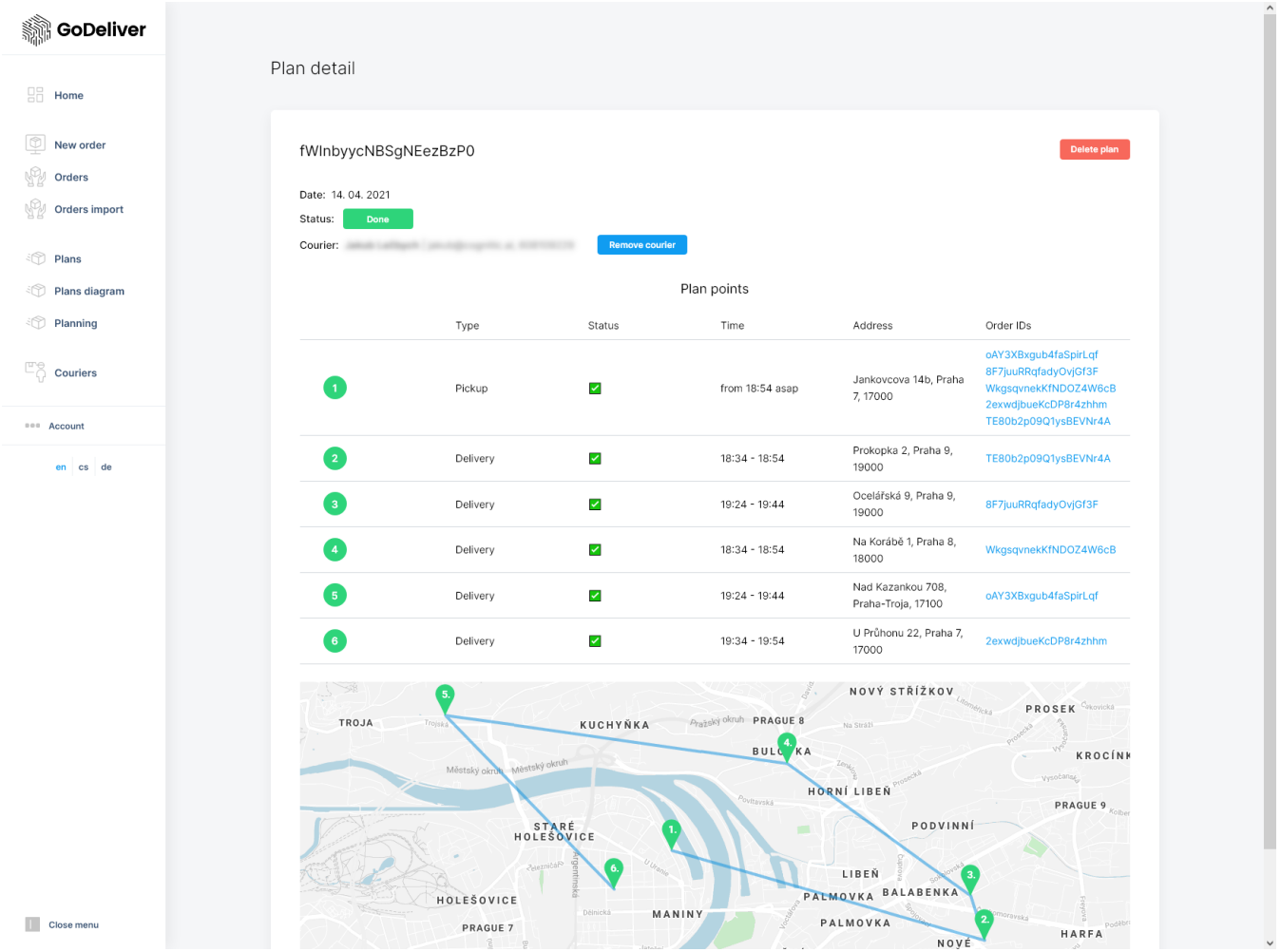
Obrázek A.13 Implementace přehledu kurýrů



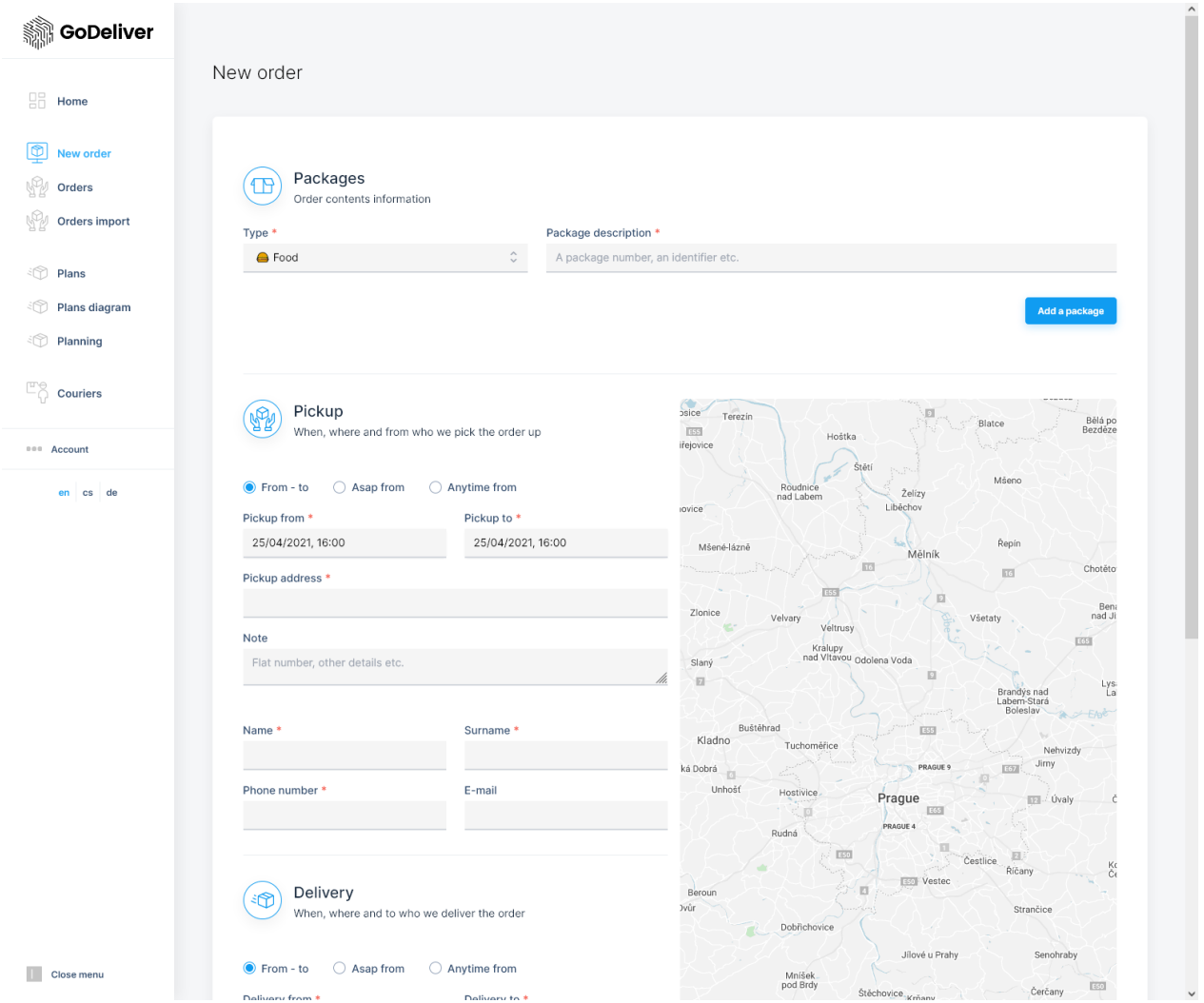
Obrázek A.14 Implementace detailu zásilky



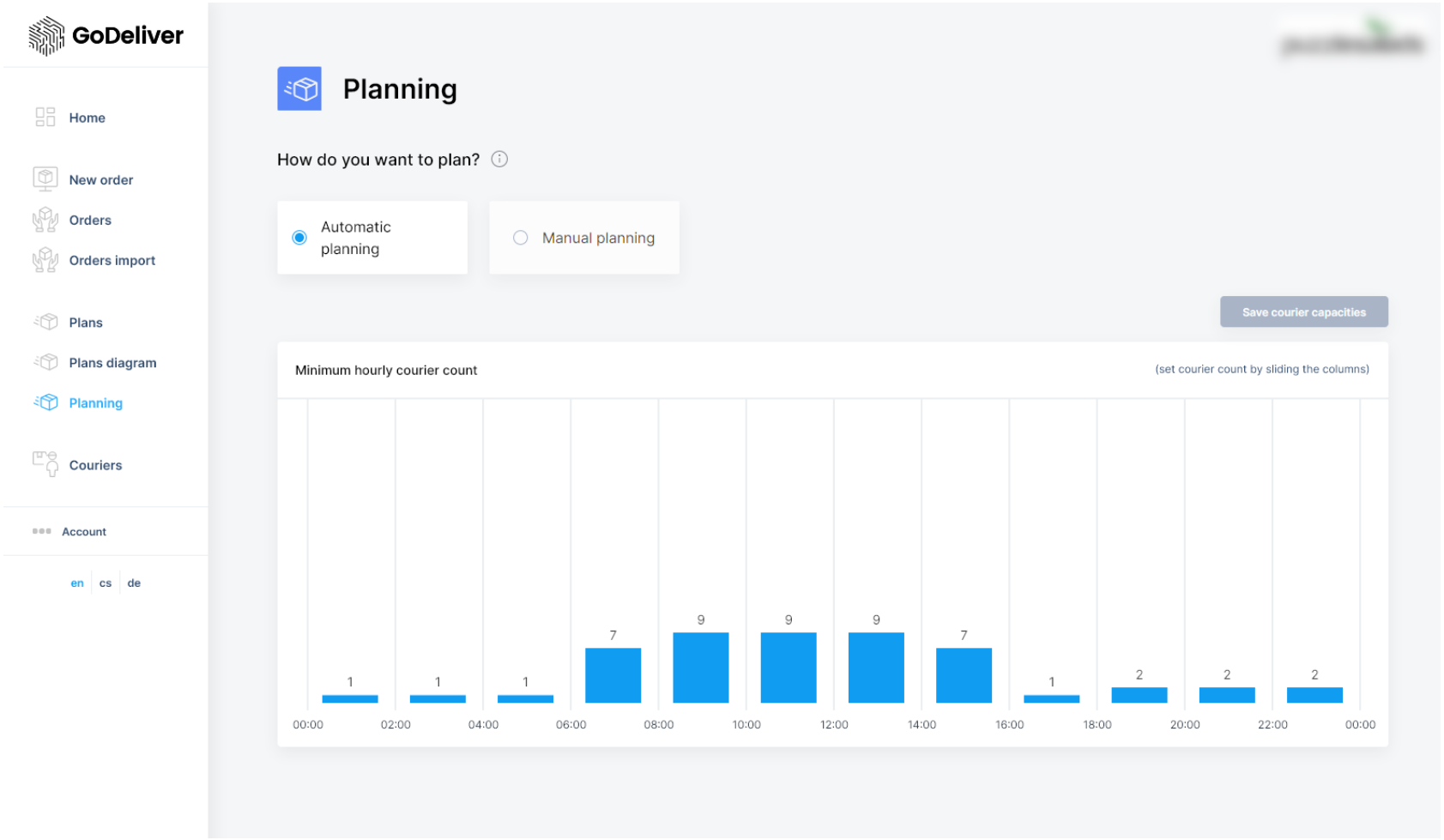
Obrázek A.15 Implementace editace zásilky



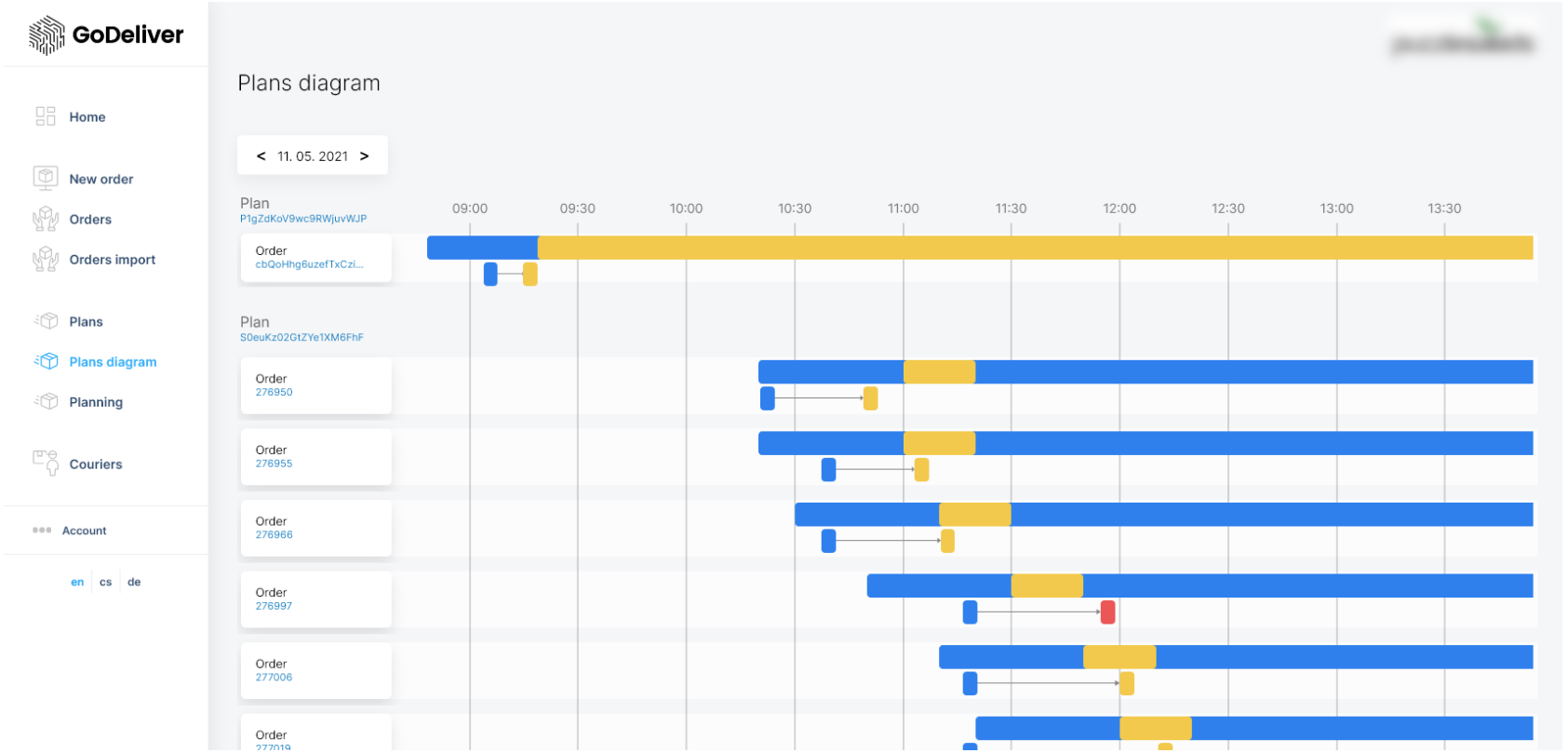
Obrázek A.16 Implementace detailu plánu



Obrázek A.17 Implementace vytvoření zásilky



Obrázek A.18 Implementace nastavení plánovače



Obrázek A.19 Implementace diagramu plánů

Bibliografie

1. CONTRIBUTORS, MDN. *HTML: HyperText Markup Language* [online]. Mozilla, 2021-04-14 [cit. 2021-05-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML.
2. CONTRIBUTORS, MDN. *CSS: Cascading Style Sheets* [online]. Mozilla, 2021-04-14 [cit. 2021-05-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS.
3. CONTRIBUTORS, MDN. *Cascade and inheritance* [online]. Mozilla, 2021-01-27 [cit. 2021-05-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Cascade_and_inheritance.
4. *Node.js* [online]. OpenJS Foundation, 2021 [cit. 2021-05-13]. Dostupné z: https://nodejs.org.
5. CONTRIBUTORS, MDN. *What is JavaScript?* [Online]. Mozilla, 2021-04-27 [cit. 2021-05-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.
6. *TypeScript for the New Programmer* [online]. Microsoft, 2021-05-10 [cit. 2021-05-12]. Dostupné z: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.
7. *React* [online]. Facebook Inc., 2021. Ver. 17.0.2 [cit. 2021-04-23]. Dostupné z: https://reactjs.org/.
8. *Introducing JSX* [online]. Facebook Inc., 2021 [cit. 2021-05-12]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html.
9. *Introducing Hooks* [online]. Facebook Inc., 2021 [cit. 2021-05-12]. Dostupné z: https://reactjs.org/docs/hooks-intro.html.
10. *Create a Next.js App* [online]. Vercel, 2021 [cit. 2021-05-12]. Dostupné z: https://nextjs.org/learn/basics/create-nextjs-app?utm_source=next-site&utm_medium=homepage-cta&utm_campaign=next-website.
11. *About npm* [online]. npm, 2021 [cit. 2021-05-10]. Dostupné z: https://docs.npmjs.com/about-npm.
12. *package.json* [online]. npm, 2021. Ver. 7.x [cit. 2021-05-10]. Dostupné z: https://docs.npmjs.com/cli/v7/configuring-npm/package-json.
13. *package-lock.json* [online]. npm, 2021. Ver. 7.x [cit. 2021-05-10]. Dostupné z: https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json.
14. *CLI commands* [online]. npm, 2021. Ver. 7.x [cit. 2021-05-10]. Dostupné z: https://docs.npmjs.com/cli/v7/commands.

15. *scripts* [online]. npm, 2021. Ver. 7.x [cit. 2021-05-10]. Dostupné z: https://docs.npmjs.com/cli/v7/using-npm/scripts.
16. SASSI, Rakia Ben. *Compiler vs. Interpreter: Know The Difference And When To Use Each Of Them* [online]. Medium, 2021-01 [cit. 2021-05-10]. Dostupné z: https://betterprogramming.pub/compiler-vs-interpreter-d0a12ca1c1b6.
17. GUIMARÃES, George. *What is a linter and why your team should use it?* [Online]. SourceLevel, 2020-06 [cit. 2021-05-10]. Dostupné z: https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it.
18. SACOLICK, Isaac. *What is CI/CD? Continuous integration and continuous delivery explained* [online]. IDG Communications, Inc., 2020-01 [cit. 2021-05-05]. Dostupné z: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html.
19. *Snyk* [online]. 2021 [cit. 2021-05-04]. Dostupné z: https://snyk.io/.
20. *GitLab* [online]. 2021 [cit. 2021-05-04]. Dostupné z: https://gitlab.com.
21. *GitHub* [online]. 2021 [cit. 2021-05-04]. Dostupné z: https://github.com.
22. SEROTER, Richard. *Exploring the ENTIRE DevOps Toolchain for (Cloud) Teams* [online]. InfoQ, 2014-05 [cit. 2021-05-05]. Dostupné z: https://www.infoq.com/articles/devops-toolchain/.
23. ANASTASOV, Marko. *CI/CD Pipeline: A Gentle Introduction* [online]. Rendered Text, 2019-04 [cit. 2021-05-05]. Dostupné z: https://semaphoreci.com/blog/cicd-pipeline.
24. *Slack* [online]. Slack Technologies, Inc., 2021 [cit. 2021-05-12]. Dostupné z: https://slack.com.
25. *GitHub Actions* [online]. 2021 [cit. 2021-05-09]. Dostupné z: https://docs.github.com/en/actions.
26. *Events that trigger workflows* [online]. 2021 [cit. 2021-05-09]. Dostupné z: https://docs.github.com/en/actions/reference/events-that-trigger-workflows.
27. *Workflow syntax for GitHub Actions* [online]. 2021 [cit. 2021-05-09]. Dostupné z: https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions.
28. *GitHub Marketplace - Actions* [online]. 2021 [cit. 2021-05-09]. Dostupné z: https://github.com/marketplace?type=actions.
29. PAVLÍČEK, Josef. *User Interface Testing* [online]. ČVUT, Fakulta informačních technologií, 2020 [cit. 2021-05-03]. Dostupné z: https://docs.google.com/presentation/d/1t-4kCvHJSqpzqff30JhoAzPvaJQQElJ990geai3K9e8/edit#slide=id.ga8c1693005_0_368.
30. BEDŘICHOVÁ, Petra. *Návrh uživatelského rozhraní webové aplikace* [online]. Vysoká škola ekonomická v Praze: Katedra multimédií, 2018-08 [cit. 2021-04-28]. Dostupné z: https://kme.vse.cz/wp-content/uploads/page/534/10.-N%C3%A1vrh-u%C5%BEivatelsk%C3%A9ho-rozhran%C3%AD-webov%C3%A9-aplikace.pdf.
31. NIELSEN, Jakob. Enhancing the Explanatory Power of Usability Heuristics. In: CHI '94: Association for Computing Machinery, 1994, s. 152–158. ISBN 0897916506. Dostupné z DOI: 10.1145/191666.191729.
32. DESIGNORATE, Team members. *Applying Heuristic Evaluation in Usability Testing* [online]. Designorate, 2021 [cit. 2021-05-03]. Dostupné z: https://www.designorate.com/applying-heuristic-evaluation-in-usability-testing/.
33. *Figma* [online]. Figma, 2021 [cit. 2021-05-13]. Dostupné z: https://www.figma.com.

Obsah přiloženého média

- `src`
  - `implementation` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF