

Progresivní webová aplikace pro monitoring strojů

Bc. Filip Jadrníček

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Filip Jadrníček
Osobní číslo: A23404
Studijní program: N0613A140022 Informační technologie
Specializace: Kybernetická bezpečnost
Forma studia: Kombinovaná
Téma práce: Progressivní webová aplikace pro monitoring strojů
Téma práce anglicky: Progressive Web Application for Machine Monitoring

Zásady pro vypracování

- Popište základní principy a problematiku vývoje progresivních webových aplikací.
- Shrňte možnosti vývoje aplikací pro mobilní zařízení a porovnejte výhody a nevýhody jednotlivých přístupů.
- Stanovte funkcionální a nefunkcionální požadavky na aplikaci pro monitoring tryskacích zařízení.
- Navrhněte uživatelské rozhraní aplikace a zaměřte se na uživatelský zážitek, tak aby zásadní funkcionality byla jednoduše a intuitivně dostupná.
- Implementujte aplikaci dle výše uvedených požadavků tak, aby splňovala doporučení PWA.
- Shrňte dosažené výsledky a vhodnost zvoleného přístupu pro daný případ užití.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. *Vue.js*. Online. 2014. Dostupné z: <https://vuejs.org/>. [cit. 2024-10-23].
2. *Ionic*. Online. Dostupné z: <https://ionicframework.com/docs>. [cit. 2024-10-23].
3. *Tailwind*. Online. 2019. Dostupné z: <https://tailwindcss.com/>. [cit. 2024-10-23].
4. *Vite*. Online. 2020. Dostupné z: <https://vite.dev/>. [cit. 2024-10-23].
5. *Frontend Architecture for Design Systems*. O'Reilly Media, 2016. ISBN 9781491926789.
6. *Vite PWA*. Online. Dostupné z: <https://vite-pwa-org.netlify.app/>. [cit. 2024-10-23].
7. TIDWELL, Jenifer; VALENCIA, Aynne a BREWER, Charles. *Designing Interfaces*. 3rd ed. O'Reilly Media, 2020. ISBN 9781492051961.

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.
- Prohlašuji, že při tvorbě této práce jsem použil nástroj generativního modelu AI ChatGPT (<https://chat.openai.com>) za účelem jazykové úpravy textu, zlepšení stylistiky a vyhledávání důvěryhodných zdrojů. Po použití tohoto nástroje jsem provedl kontrolu obsahu a přebírám za něj plnou odpovědnost.

Ve Zlíně, dne

.....

podpis autora

ABSTRAKT

Tato diplomová práce se zaměřuje na návrh a vývoj progresivní webové aplikace pro monitoring strojů. Cílem bylo vytvořit multiplatformní nástroj, který usnadní správu provozních a ekonomických dat. Aplikace byla vyvinuta pomocí frameworku *Vue.js* a *Ionic* a využívá technologie jako *JSON Server*, *Tailwind CSS* a knihovnu *Pinia*. Práce detailně popisuje klíčové obrazovky jako dashboard, statistiky, ekonomiku a provozní knihu. Důraz byl kladen na intuitivní ovládání, responzivní design a přehlednost. Výsledná aplikace nabízí uživatelsky přívětivé prostředí a plnohodnotnou funkcionalitu pro sledování a správu výrobních zakázek.

Klíčová slova: progresivní webová aplikace, monitoring strojů, *Vue.js*, multiplatformní vývoj, UX design, *Ionic*

ABSTRACT

This diploma thesis focuses on the design and development of a progressive web application for machine monitoring. The goal was to create a cross-platform tool that facilitates the management of operational and economic data. The application was developed using the *Vue.js* and *Ionic frameworks* and incorporates technologies such as *JSON Server*, *Tailwind CSS*, and the *Pinia library*. The thesis provides a detailed description of key interfaces including the dashboard, statistics, economy, and operations log. Emphasis was placed on intuitive control, responsive design, and clarity. The resulting application offers a user-friendly interface and full functionality for tracking and managing production tasks.

Keywords: progressive web application, machine monitoring, *Vue.js*, cross-platform development, UX design, *Ionic*

Rád bych poděkoval svému vedoucímu diplomové práce **Ing. Radku Valovi, Ph.D.**, za odborné vedení, cenné rady a možnost spolupracovat na tématu, které mě po celou dobu upřímně bavilo a profesně obohatilo. Jeho vstřícnost a konstruktivní přístup byly pro mě velkou inspirací.

Dále děkuji své rodině a blízkým za podporu, trpělivost a víru ve mě, která mě provázela nejen během psaní této práce, ale po celou dobu mého studia. Bez jejich opory by cesta k cíli byla mnohem obtížnější.

Na závěr bych rád uvedl citát z knihy *Hovory k sobě* od Marca Aurelia, který mě provázel nejen při psaní této práce, ale i v životních rozhodnutích:

„Nezáleží na tom, co se ti přihodí, ale jak na to reaguješ.“

OBSAH

ÚVOD.....	15
TEORETICKÁ ČÁST.....	16
1 MODERNÍ WEBOVÉ TECHNOLOGIE.....	17
1.1 JAVASCRIPT.....	17
1.1.1 STANDARDIZACE A VÝVOJ ECMAScriptu	17
1.1.2 VYUŽITÍ JAVAScriptu NA FRONT-ENDU	18
1.1.3 KNIHOVNY PRO MANIPULACI S DOM.....	18
1.1.4 RÁMCE TYPU MVC	18
1.1.5 VIRTUÁLNÍ DOM A KOMPONENTOVÉ FRAMEWORKY	18
1.1.6 JAVAScript NA BACK-ENDU: NODE.JS	19
1.1.7 JAVAScript A MOBILNÍ VÝVOJ.....	19
1.2 SOUČASNÉ TRENDY VE VÝVOJI V JAZYCE JAVAScript	19
1.2.1 POSTAVENÍ JAZYKA JAVAScript	20
1.2.2 NÁSTROJE A VÝVOJOVÁ PROSTŘEDÍ	20
1.2.3 FRAMEWORKY A KNIHOVNY.....	20
1.2.4 MODULOVÉ BUNDLERY	20
1.2.5 JEDNOTKOVÉ TESTOVÁNÍ	21
1.2.6 VÝZVY PŘI VÝVOJI V JAZYCE JAVAScript	21
1.3 RŮST A PŘÍNOSY JAZYKA TYPEScript.....	21
1.3.1 DŮVODY PŘECHODU NA TYPEScript	21
1.4 JSDOC.....	21
1.4.1 JAK JSDOC ŘEŠÍ TYPOVÁNÍ	22
1.4.2 JSDOC V PRAXI	22
1.4.3 VÝHODY A NEVÝHODY JSDOC	22
1.4.4 SROVNÁNÍ TYPEScript VS. JSDOC	22
1.4.5 ZÁVĚR SROVNÁNÍ.....	23
1.5 KOMPONENTOVÉ FRAMEWORKY: REACT, VUE, ANGULAR.....	23
1.5.1 VUE.JS.....	23
1.5.2 ANGULAR.....	24
1.5.3 REACT.JS	24
1.6 ROLE WEB COMPONENTS.....	25
1.6.1 CO JSOU TO WEB COMPONENTS	25
1.6.2 VÝHODY WEB COMPONENTS	25

1.6.3	SHRNUTÍ.....	26
1.7	MIKRO-FRONTENDY	26
1.7.1	CO JSOU TO MIKRO-FRONTENDY	27
1.7.2	ARCHITEKTURA MODULE FEDERATION.....	27
1.7.3	PŘÍNOSY MIKRO-FRONTENDŮ S MODULE FEDERATION	27
1.7.4	TYPY REPOZITÁŘOVÝCH STRATEGIÍ.....	27
1.7.5	CLIENT-SIDE VS. SERVER-SIDE KOMPOZICE	27
1.8	SÍŤOVÉ A VÝKONOVÉ STANDARDY (HTTP/3, QUIC, BROTLI, CDN)	28
1.8.1	ÚVOD DO PROBLEMATIKY HTTP PROTOKOLŮ.....	28
1.8.2	HTTP/3 A QUIC	29
1.8.3	BROTLI – POKROČILÁ KOMPRESNÍ TECHNIKA PRO WEBOVÉ APLIKACE.....	30
1.8.4	VÝZNAM CONTENT DELIVERY NETWORKS (CDN)	30
2	PROGRESIVNÍ WEBOVÉ APLIKACE (PWA).....	32
2.1	DEFINICE PROGRESIVNÍ WEBOVÉ APLIKACE	32
2.2	TECHNICKÉ VLASTNOSTI PWA.....	32
2.3	POČÁTKY A VÝVOJ PWA	33
2.4	MILNÍKY VÝVOJE PWA	33
2.4.1	VZNIK POJMU A PRVNÍ DEMONSTRACE.....	33
2.4.2	STANDARDIZACE A PODPORA NAPŘÍČ PLATFORMAMI	33
2.4.3	INTEGRACE DO APLIKAČNÍCH OBCHODŮ	33
2.4.4	PŘÍCHOD POKROČILÝCH WEBOVÝCH API	34
2.5	DŮLEŽITOST PWAS V MODERNÍM VÝVOJI.....	34
2.6	KLÍČOVÉ VLASTNOSTI.....	34
2.6.1	RESPONZIVITA A ADAPTIVNÍ LAYOUT	34
2.6.2	OFFLINE-FIRST (SERVICE WORKERS, APP SHELL).....	35
2.6.3	INSTALOVATELNOST, PUSH NOTIFIKACE, WEB APP MANIFEST	36
2.7	BEZPEČNOSTNÍ A PROVOZNÍ POŽADAVKY.....	38
2.7.1	HTTPS	39
2.7.2	CONTENT SECURITY POLICY (CSP).....	39
2.7.3	CROSS-ORIGIN RESOURCE SHARING (CORS)	40
2.7.4	ZÁVĚREČNÉ POZNÁMKY K OCHRANĚ WEBOVÝCH APLIKACÍ	41
3	MOŽNOSTI VÝVOJE MOBILNÍCH APLIKACÍ	42
3.1	NATIVNÍ PŘÍSTUP (ANDROID KOTLIN/JAVA, IOS SWIFT)	42
3.1.1	VÝVOJ PRO ANDROID: KOTLIN A JAVA.....	42

3.1.2	VÝVOJ PRO IOS: SWIFT A OBJECTIVE-C.....	43
3.1.3	SHRnutí.....	43
3.2	MULTIPLATFORMNÍ KOMPILOVANÉ FRAMEWORKY (FLUTTER, REACT NATIVE, LYNX).....	44
3.2.1	FLUTTER.....	44
3.2.2	REACT NATIVE.....	44
3.2.3	LYNX.....	45
3.2.4	SHRnutí.....	45
3.3	HYBRIDNÍ ŘEŠENÍ (IONIC, CORDOVA)	45
3.3.1	APACHE CORDOVA.....	46
3.3.2	IONIC	46
3.4	KOMPARATIVNÍ ANALÝZA.....	46
3.4.1	VÝKON A UŽIVATELSKÝ ZÁŽITEK	46
3.4.2	NÁKLADY NA VÝVOJ A ÚDRŽBU	47
3.4.3	PŘÍSTUP K HARDWARU, PUBLIKACE, AKTUALIZACE.....	48
3.4.4	BEZPEČNOST A SPRÁVA DAT.....	49
3.4.5	ZÁVĚR KOMPARATIVNÍ ANALÝZY	50
4	UI/UX DESIGN	51
4.1	UŽIVATELSKÉ ROZHRAŇÍ (UI)	51
4.1.1	VIDITELNOST STAVU SYSTÉMU.....	51
4.1.2	SYSTÉM ODPOVÍDAJÍCÍ REÁLNÉMU SVĚTU	51
4.1.3	UŽIVATELSKÁ KONTROLA A SVOBODA	51
4.1.4	KONZISTENCE A STANDARDY	52
4.1.5	PREVENCE CHYB	52
4.1.6	ROZPOZNÁNÍ MÍSTO VYBAVOVÁNÍ	52
4.1.7	FLEXIBILITA A EFEKTIVITA POUŽITÍ.....	52
4.1.8	ESTETICKÝ A MINIMALISTICKÝ DESIGN	52
4.1.9	POMOC UŽIVATELŮM ROZPOZNAT, DIAGNOSTIKOVAT A ŘEŠIT CHYBY	53
4.1.10	NÁPOVĚDA A DOKUMENTACE.....	53
4.2	UŽIVATELSKÝ ZÁŽITEK (UX)	53
4.2.1	NÁPOVĚDA A DOKUMENTACE.....	53
4.2.2	ZAMĚŘENÍ UŽIVATELE.....	54
4.2.3	POUŽITELNOST A PŘÍSTUPNOST	54
4.2.4	KONZISTENCE A HIERARCHIE	54
4.2.5	KONTROLA A SVOBODA UŽIVATELE	54

4.2.6	MINIMALIZACE KOGNITIVNÍ ZÁTĚŽE	55
4.2.7	TESTOVÁNÍ A ZPĚTNÁ VAZBA	55
4.2.8	DESIGN ZALOŽENÝ NA KONTEXTU	55
PRAKTICKÁ ČÁST.....		56
5	POPIS APLIKACE A JEJÍ ÚČEL.....	57
5.1	HLAVNÍ FUNKCE APLIKACE	57
5.1.1	SBĚR A ZPRACOVÁNÍ DAT	57
5.1.2	DIGITÁLNÍ PROVOZNÍ KNIHA.....	57
5.1.3	SLEDOVÁNÍ PRÁCE ZAMĚSTNANCŮ	58
5.1.4	STATISTIKY A PROVOZNÍ DATA STROJŮ	58
5.1.5	EKONOMICKÝ PŘEHLED SPOTŘEBY	58
5.1.6	PRÁCE S MANUÁLY	58
5.1.7	PŘÍSTUP DO SYSTÉMU	58
6	FUNKCIONÁLNÍ A NEFUNKCIONÁLNÍ POŽADAVKY NA APLIKACI PRO MONITORING TRYSKACÍCH ZAŘÍZENÍ	59
6.1	FUNKCIONÁLNÍ POŽADAVKY	59
6.1.1	CHARAKTERISTIKA FUNKCIONÁLNÍCH POŽADAVKŮ	59
6.2	NEFUNKCIONÁLNÍ POŽADAVKY	59
6.2.1	CHARAKTERISTIKA NEFUNKCIONÁLNÍCH POŽADAVKŮ	59
6.3	POŽADAVKY NA SYSTÉM	60
6.4	FUNKCIONÁLNÍ POŽADAVKY NA SYSTÉM	60
6.5	NEFUNKCIONÁLNÍ POŽADAVKY NA SYSTÉM.....	61
7	TECHNOLOGICKÁ ARCHITEKTURA PROJEKTU	63
7.1	ARCHITEKTURA KLIENT-SERVER.....	63
7.1.1	DŮVOD ZAVEDENÍ KLIENT-SERVER ARCHITEKTURY	63
7.1.2	POROVNÁNÍ S PEER-TO-PEER	63
7.2	POPIS ZPŮSOBU KOMUNIKACE S API.....	64
7.2.1	JAK FUNGUJE VOLÁNÍ API.....	64
7.2.2	FORMÁT DAT	64
8	NÁVRH APLIKACE	65
8.1	GRAFICKÝ STYL A BAREVNÁ IDENTITA APLIKACE	65
8.2	NÁVRH OBRAZOVEK.....	65
8.2.1	PŘIHLAŠOVACÍ OBRAZOVKA.....	66
8.2.2	NAVIGACE V APLIKACI	66
8.2.3	OBRAZOVKA PRO OZNÁMENÍ	68

8.2.4	OBRAZOVKA STAVU STROJE	71
8.2.5	OBRAZOVKA STATISTIKY STROJE	72
8.2.6	OBRAZOVKA PROVOZNÍ KNIHY	73
8.2.7	OBRAZOVKA EXPORTU DAT	74
8.2.8	OBRAZOVKA PRO SEZNAM MANUÁLŮ	75
9	IMPLEMENTACE APLIKACE	77
9.1	SOUBOROVÁ STRUKTURA	77
9.1.1	API 78	
9.1.2	COMPONENTS	78
9.1.3	CONSTANTS	78
9.1.4	DUMMYJSON	79
9.1.5	ROUTER	79
9.1.6	STORES	79
9.1.7	THEME	79
9.1.8	UTILS	80
9.1.9	VIEWS	80
9.1.10	HLAVNÍ SOUBORY	80
9.2	NAVIGACE A SMĚROVÁNÍ (ROUTING)	80
9.2.1	STATICKÝ ROUTING	80
9.2.2	DYNAMICKÝ ROUTING	81
9.2.3	NAVIGACE POMOCÍ KOMPONENT A FUNKCÍ	81
9.2.4	ZÁVĚR K SMĚROVÁNÍ	81
9.3	KOMPONENTOVÁ ARCHITEKTURA	82
9.3.1	STRUKTURA SOUBORŮ KOMPONENT	82
9.3.2	IMPORT A REGISTRACE KOMPONENTY	82
9.3.3	POUŽITÍ KOMPONENTY V ŠABLONĚ	82
9.4	PŘENOS DAT MEZI VRSTVAMI APLIKACE	83
9.4.1	PŘÍJEM A SPOTŘEBA DAT V KOMPONENTĚ	83
9.4.2	VNOŘOVÁNÍ KOMPONENT A PROP-DRILLING VS. PROVIDE/INJECT	85
9.4.3	CENTRÁLNÍ SPRÁVA STAVU POMOCÍ PINIA	86
9.4.4	MOCK BACK-END – JSON SERVER	87
9.4.5	KOMUNIKACE FRONT-ENDU S JSON SERVEREM / REST API	87
9.5	PŘIHLÁŠENÍ UŽIVATELE A AUTENTIZACE	88
9.5.1	INICIALIZACE A KONFIGURACE JSON-SERVER-AUTH	88
9.5.2	ZÍSKÁNÍ JWT TOKENU A HASHOVÁNÍ HESEL	89

9.5.3	OBNOVA RELACE A PRÁCE S TOKENEM NA FRONT-ENDU	90
9.5.4	CHRÁNĚNÝ ROUTING VE VUE ROUTERU	91
9.5.5	OCHRANA API.....	92
9.5.6	PŘIHLAŠOVACÍ FORMULÁŘ	92
9.5.7	SHRnutí PŘIHLÁŠENÍ UŽIVATELE A AUTENTIZACE	93
9.6	DETAILNÍ ROZBOR OBRAZOVKY „KALKULÁTOR“	93
9.6.1	PŘEDÁVÁNÍ DAT A VÝPOČTY	94
9.6.2	VÝPOČTOVÉ FUNKCE	95
9.6.3	KOMPONENTY A ARCHITEKTURA	95
10	FINÁLNÍ VERZE APLIKACE	97
10.1	STAV APLIKACE	97
10.2	STATISTIKY.....	98
10.3	EKONOMIKA.....	99
10.3.1	ZOBRAZENÍ SUMACE	99
10.4	PROVOZNÍ KNIHA	100
10.4.1	SEZNAM ZAKÁZEK.....	101
ZÁVĚR.....		103
SEZNAM POUŽITÉ LITERATURY		105
SEZNAM OBRÁZKŮ		111
SEZNAM TABULEK.....		112
SEZNAM KÓDŮ.....		113
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK		115

ÚVOD

Digitalizace průmyslových procesů a zavádění konceptů Průmyslu 4.0 si kladou za cíl zefektivnit výrobu, snížit náklady a zvýšit transparentnost provozu. Klíčovým předpokladem pro dosažení těchto cílů je schopnost sbírat a analyzovat provozní data v reálném čase a na jejich základě činit kvalifikovaná rozhodnutí.

Cílem práce je navrhnout a implementovat moderní progresivní webovou aplikaci (*Progressive Web App – PWA*) pro monitoring strojů, která bude multiplatformní, uživatelsky přívětivá a funkčně přizpůsobená potřebám výrobního prostředí. Aplikace by měla umožnit sledování aktuálního stavu zařízení, záznam zakázek, evidenci nákladů, vyhodnocování statistik a nabídnout užitečné podpůrné nástroje, jako jsou technické kalkulátory.

Výsledná aplikace je vyvíjena s využitím moderních technologií – konkrétně frameworku *Vue.js* s *Composition API*, rozhraní *Ionic* pro tvorbu mobilního uživatelského prostředí, knihovny *Pinia* pro správu stavu aplikace, a *Tailwind CSS* pro efektivní stylování komponent. Zaměřuje se na multiplatformní využití, a je tak dostupná jak na mobilních zařízeních, tak na desktopu bez nutnosti instalace.

Struktura práce je rozdělena do dvou hlavních částí. **Teoretická část** popisuje principy vývoje progresivních webových aplikací, srovnání nativního, hybridního a multiplatformního přístupu a výhody jednotlivých použitých nástrojů. **Praktická část** se věnuje návrhu uživatelského rozhraní (*UI*), datovým tokům, návrhu komponentové architektury, správě stavu pomocí knihovny *Pinia* a řešení autentizace pomocí *JWT*. Součástí je i dokumentace důležitých obrazovek jako dashboard, statistiky, ekonomika a provozní kniha.

I TEORETICKÁ ČÁST

1 MODERNÍ WEBOVÉ TECHNOLOGIE

JavaScript se stal klíčovým jazykem webového vývoje a jeho ekosystém se v posledních letech výrazně rozšířil. Vedle původního využití na straně klienta dnes zahrnuje i serverové prostředí, mobilní aplikace a nástroje pro vývoj rozsáhlých systémů. Důležitou roli hrají moderní frameworky, standardy a architektonické přístupy, které umožňují efektivní, škálovatelný a udržitelný vývoj webových aplikací.

1.1 JavaScript

JavaScript, jeden z pilířů moderního webového vývoje, vznikl v roce 1995 v době tzv. „války prohlížečů“. Jeho autor **Brendan Eich**, tehdejší vývojář společnosti *Netscape*, dostal za úkol vytvořit nový programovací jazyk, který by umožnil přidat do webových stránek interaktivní prvky. Měl být flexibilní, snadno pochopitelný a rychle nasaditelný — první verze vznikla během pouhých 10 pracovních dní. [1]

Při vývoji byly použity prvky z jazyků jako *Scheme* (funkce jako hodnoty), *Perl* (*weak-typing*), *Self* (*prototypové dědičnosti*) a *Java* (*syntax a konvence*). První název byl *Mocha*, později se jazyk publikoval jako *JavaScript*, což byl z marketingového hlediska záměr – využít tehdejší popularitu *Javy*. [1]

1.1.1 Standardizace a vývoj ECMAScriptu

Vzhledem k nekompatibilitám mezi různými implementacemi jazyků *Netscape* a *Microsoftu* (např. *JScript*) byla v roce 1997 jazyková specifikace předána k normalizaci organizaci *Ecma International*. Výsledkem je standard *ECMAScript*, jehož název se používá pro jednotlivé verze specifikace, zatímco název *JavaScript* zůstal de facto označením samotného jazyka. [1]

Standardizace a vývoj **ECMAScriptu**:

- **ES3 (1999)**: podpora regulárních výrazů, výjimek a nových řídicích struktur. [1]
- **ES5 (2009)**: funkce vyššího řádu (*map*, *reduce*), podpora *JSON*, gettery/settery. [1]
- **ES6/ES2015**: přelomová verze s přidáním *let/const*, tříd, modulů, *Promises*, *arrow function* (*=>*). [1]
- **ES2017–2020**: postupné přidávání podpory pro *async/await*, operátory *?* a *??*, metody *flat*, *flatMap*, *finally* a další. [1]

1.1.2 Využití JavaScriptu na front-endu

Zpočátku sloužil *JavaScript* hlavně pro drobné animace a interaktivní prvky, ale s příchodem *AJAX* (1999) se stal klíčovým nástrojem pro komunikaci s *back-endem* bez nutnosti opětovného načtení stránky. Od té doby se rozvíjel jako plnohodnotný jazyk pro klientskou logiku. [1]

1.1.3 Knihovny pro manipulaci s DOM

S rostoucí složitostí webových aplikací se ukázalo, že samotný *JavaScript* není pro manipulaci s *DOM* vždy efektivní. Proto vznikly knihovny jako:

- **Prototype (2005):** inspirováno *Ruby*, přidávalo metody přímo do vestavěných objektů *JavaScript* (např. *Array*, *String*). [1]
- **jQuery (2006):** zaměřen na zjednodušení práce s *DOM*, událostmi a *AJAX*. Stalo se nejpopulárnější knihovnou své doby. [1]

1.1.4 Rámce typu MVC

Při vývoji složitějších aplikací se začaly využívat architektonické vzory jako *MVC*:

- **Backbone (2010):** využívá *model-view-presenter* architekturu, vhodný pro *SPA*. [1]
- **Angular:** komplexní framework od společnosti *Google*, jenž poskytuje „*all-in-one*“ řešení s důrazem na striktní strukturu a *TypeScript*. [1]
- **React** – knihovna od společnosti *Meta*, zaměřená na *UI* a **virtuální DOM**. Díky modulárnosti a široké ekosystémové podpoře je považována za nejžádanější technologii u zaměstnavatelů. [2][3]
- **Vue:** progresivní *framework* s nízkou vstupní bariérou a flexibilní architekturou. Jeho podíl roste zejména v menších týmech a *start-upech*. [2]

1.1.5 Virtuální DOM a komponentové frameworky

Společnost *Facebook* představil **React** jako nový přístup k vývoji *UI* – zavedl *Virtual DOM*, komponenty a přímou vazbu mezi stavem a zobrazením. Následně se objevily koncepty jako:

- **JSX** – zápis HTML přímo v *JavaScriptu*. [1]
- **Flux/Redux** – centrální správa stavu pomocí čistých funkcí. [1]
- **GraphQL** – dotazovací jazyk pro *API* vyvinutý společností *Facebook*, který umožňuje přesně definovat požadovaná data. [1]

1.1.6 JavaScript na back-endu: Node.js

Node.js, představené **Ryanem Dahlem** v roce 2009, sjednotilo *front-end* a *back-end* vývoj pod jediný jazyk. Běží na enginu **V8** a kombinuje **just-in-time kompilaci** s **událostmi řízenou, neblokující architekturou**. Klíčovým prvkem je **event loop**: na jediné vlákno zpracovává frontu událostí, zatímco knihovna **libuv** deleguje náročné *I/O* do vláknového poolu. Díky tomu dokáže server paralelně obsloužit tisíce spojení bez nutnosti vytvářet nové procesy či vlákna. [5]

Hlavní rysy a přínosy

- **Asynchronní programování** – moderní konstrukce `async/await` zpřehledňují dříve zanořené *callback* a usnadňují práci s *promise*. [5]
- **Ne-blokující I/O** – operace se předají jádru *OS*, *Node.js* mezitím zpracuje další požadavky; ideální pro „*real-time*“ aplikace, *streaming* nebo *chat*. [1][5]
- **Modulární ekosystém** – *CommonJS/ES M* moduly a **npm**, který hostí přes dva miliony balíčků. Mezi nejpoužívanější patří:
 - **Express** – minimalistický *HTTP* framework. [5]
 - **Mongoose** – *ORM* nad *MongoDB*. [5]
 - **Dotenv, Lodash, Jest** a další nástroje pro konfiguraci, utilitní funkce a testování. [5]
- **Škálovatelnost** – možnost spuštění v *clusteru* nebo v kontejnerech, monitoring *event-loopu* a *CPU* pomocí platform typu **N | Solid**. [5]

1.1.7 JavaScript a mobilní vývoj

Díky projektům jako **Apache Cordova**, **Ionic**, **React Native** nebo **NativeScript** lze *JavaScript* využívat i pro tvorbu mobilních aplikací. Tyto nástroje umožňují vývoj multiplatformních řešení s přístupem k nativním funkcím zařízení. [1]

1.2 Současné trendy ve vývoji v jazyce JavaScript

Každoroční průzkum **Developer Ecosystem Survey** (2023) přinesl cenné poznatky o vývoji softwarového inženýrství. Z nejnovější verze 2024 zároveň vyplývá, že i když *JavaScript* zůstává nejpoužívanějším jazykem (61 % respondentů), adopce jazyka *TypeScript* rychle roste – od roku 2017 stoupla z **12 % na 35 %** a dvě třetiny vývojářů píšou více kódu v *TypeScriptu* než v čistém *JavaScriptu*. [3][7]

1.2.1 Postavení jazyka JavaScript

JavaScript zůstává i po sedmi letech **nejpoužívanějším** programovacím jazykem, přestože jeho podíl v posledních třech letech mírně klesá. Hlavním důvodem jeho dominance je nízká vstupní bariéra a široké uplatnění v oblasti „*front-end*“ vývoje. Až 80 % vývojářů jej používá pro tvorbu webových stránek, přičemž 86 % pracuje na front-endu a 34 % rovněž zasahuje do back-endu. [7]

1.2.2 Nástroje a vývojová prostředí

Nejpopulárnějším nástrojem pro vývoj v *JavaScriptu* i *TypeScriptu* je **Visual Studio Code**, které používá více než polovina vývojářů (51 %). Následuje **WebStorm** (21 %) a další *IDE* založené na **IntelliJ** platformě (16 %). [7]

Používání editací a *IDE* se mezi vývojáři často kombinuje – jednoduché úpravy se často řeší v lehčích editorech, zatímco pro větší projekty se volí komplexnější prostředí s širší podporou. [7]

1.2.3 Frameworky a knihovny

Používání frameworků a knihoven je v komunitě *JavaScriptu* téměř univerzální – pouze 8 % vývojářů je nepoužívá. Nejoblíbenějším front-endovým frameworkem je **React** (57 %), následuje **Vue** (32 %) a **Next.js** (27 %). Na čtvrtém místě se umístily **Angular** a **Express**, shodně s 20 %. [2][3][7]

Důvody obliby frameworku **React** spočívají v jeho rané adopci, modularitě, flexibilitě a použití *JSX* – rozšíření syntaxe *JavaScriptu*, které připomíná *HTML* a usnadňuje tvorbu komponent. React umožňuje tvorbu opakovaně použitelných komponent, které tvoří stromovou strukturu celé aplikace. [7]

1.2.4 Modulové bundlery

Bundlery hrají klíčovou roli při správě závislostí a kompilaci kódu v jazyce *JavaScript*. Zatímco **Webpack** zůstává **nejpoužívanějším** (57 %), modernější nástroje **Vite** (38 %) a **esbuild** (11 %) nabízejí podstatně rychlejší startovací časy díky využití **ES Modules** a **on-demand bundlingu**. Porovnání mezi *Vite* a *Webpack* ukázalo, že **Vite** dokázalo zkrátit čas

prvního *dev buildu* z 6 s na méně než 0,5 s a výrazně zmenšilo výslednou velikost balíčku. [4]

1.2.5 Jednotkové testování

Jednotkové testování není mezi vývojáři JavaScriptu příliš rozšířené – 32 % dotazovaných žádné testy nepíše. [7]

- **Jest** (40 %) zůstává nejpoužívanějším frameworkem, následují **Cypress** (15 %) a **Mocha**, **Vitest** či **Playwright** (okolo 10 %). [6][7]
- Aktuální přehled doporučuje pro komplexní *end-to-end* scénáře *Cypress* nebo *Playwright*, pro rychlé běhy unit testů *Jest* či *Vitest* a pro starší *code-base* *Mocha*. [6]

1.2.6 Výzvy při vývoji v jazyce JavaScript

I přes široké využití má JavaScript své slabiny. Asynchronní zpracování sice bylo zlepšeno díky zavedení *async/await*, ale jazyk stále postrádá silné typování. To je důvodem, proč si vývojáři čím dál častěji osvojují **TypeScript**, který přináší typovou bezpečnost a snižuje množství chyb v aplikacích. [7]

1.3 Růst a přínosy jazyka TypeScript

Zatímco JavaScript je jazykem s dynamickým typováním, *TypeScript* umožňuje vývojářům psát bezpečnější kód díky statickému typování. Podle **State of JS 2024** jej 67 % respondentů upřednostňuje před čistým *JavaScriptem* a trend jeho adopce stále stoupá. [3]

1.3.1 Důvody přechodu na TypeScript

TypeScript je plně kompatibilní s jazykem *JavaScript* což umožňuje postupnou migraci stávajících projektů. Podporuje nejnovější standardy *ECMAScript* a často zavádí nové jazykové funkce ještě před tím, než jsou dostupné ve webových prohlížečích. Díky tomu mohou vývojáři využívat moderní přístupy a zjednodušit si práci bez obav o zpětnou kompatibilitu. [7]

1.4 JSDoc

JSDoc je formátovací jazyk pro komentáře v jazyce *JavaScript*, který se používá k dokumentaci kódu a doplňování typové informace. Byl představen v roce 1999 a funguje podobně jako **Javadoc** v jazyce *Java* nebo **YARD** v *Ruby*. Pomocí komentářových značek

jako `@param`, `@returns` nebo `@throws` lze popsat funkce, třídy i datové typy. Díky podpoře ze strany *TypeScriptu* lze tyto informace následně využít i pro statickou kontrolu typů v čistém *JavaScriptu*. [8][9]

1.4.1 Jak JSDoc řeší typování

Anotace v *JSDoc* umožňují vývojářům popsat typy **bez nutnosti přechodu na plnohodnotný TypeScript**. Pomocí jednoduché syntaxe lze definovat typy proměnných, argumentů a návratových hodnot funkcí. Typové informace lze dokonce importovat z `.d.ts` souborů, čímž se otevírá možnost využívat i složitější typové konstrukce bez kompilace. Tato metoda je vhodná například pro menší projekty, rychlé skripty nebo týmy, které se chtějí vyhnout překladu pomocí transpileru. [8][9]

1.4.2 JSDoc v praxi

Vzhledem k tomu, že *TypeScript* podporuje kontrolu typů v JavaScriptových souborech s *JSDoc* pomocí volby `--checkJs`, lze využívat výhod TypeScriptového linteru i **bez kompilace**. Tento přístup byl úspěšně aplikován v projektu *Webpack*, kde se díky aktivitě komunity podařilo zavést typovou kontrolu pomocí *JSDoc* i ve velkém kódu. Od verze *TypeScript* 2.9 byly do *JSDoc* přidány další vylepšení, která tento způsob ještě více zpřístupnila. [8]

1.4.3 Výhody a nevýhody JSDoc

Mezi výhody **JSDoc** patří nižší složitost při zavádění do stávajícího JavaScriptového projektu, možnost postupné adopce a odstranění nutnosti kompilace. Na druhou stranu jsou anotace zapisovány jako komentáře, což může být méně přehledné, zvláště u složitějších typů. *JSDoc* navíc podporuje jen podmnožinu syntaxe *TypeScriptu*, což může být limitující pro pokročilé případy. [9]

1.4.4 Srovnání TypeScript vs. JSDoc

Rozhodnutí mezi *TypeScriptem* a *JSDoc* závisí na konkrétním scénáři. *TypeScript* nabízí **lepší integraci s vývojovými nástroji** (např. refaktoring, nápověda v editoru) a **čistší syntaxi**, zatímco *JSDoc* může být výhodný tam, kde je potřeba **lehčí řešení bez buildovacího kroku**. Anotace v *JSDoc* jsou však „hlučnější“ – zabírají více místa a jsou méně přehledné než nativní *TypeScriptová* syntaxe. [9]

Například jednoduchá funkce s volitelnými a výchozími parametry se v *TypeScriptu* zapisuje stručně a čitelně, zatímco v *JSDoc* je nutné vše rozepsat v komentáři, což snižuje přehlednost a zvyšuje náročnost na údržbu. [9]

```
function filterByLength(items: string[], minLength: number = 3): string[] {
    return items.filter(item => item.length >= minLength);
}
```

Kód 1: Ukázka typování v jazyce TypeScript

```
/**
 * Vrátí položky, které mají délku alespoň `minLength`.
 * @param {string[]} items
 * @param {number} [minLength=3]
 * @returns {string[]}
 */
function filterByLength(items, minLength = 3) {
    return items.filter(item => item.length >= minLength);
}
```

Kód 2: Ukázka typování pomocí JSDoc

1.4.5 Závěr srovnání

Zatímco *TypeScript* je robustní řešení pro typově bezpečný vývoj, *JSDoc* představuje elegantní způsob, jak do klasického *JavaScriptu* přinést základní typovou kontrolu bez nutnosti přechodu na nový jazyk nebo změny build procesu. V menších projektech nebo při postupné migraci může být *JSDoc* ideální volbou, zatímco v rozsáhlých aplikacích se *TypeScript* jeví jako jasná volba. Oba přístupy však přispívají ke zvýšení kvality a bezpečnosti kódu a jejich využití závisí na konkrétním kontextu. [8][9]

1.5 Komponentové frameworky: React, Vue, Angular

Současné front-endové prostředí dominuje trojice komponentových technologií – **React**, **Vue** a **Angular** – které vývojářům poskytují odlišné, ale vzájemně srovnatelné přístupy k tvorbě uživatelských rozhraní. Každý z těchto frameworků či knihoven staví na deklarativním popisu *UI*, podporuje znovupoužitelné komponenty a usiluje o efektivní práci s *DOM*, ovšem liší se v architektuře, nástrojové podpoře i míře „řízenosti“ vývojového procesu.

1.5.1 Vue.js

Vue.js rozšiřuje standardní *HTML* o šablonovou syntaxi a **reaktivní datový model**, takže vývojář může deklarativně vyjádřit vazbu mezi stavem a zobrazením. Díky vestavěné reaktivitě se změny dat promítnou do *DOM* automaticky, bez nutnosti ručních aktualizací.

Vue je navrženo jako progresivní řešení: lze je postupně nasazovat od drobného „oživení“ statické stránky přes vkládání *Web Component* až po plnohodnotnou *SPA* se server-side renderingem či statickou generací. [10]

Výrazným rysem je formát *.vue* (*Single-File Component*), jenž v jednom souboru sdružuje šablonu, logiku i styly. Komponenty lze psát buď klasickým **Options API** (*data*, *methods*, *lifecycle*) nebo modernějším **Composition API**, které zpřístupňuje stav a lifecycle přímo skrze importované funkce v tagu `<script setup>`. Oba styly pokrývají stejné scénáře; *Options API* je obvykle srozumitelnější pro začátečníky, zatímco *Composition API* nabízí vyšší flexibilitu a lepší opětovnou použitelnost logiky. [10]

1.5.2 Angular

Angular reprezentuje plnohodnotný webový framework spravovaný týmem *Google*. Nabízí rozsáhlou sadu nástrojů, rozhraní *API* a knihoven, která zjednodušuje a sjednocuje celý vývojový proces. Je navržený tak, aby podporoval rychlou tvorbu spolehlivých aplikací, které snadno škálují jak s velikostí týmu, tak s objemem kódu. Vývojářům přináší „pevnou půdu“ v podobě ucelené platformy – od základní struktury projektu přes datovou vrstvu až po testování a *build pipeline*. [11]

1.5.3 React.js

React je lehká *JavaScriptová* knihovna, která se soustředí čistě na vrstvu *view*. Její komponentové jádro umožňuje vytvářet jednoduché prvky (např. tlačítka) i komplexní rozhraní složená z hierarchie komponent. Klíčovou inovací je **JSX** – syntaktické rozšíření *JavaScriptu* připomínající *HTML*, které zpřehledňuje strukturu *UI* přímo v kódu. Interně *React* používá **virtuální DOM**; rozdíly mezi virtuální a skutečnou strukturou stránky počítá při každé změně stavu a do prohlížeče posílá jen nezbytné aktualizace, což zvyšuje rychlost a odezvu aplikací. [12]

React je „*ne-opinionated*“: nenutí konkrétní řešení routování ani správy stavu, takže vývojář si může vybrat knihovny (např. *React Router* či *Redux*) podle potřeb projektu. Kromě webu se *React* uplatňuje i v mobilním vývoji prostřednictvím **React Native**, který umožňuje sdílet část kódu mezi webem a nativními aplikacemi. [12]

1.6 Role Web Components

Od samých počátků webu se vývojáři snažili posunout hranice toho, co platforma umožňuje. Ve snaze o opakovaně použitelné komponenty byly nejprve vytvářeny vlastní knihovny, později vznikaly silnější frameworky jako *jQuery*, *AngularJS* nebo *React*. Až v roce 2013 *Google* oficiálně představil iniciativu **Web Components** jako snahu o zavedení **jednotného, nativního komponentového modelu** napříč všemi prohlížeči. I přes určitou počáteční skepsi se díky podpoře v novém *Chromium-based Microsoft Edge* (2020) staly *Web Components* plně podporovaným standardem napříč všemi hlavními prohlížeči.

1.6.1 Co jsou to Web Components

Web Components nejsou frameworkem ani knihovnou. Jedná se o sadu standardů, které byly přidány do *HTML*, *JavaScript* a *CSS* s cílem vytvořit nativní komponentový systém na úrovni platformy. *Web Components* zahrnují čtyři hlavní technologie:

1. **HTML Templates** – definují opakovaně použitelné *HTML* struktury, které jsou neaktivní (inertní) do doby, než jsou vloženy do *DOM*.
2. **Custom Elements** – umožňují definovat nové *HTML tagy* s vlastním chováním, které dědí z `HTMLElement` a používají „*lifecycle hooky*“ jako `connectedCallback` nebo `attributeChangedCallback`.
3. **Shadow DOM** – poskytuje komponentám zapouzdřené *DOM* stromy a stylování, což chrání jejich interní strukturu před vnějšími vlivy a naopak.
4. **CSS Custom Properties** – umožňují sdílení stylů skrz komponenty i skrz jejich shadow boundary, čímž zajišťují designové konzistence v aplikacích.

1.6.2 Výhody Web Components

Využívání *Web Components* přináší řadu praktických výhod, které reagují na běžné výzvy moderního webového vývoje. Od technologické nezávislosti přes lepší výkon až po snazší integraci do existujících projektů – právě díky těmto vlastnostem si *Web Components* postupně získávají pevné místo v architektuře webových aplikací.

1.6.2.1 Univerzální dosah

Jakožto součást oficiálních webových standardů jsou *Web Components* podporovány ve všech hlavních prohlížečích. Podle dat z prohlížeče *Chrome* se s *Web Components* setkáme

na více než 18 % všech načtených stránek, což z nich činí nejrozšířenější komponentovou technologii napříč webem. [13]

1.6.2.2 Interoperabilita a inkrementální adopce

Web Components lze využít v libovolném projektu – ať už je postavený na *React*, *Vue*, *Angularu* nebo zcela bez frameworku. Díky tomu je možné je začít adoptovat postupně, bez nutnosti přepisovat existující kód. [13]

1.6.2.3 Oddělení komponent od frameworků

Díky nativní implementaci v prohlížeči nemusí být *Web Components* navázány na žádný konkrétní framework. Komponenty tak mohou být znovu použity v různých kontextech a aplikacích bez ohledu na zvolený vývojový stack. [13]

1.6.2.4 Výkon a efektivita

Vzhledem k tomu, že jádro *Web Components* je implementováno v **C++/Rust** jako součást prohlížeče, nabízí vyšší výkon než JavaScriptové frameworky. *Shadow DOM* poskytuje efektivní aktualizace a encapsulaci, čímž zlepšuje renderovací výkon, zejména u rozsáhlých aplikací se složitým CSS. [13]

1.6.2.5 Budoucí kompatibilita a přístupnost

Web Components jsou připraveny na budoucí rozšíření *HTML* specifikací. Díky oddělení logiky od *DOM* struktury jsou přístupnější i pro neprogramátory (např. designéry nebo copywritery), kteří je mohou použít bez znalosti JavaScriptu. [13]

1.6.3 Shrnutí

Web Components představují budoucnost vývoje opakovaně použitelných *UI* prvků na webu. Spojují výhody standardizace, výkonu, modularity a interoperabilní architektury. Jejich schopnost fungovat nezávisle na frameworku z nich dělá ideální technologii pro škálovatelné systémy, které se musí přizpůsobit změnám, potřebám týmů i uživatelům. [13]

1.7 Mikro-frontendy

S rostoucí složitostí webových aplikací přestává být monolitický *front-end* udržitelný; vývojáři proto hledají způsob, jak rozdělit *UI* na menší, samostatně vyvíjené celky inspirované architekturou *microservices*. [14][15]

1.7.1 Co jsou to mikro-frontendy

Mikro-frontend představuje samostatný modul zodpovědný za konkrétní část uživatelského rozhraní. Každý modul má vlastní repozitář, životní cyklus i technologický *stack* a lze jej samostatně nasadit, aktualizovat či nahradit. Tím se podporuje paralelní vývoj, rychlejší cykly vydání a snadnější údržba. [14][15]

1.7.2 Architektura Module Federation

Module Federation je funkce balíčkovacích nástrojů (*Webpack 5*, *Vite/Rollup*, *Snowpack*), která umožňuje dynamicky načítat a sdílet JavaScriptové moduly mezi aplikacemi. Základními pojmy jsou *Host* (aplikace, která moduly konzumuje), *Remote* (aplikace, která je poskytuje) a *Bidirectional Host* (současně konzumuje i exponuje). [14]

1.7.3 Přínosy mikro-frontendů s Module Federation

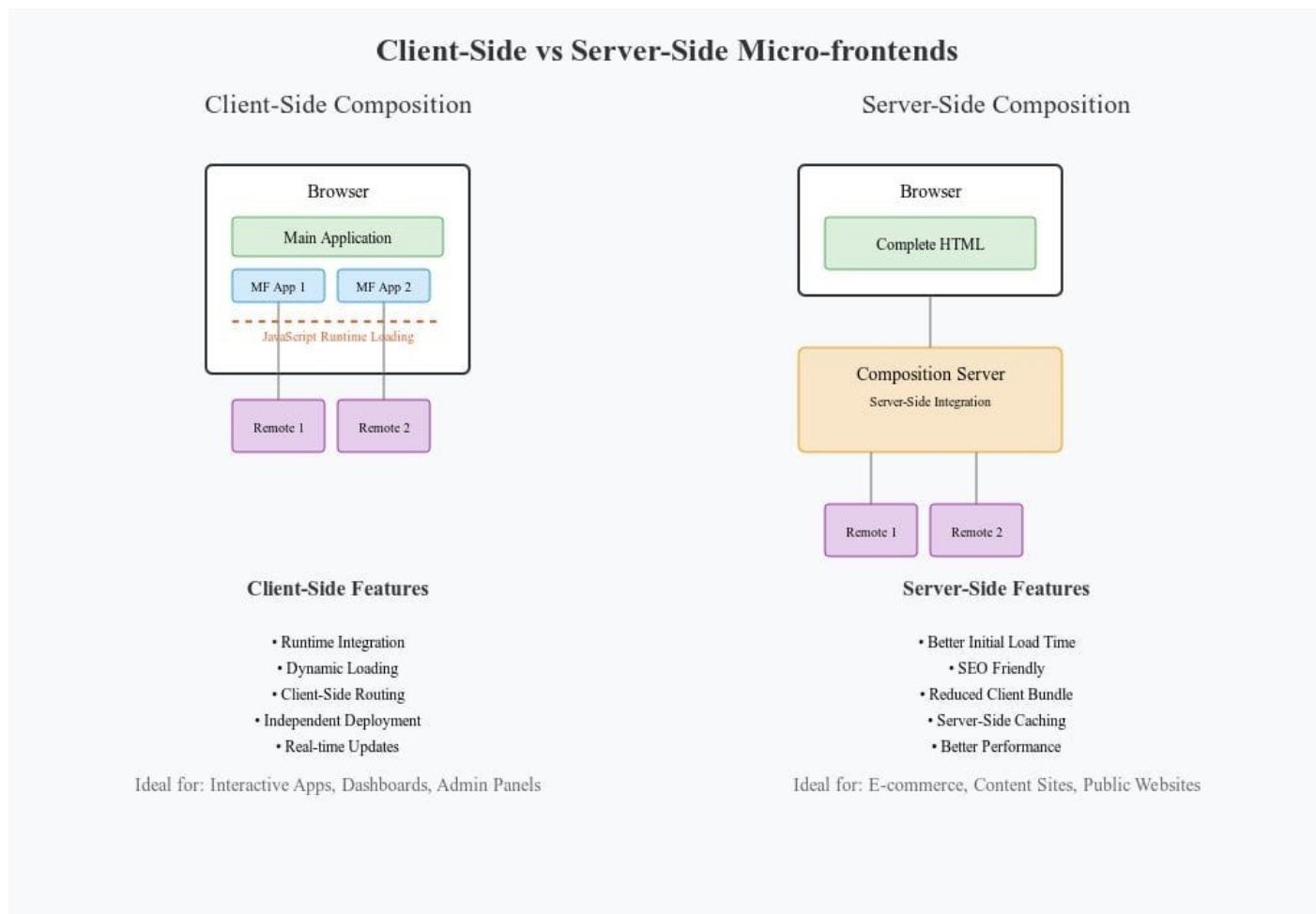
Mezi klíčové výhody patří **škálovatelnost, nezávislý vývoj a nasazování, lazy-loading a sdílení závislostí**, čímž se snižuje datová zátěž a zlepšuje výkon. Současně se podporuje týmová autonomie i opětovná použitelnost komponent napříč projekty. [14][15]

1.7.4 Typy repozitářových strategií

Architektura může využívat **monorepo** (všechny moduly v jednom repozitáři), **multirepo** (každý modul ve vlastním repozitáři) nebo **metarepo**, které kombinuje oba přístupy. **Monorepo** zjednodušuje sdílení závislostí, **multirepo** usnadňuje paralelní vývoj a **metarepo** poskytuje centralizovaný přehled nad více repozitáři. [15]

1.7.5 Client-side vs. server-side kompozice

Server-side mikro-frontendy skládají rozhraní na serveru a dodávají hotové *HTML* — ideální pro *SEO*-náročné weby. **Client-side** varianta načítá a skládá moduly v prohlížeči, což se hodí pro vysoce interaktivní aplikace. [16]



Obrázek 1: Micro-frontends CSR vs. SSR

[16]

1.8 Síťové a výkonové standardy (HTTP/3, QUIC, Brotli, CDN)

Moderní webové aplikace se neobejdou bez optimalizace síťového provozu a efektivního doručování obsahu. Právě proto hrají novodobé síťové standardy zásadní roli při zajištění rychlosti, bezpečnosti a stability. Kombinace nových protokolů, pokročilých kompresních algoritmů a distribuovaných systémů pro doručování dat umožňuje vytvářet webové služby, které jsou výkonné i ve ztížených síťových podmínkách. Následující část se zaměřuje na klíčové technologie, které posunuly webový výkon na novou úroveň.

1.8.1 Úvod do problematiky HTTP protokolů

Hypertext Transfer Protocol (HTTP) prošel od svého vzniku v roce 1996 řadou významných vylepšení. Původní verze *HTTP/1.0* byla brzy nahrazena verzí *HTTP/1.1*, která řešila problémy s cachováním, persistentními spojeními a podporou virtuálních hostitelů.

Přesto však stále existovaly zásadní limity, zejména tzv. „*Head-of-line blocking*“, kdy webový prohlížeč mohl stahovat vždy **pouze jeden soubor** přes jedno *TCP* spojení, což výrazně omezovalo rychlost načítání webových stránek. [17]

HTTP/2, který byl představen v roce 2015, měl za cíl odstranit tento problém **multiplexováním**, tedy paralelním stahováním více souborů přes jediné *TCP* spojení. Nicméně *HTTP/2* se setkal s omezením způsobeným samotným protokolem *TCP*, který stále způsoboval blokování na transportní vrstvě v případě ztráty paketů. [17]

1.8.2 HTTP/3 a QUIC

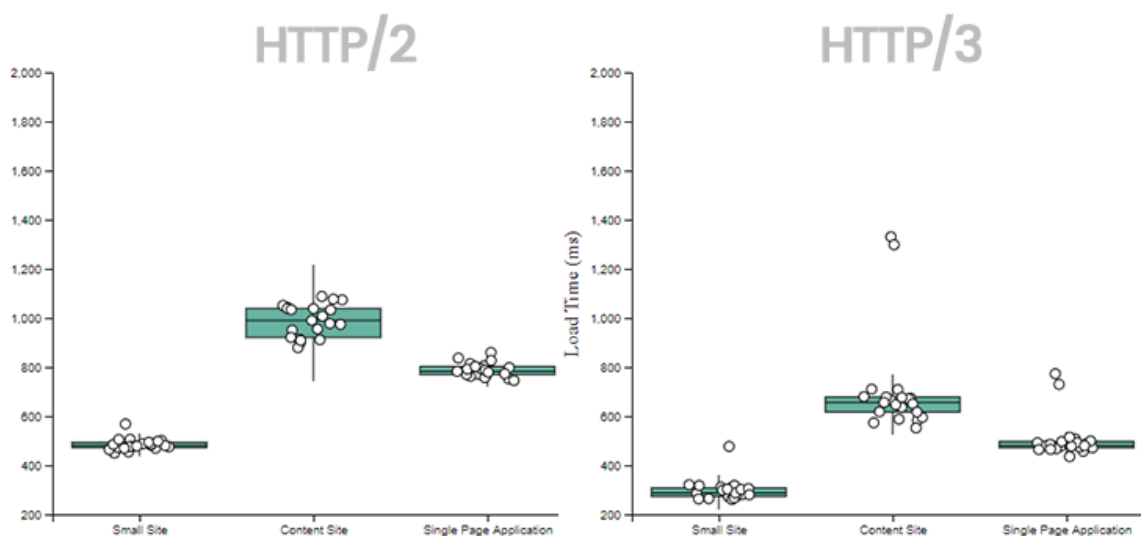
Nejnovější verze protokolu *HTTP*, **HTTP/3**, přináší radikální změnu. Nahrazuje *TCP* novým transportním protokolem **QUIC** (*Quick UDP Internet Connections*), který pracuje nad *UDP*. *QUIC* integruje multiplexování bez blokování, což znamená, že při ztrátě paketu se blokuje pouze postižený datový stream, a nikoliv celé spojení. Kromě toho *QUIC* obsahuje integrované šifrování **TLS 1.3** a umožňuje rychlejší navazování spojení díky funkci **0-RTT** (*Zero Round Trip Time*). [17][19]

Protokol *QUIC* tak řeší klíčové problémy *TCP*, jako jsou vysoká latence při opakovaném navazování spojení nebo *head-of-line blocking*, což významně zlepšuje výkon webových aplikací, zejména na velké vzdálenosti a v nestabilních sítích. [17][19]

1.8.2.1 Výsledky výkonnostních testů HTTP/3

Benchmarkové testy jasně ukazují, že *HTTP/3* nabízí výrazné zrychlení oproti předchozím verzím protokolu. V testech realizovaných mezi datovými centry v různých částech světa (například USA, Anglie) byly zaznamenány výrazné zlepšení rychlosti načítání stránek:

- V USA *HTTP/3* dosáhlo zrychlení až o **325 ms** u obsahově náročných webů. [17]
- Mezi USA a Anglií činilo zrychlení dokonce až **1200 ms**. [17]



Obrázek 2: Porovnání HTTP/2 a HTTP/3 v USA
[17]

1.8.3 Brotli – pokročilá kompresní technika pro webové aplikace

Dalším významným faktorem optimalizace výkonu webových aplikací je efektivní komprese dat. Algoritmus **Brotli**, vyvinutý společností *Google* v roce 2015, nabízí bezztrátovou kompresi s vyššími účinností než tradiční metody, jako jsou *gzip* či *deflate*. Díky tomu se snižuje objem přenášených dat, což vede k **rychlejšímu načítání webových stránek** a lepšímu uživatelskému zážitku. [18][19]

Brotli podporuje jak dynamickou kompresi v reálném čase, tak statickou předkompresi, což umožňuje optimalizaci výpočetní zátěže serverů a maximalizaci výkonu aplikací. Kompresní úrovně **Brotli** sahají od 0 do 11, přičemž nejvyšší úroveň (11) poskytuje maximální komprimaci dat, ovšem za cenu vyššího výpočetního výkonu. [18]

1.8.4 Význam Content Delivery Networks (CDN)

Technologie CDN jsou dnes klíčové pro efektivní doručování obsahu uživatelům po celém světě. *CDN* výrazně zkracují dobu načítání webových stránek tím, že **doručují obsah z geograficky nejbližšího dostupného serveru**. Integrace protokolů jako *HTTP/3* a kompresních algoritmů *Brotli* a *Zstandard* umožňuje *CDN* poskytovat ještě rychlejší, efektivnější a bezpečnější doručování obsahu. [19]

CDN jsou obzvláště **důležité** v oblastech jako **streamovací služby**, **e-commerce** nebo **online hraní**, kde i minimální zpoždění může ovlivnit uživatelský zážitek a rozhodování zákazníků.

Díky pokročilým kompresním technikám a využití protokolu *HTTP/3* je možné zajistit bezproblémové a plynulé doručování obsahu i v náročných podmínkách. [19]

1.8.4.1 Budoucí vývoj CDN technologií

Budoucnost *CDN* slibuje další inovace, například v oblasti **edge computingu** nebo umělé inteligence (*AI*). **Edge computing** posunuje výpočetní výkon blíže k uživatelům, což dále snižuje latenci a zvyšuje rychlost doručování obsahu. Integrace *AI* do *CDN* umožňuje optimalizovat strategie cachování, předvídat špičky v provozu a efektivněji zvládat bezpečnostní rizika. [19]

2 PROGRESIVNÍ WEBOVÉ APLIKACE (PWA)

Progresivní webové aplikace (*Progressive Web Apps – PWA*) představují moderní přístup k vývoji webových aplikací, který kombinuje výhody klasických webových stránek s funkcionalitou nativních mobilních aplikací. Díky využití standardních webových technologií a pokročilých *API* umožňují *PWA* vytvořit rychlé, spolehlivé a uživatelsky přívětivé aplikace, které fungují i offline, podporují **push notifikace** a mohou být instalovány přímo z prohlížeče. Tato kapitola se zaměřuje na definici, technické principy, vývoj a klíčové vlastnosti *PWA* v kontextu moderního webového vývoje.

2.1 Definice progresivní webové aplikace

Progressive Web Apps (*PWAs*) jsou aplikace vytvářené pomocí standardních webových technologií, jako jsou *HTML*, *CSS* a *JavaScript*, avšak poskytují uživatelský zážitek podobný nativním mobilním aplikacím. *PWAs* umožňují instalaci přímo z webových prohlížečů, fungují offline, zasílají push notifikace a poskytují rychlé a spolehlivé uživatelské zážitky napříč různými zařízeními a operačními systémy. [20][21][22]

PWAs kombinují **výhody tradičních webových stránek** (jednoduchá distribuce pomocí *URL*, indexace vyhledávači) s výhodami nativních aplikací (přístup k hardwarovým funkcím, offline režim, notifikace, instalace na domácí obrazovku). [20][21][22]

2.2 Technické vlastnosti PWA

Klíčovými technickými komponentami *PWA* jsou „*web app manifest*“ a „*service workers*“.

Web app manifest je soubor ve formátu *JSON*, který definuje základní informace o aplikaci, například její název, ikonu, barvy a možnosti instalace. **Service workers** jsou skripty v *JavaScriptu*, které běží na pozadí a umožňují funkcionality jako *offline* režim, cacheování dat, zachytávání síťových požadavků nebo zasílání push notifikací. [20][21][22]

Další důležitou vlastností *PWA* je **progresivní vylepšování** (*progressive enhancement*), které zajistí, že aplikace bude fungovat na širší řadě zařízení a prohlížečů, i když některé pokročilé funkce nemusí být dostupné. [20][21]

2.3 Počátky a vývoj PWA

Koncept *Progressive Web Apps* byl poprvé představen v roce 2015 inženýrem **Alexem Russellem** a designérkou **Frances Berriman**. Cílem bylo vytvořit novou kategorii aplikací, která kombinuje přístupnost webu a bohatý uživatelský zážitek nativních mobilních aplikací. [21][22]

Inspirací pro vznik *PWAs* byla potřeba překlenout propast mezi pomalými, statickými weby a rychlými, ale uzavřenými mobilními aplikacemi. Vývoj webových standardů jako jsou *service workers*, *web app manifesty* a rozšíření responsivního designu tvořily technologický základ pro vznik *PWAs*. [21][22]

2.4 Milníky vývoje PWA

Vývoj progresivních webových aplikací (*PWA*) prošel za poslední dekádu dynamickým vývojem, který výrazně ovlivnil podobu moderního webu. Od prvotního konceptu přes technickou standardizaci až po integraci do aplikačních ekosystémů – jednotlivé etapy přispěly k širšímu přijetí této technologie. Následující podkapitoly shrnují nejvýznamnější historické momenty a technologické mezníky, které formovaly současnou podobu PWA.

2.4.1 Vznik pojmu a první demonstrace

Na konferenci *Google I/O* 2015 byl oficiálně představen termín „*Progressive Web Apps*“. Brzy na to začaly vznikat první *PWA* projekty jako *Twitter Lite*, *Flipboard* nebo *Spotify*, které demonstrovaly jejich schopnosti v praxi. [21][22]

2.4.2 Standardizace a podpora napříč platformami

Podpora klíčových technologií (*service workers*, *web app manifest*) se rychle rozšířila mezi hlavní prohlížeče jako **Chrome**, **Firefox**, **Edge** a **Safari**. To umožnilo širší adopci *PWAs* jak mezi vývojáři, tak mezi firmami. [20][21][22]

2.4.3 Integrace do aplikačních obchodů

Google Play a **Microsoft Store** začaly nabízet možnost publikace *PWAs*, díky čemu se snížila bariéra vstupu a zvýšila viditelnost těchto aplikací. [21]

2.4.4 Příchod pokročilých webových API

Technologie jako **Web Assembly**, **Background Sync API**, **WebRTC** a **IndexedDB** posunuly možnosti *PWAs* na novou úroveň, umožňující složitější funkcionality a výkon na podobné úrovni nativních aplikací. [21][22]

2.5 Důležitost PWAs v moderním vývoji

Progressive Web Apps dnes představují klíčovou strategii pro firmy a vývojáře, kteří chtějí nabídnout rychlé, spolehlivé a angažované uživatelské zážitky napříč různými platformami. *PWAs* nabízejí významné výhody jako jsou nižší náklady na vývoj, větší dosah, **lepší SEO** výsledky a větší angažovanost uživatelů. Vzhledem k tomu, že se očekává další vývoj webových technologií a podporu nových funkcí v prohlížečích, lze předpokládat, že *PWAs* budou hrát ještě větší roli v budoucnosti mobilního a webového vývoje. [20][21][22]

2.6 Klíčové vlastnosti

V této podkapitole jsou představeny zásadní vlastnosti, které formují technické i uživatelské kvality progresivních webových aplikací. Jednotlivé podkapitoly se věnují jak vizuálnímu přizpůsobení a odezvě rozhraní, tak i aspektům výkonu, offline dostupnosti, možnost instalace a interaktivním funkcím.

2.6.1 Responzivita a adaptivní layout

Jedním z hlavních požadavků moderních webových aplikací je schopnost přizpůsobit se různým velikostem a typům zařízení. Tato podkapitola se zaměřuje na rozdíl mezi responsivním a adaptivním návrhem rozhraní, jejich výhodami, nároky na implementaci a využitím v praxi. Oba přístupy mají své specifické uplatnění v závislosti na typu projektu, cílové skupině i technickém zázemí.

2.6.1.1 Základní rozdíly mezi adaptivním a responsivním designem

Responsivní a adaptivní návrh představují dva přístupy k řešení zobrazení webových aplikací na různých zařízeních. **Responsivní design** je založen na flexibilních mřížkách a využívá „*CSS media queries*“ k přizpůsobení obsahu velikosti obrazovky, což zajišťuje plynulé přizpůsobení rozvržení bez ohledu na konkrétní zařízení. Naproti tomu **adaptivní**

design využívá pevně definované rozvržení pro různé šířky obrazovek (např. 320, 480, 760, 960, 1200, 1600 px), které se načítají podle rozpoznané velikosti zařízení. [23]

2.6.1.2 Vývoj a údržba

Z hlediska vývoje má **responsivní design** výhodu v nižší **náročnosti na údržbu a implementaci**, protože využívá jednotný kód pro všechny velikosti obrazovek. Naopak **adaptivní design** poskytuje **větší kontrolu nad vzhledem** na různých zařízeních, což může být výhodné například při zpětném návrhu rozsáhlých existujících systémů. [23]

2.6.1.3 Příklad z praxe

V praxi se responsivní design stal preferovanou volbou zejména pro nové projekty díky jednoduché údržbě a lepší *SEO* kompatibilitě. Například platformy jako **Slack** nebo **Dribbble** využívají responsivní design k zajištění konzistentního uživatelského zážitku napříč zařízeními. Adaptivní přístup naopak volí velké společnosti jako **Amazon** či **USA Today**, kde je důležitá konzistentní funkčnost napříč různými systémy a zpětná kompatibilita. [23]

2.6.2 Offline-first (Service workers, App Shell)

Moderní webové aplikace čelí nárokům na vysokou dostupnost a rychlou odezvu i v prostředí s omezeným připojením. Z tohoto důvodu se stále častěji uplatňuje **offline-first přístup**, který staví na **zásadních technologiích** jako jsou *Service Workers* a architektura *App Shell*. Tyto prvky umožňují vytvářet robustní aplikace s plynulým uživatelským zážitkem i v offline režimu a přinášejí nové možnosti pro optimalizaci výkonu a spolehlivosti.

2.6.2.1 Offline-first přístup

Moderní progresivní webové aplikace (*PWA*) kladou důraz na dostupnost a výkon i v podmínkách nestabilního nebo chybějícího připojení. Klíčovou roli zde hraje tzv. **offline-first přístup**, jehož základem jsou technologie **Service Workers** a architektura **App Shell**. [24][25]

2.6.2.2 *Service Workers*

Service Worker je skript v JavaScriptu, který běží nezávisle na hlavním vláknu aplikace a umožňuje přístup k událostem jako je **instalace**, **aktivace** nebo **zachytávání síťových požadavků**. Tím lze např. uložit statické zdroje do cache při prvním načtení, obsloužit požadavky z cache při ztrátě připojení nebo poskytovat okamžitou odezvu při opakovaných návštěvách. [24][25]

2.6.2.3 *Architektura App Shell*

App Shell je architektonický vzor, který odděluje statické prvky uživatelského rozhraní (např. hlavička, menu, navigace) od dynamického obsahu. Tato statická vrstva se přednačítá a ukládá do cache prostřednictvím service workeru. Díky tomu se při opětovném načtení aplikace zobrazí okamžitě známá struktura *UI*, a teprve následně se načítá obsah např. pomocí *API*. Tento přístup dramaticky zvyšuje vnímanou rychlost a zajišťuje funkčnost i v offline režimu. [24][25]

2.6.2.4 *Strategické cachování a praktické využití*

Service workers mohou navíc využívat různé strategie cachování (např. *cache-first* nebo *network-first fallback*), což umožňuje vyvážený přístup mezi aktuálností obsahu a offline dostupností. [25]

Offline-first architektura založená na *App Shell* a *service workerech* je dnes považována za standard pro vysoce výkonné, spolehlivé a uživatelsky přívětivé webové aplikace. [24][25]

2.6.3 Instalovatelnost, push notifikace, Web App Manifest

Instalovatelnost, *push notifikace* a použití *Web App Manifestu* patří mezi klíčové funkce, které posouvají progresivní webové aplikace blíže k nativním mobilním řešením. Díky těmto technologiím mohou být webové aplikace přidány na domovskou obrazovku, fungovat *offline*, zasílat notifikace nebo být distribuovány přes aplikační obchody. Jednotlivé principy a technické požadavky jsou shrnuty v následujících podkapitolách.

2.6.3.1 *Instalovatelnost*

Jednou z klíčových vlastností progresivních webových aplikací (*PWA*) je jejich schopnost být instalovatelné na zařízení uživatele. Po instalaci se *PWA* chová jako nativní aplikace –

má vlastní ikonu na domovské obrazovce, spouští se jako samostatná aplikace bez viditelného prohlížeče a lze ji ovládat přímo z operačního systému. [26]

Aby bylo možné PWA instalovat, musí splňovat několik technických požadavků:

- Webová aplikace musí být zabezpečená (*HTTPS*, *localhost* nebo *127.0.0.1*). [26]
- Musí obsahovat validní **Web App Manifest**, který definuje základní informace o aplikaci (název, ikony, výchozí *URL* adresa, způsob zobrazení apod.). [26]
- Manifest musí obsahovat minimálně položky: `name` nebo `short_name`, `start_url`, `display`, a ikony velikostí 192px a 512px. [26]

Pro distribuci *PWA* existují dvě hlavní cesty – instalace přímo z webu nebo prostřednictvím aplikačních obchodů jako **Google Play Store** či **Microsoft Store**. Pro snadnější publikaci *PWA* do obchodů lze využít nástroj **PWABuilder**. [26]

2.6.3.2 Web App Manifest

Web App Manifest je *JSON* soubor, který obsahuje metadata o *PWA*, například název aplikace, ikony, barevné schéma, výchozí stránku nebo způsob zobrazení. Tento soubor je nezbytnou podmínkou pro to, aby prohlížeč mohl nabídnout uživateli instalaci aplikace. [26]

Ukázka základní struktury souboru `manifest.json`:

```
{
  "name": "Moje PWA",
  "short_name": "PWA",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "icon.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ]
}
```

Kód 3: Ukázka JSON Web App Manifest

Manifest je referencován v hlavičce *HTML* souboru pomocí `<link rel="manifest" href="manifest.json">` [26]

2.6.3.3 Instalace z webů a aplikačních obchodů

Pokud prohlížeč zjistí, že webová aplikace splňuje kritéria pro instalovatelnost, nabídne uživateli možnost instalace. Například **Chrome** zobrazí ikonu „Instalovat“ v adresním řádku. Instalace z webu funguje na většině moderních mobilních i desktopových prohlížečů, s drobnými výjimkami jako je **Safari** na desktopu. [26]

PWA lze také publikovat v aplikačních obchodech, kde je proces specifický pro každou platformu. Přítomnost aplikace v obchodě zvyšuje její důvěryhodnost a dosah k uživatelům. [26]

2.6.3.4 Push Notifikace v PWA

Push notifikace jsou dalším důležitým prvkem *PWA*, který umožňuje udržet kontakt s uživateli i v případě, že aplikaci aktivně nepoužívají. Notifikace jsou implementovány pomocí **Service Workerů** a **Push API**. [27]

Pro aktivaci push notifikací je potřeba:

- Registrovat a nastavit **service worker**, který bude zachytávat push zprávy. [27]
- Konfigurovat *opt-in mechanismus*, kde uživatel souhlasí s přijímáním notifikací. [27]
- Vytvořit vhodnou strukturu pro manifest a zajistit kompatibilitu pro *iOS* i *Android*.

[27]

Například na platformě *iOS* je potřeba doplnit specifické položky do manifestu, aby bylo možné push notifikace používat. K tomu lze využít nástroje jako **PushAlert**, které celý proces výrazně zjednodušují. [27]

2.7 Bezpečnostní a provozní požadavky

Bezpečnost webových aplikací je klíčová pro ochranu citlivých dat uživatelů a zajištění důvěryhodnosti webu. V online prostředí se často setkáváme s útoky typu odmítnutí služby (*DoS*), neoprávněným přístupem k datům, nebo únikem informací, jako jsou hesla či platební údaje. Úspěšné útoky bývají často důsledkem důvěry webové aplikace ve vstupní data pocházející od uživatele, která nejsou řádně ověřena ani ošetřena. Proto je naprosto nezbytné aplikovat strategii vícevrstvé ochrany zahrnující server, klienta i samotnou logiku aplikace.

[28]

2.7.1 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) je základní bezpečnostní opatření, které zajišťuje šifrování dat přenášených mezi webovým serverem a prohlížečem uživatele. Chrání tak před odposlechem a úpravami komunikace během přenosu, což je zvláště důležité u citlivých údajů, jako jsou přihlašovací údaje nebo platební informace. Použití *HTTPS* by mělo být standardem pro všechny weby bez ohledu na jejich obsah. Implementace *HTTPS* také přináší další výhody, jako je lepší hodnocení ve výsledcích vyhledávání *Google*. [29]

2.7.1.1 Vyvarování se smíšenému obsahu (*mixed content*)

Přestože stránka používá *HTTPS*, její bezpečnost může být ohrožena, pokud načítá některé zdroje (například *JavaScript*, obrázky nebo *CSS*) přes nezabezpečené *HTTP*. Tento tzv. smíšený obsah může být zneužit útočníkem k injektáži škodlivého kódu. Moderní prohlížeče tyto pokusy blokují a uživatelům zobrazují varování. Aby se předešlo těmto problémům, je nutné zajistit, aby všechny zdroje byly načítány přes protokol *HTTPS*. [29]

2.7.1.2 Bezpečné formuláře a přihlašování

Přihlašovací formuláře by měly být chráněny nejen tím, že běží na stránce využívající *HTTPS*, ale také tím, že jejich odeslání (atribut `action`) musí směřovat na zabezpečenou adresu (`https://`) a používat metodu `POST`. Tím se zabrání úniku citlivých dat, například v historii prohlížeče nebo v hlavičkách `Referer`. Správná implementace těchto opatření významně snižuje riziko zneužití hesel. [29]

2.7.2 Content Security Policy (CSP)

Content Security Policy (CSP) je mechanismus, který umožňuje specifikovat, odkud může webová stránka načítat skripty, styly a další zdroje, čímž výrazně zvyšuje ochranu proti útokům typu **Cross-Site Scripting (XSS)**. Zavedením přísné **CSP** lze omezit provádění nepovoleného *JavaScriptu*, zakázat inline skripty, omezit použití rizikových *API* jako `eval()`, a dokonce zamezit načítání neautorizovaných zdrojů. [30]

2.7.2.1 Praktická implementace CSP

Pro efektivní implementaci **CSP** se doporučuje využívat tzv. **nonce** nebo **hash hodnoty**, které zajišťují, že pouze explicitně povolené skripty mohou být spuštěny. Než je *CSP* nasazena „naostro“, je vhodné nejprve testovat pomocí hlavičky `Content-Security-Policy-Report-Only`, která slouží k monitorování porušení bez blokování zdrojů. Kromě

ochrany před XSS pomáhá CSP i proti **clickjackingu** a umožňuje vynucení *HTTPS* připojení pomocí direktivy `upgrade-insecure-requests`. [30]

2.7.3 Cross-Origin Resource Sharing (CORS)

Mechanismus Cross-Origin Resource Sharing (CORS) slouží k bezpečnému sdílení zdrojů mezi různými doménami. Vzhledem k tomu, že moderní webové aplikace často integrují externí služby – například načítají fonty z veřejných knihoven, získávají data z *API* nebo zobrazují multimedia z jiných serverů – je potřeba tento přístup kontrolovat a autorizovat. Standardní politika stejného původu (*same-origin policy*) tyto požadavky omezuje, aby zabránila útokům typu **Cross-Site Request Forgery (CSRF)**. *CORS* tento model rozšiřuje tím, že server může výslovně povolit přístup z jiných domén pomocí *HTTP* hlaviček jako `Access-Control-Allow-Origin`. [32]

2.7.3.1 Jak CORS funguje

Při cross-origin požadavku prohlížeč přidává do hlaviček požadavku informaci o původu (`Origin`) a očekává, že server odpoví s hlavičkou `Access-Control-Allow-Origin`, která povoluje či zamítá přístup. Pokud je požadavek **povoleno**, prohlížeč **předá data zpět skriptům** na stránce. Pokud není, prohlížeč přístup **zablokuje**. *CORS* tak vytváří řízený mechanismus přístupu k datům, který je bezpečnější než historické přístupy jako *JSONP*, které byly méně kontrolovatelné a zranitelnější. [32]

2.7.3.2 Preflight požadavky a hlavičky

Některé složitější požadavky (např. s metodou *DELETE* nebo s vlastními hlavičkami) vyžadují tzv. **preflight požadavek** pomocí metody *OPTIONS*. Prohlížeč tím ověřuje, zda server akceptuje plánovaný požadavek. Server musí odpovědět sadou *CORS* hlaviček – `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers` – a může navíc určit délku platnosti odpovědi pomocí `Access-Control-Max-Age`. [32]

2.7.3.3 CORS a přihlašovací údaje

Pro přenos **cookies** nebo *HTTP* autentifikace je třeba, aby požadavek obsahoval atribut `credentials: include` a server odpověděl hlavičkou `Access-Control-Allow-Credentials: true`. V tomto případě nelze použít hvězdičku `*` jako hodnotu hlavičky

`Access-Control-Allow-Origin` – musí být uveden konkrétní povolený původ. Jinak prohlížeč odpověď z bezpečnostních důvodů ignoruje. [32]

2.7.4 Závěrečné poznámky k ochraně webových aplikací

Bezpečnostní a provozní opatření, jako jsou **HTTPS**, **Content Security Policy (CSP)** a **Cross-Origin Resource Sharing (CORS)**, představují základní pilíře ochrany moderních webových aplikací. Jejich správná implementace zajišťuje šifrovaný přenos dat, zabraňuje spouštění neautorizovaných skriptů a umožňuje bezpečné sdílení zdrojů mezi různými doménami. V kombinaci s důsledným ověřováním a sanitizací vstupních dat tvoří robustní obranný mechanismus proti nejčastějším útokům v prostředí webu.

3 MOŽNOSTI VÝVOJE MOBILNÍCH APLIKACÍ

V této kapitole jsou shrnuty a vzájemně porovnány nejvýznamnější přístupy k vývoji mobilních aplikací, jak jsou popsány v odborné literatuře a aktuálních technických zdrojích. Důraz je kladen na rozdíly mezi nativním vývojem, multiplatformními frameworky a hybridními řešeními. Porovnání jednotlivých přístupů vychází z publikovaných dat a zaměřuje se na klíčové aspekty, jako jsou výkon, náklady na vývoj, přístup k hardwarovým funkcím, bezpečnost či způsob správy a publikace aktualizací. Cílem je poskytnout přehled o současných možnostech vývoje mobilních aplikací a usnadnit tak orientaci v této problematice.

3.1 Nativní přístup (Android Kotlin/Java, iOS Swift)

Vývoj mobilních aplikací nativním způsobem představuje jednu z nejefektivnějších cest, jak vytvářet výkonné, rychlé a stabilní mobilní aplikace. V rámci tohoto přístupu jsou aplikace vyvíjeny přímo pro konkrétní operační systém – tedy **Android** nebo **iOS** – pomocí oficiálně podporovaných jazyků, nástrojů a knihoven. [33][34][35]

3.1.1 Vývoj pro Android: Kotlin a Java

Java byla dlouhá léta hlavním jazykem pro vývoj aplikací pro **Android**. Jedná se o objektově orientovaný jazyk, který běží na **Java Virtual Machine (JVM)** a nabízí širokou škálu knihoven, komunitní podporu a stabilitu. **Kotlin**, který je rovněž jazykem běžícím na **JVM**, byl v roce 2017 vyhlášen společností *Google* jako **oficiální jazyk pro Android vývoj**. *Kotlin* je modernější, konciznější a bezpečnější než *Java*, přičemž zůstává plně interoperabilní s *Java* kódem. [33]

3.1.1.1 Syntaxe a jazykové vlastnosti

Java je známá svou „upovídaností“ a nedostatkem moderních jazykových prvků, jako je typová inference nebo bezpečnost vůči *null* hodnotám. **Kotlin** naproti tomu přináší funkce jako chytrý casting, datové třídy, bezpečnost vůči *null* hodnotám (*null safety*), korutiny (*coroutines*) pro asynchronní operace nebo jednodušší deklaraci proměnných. Díky těmto vlastnostem lze v jazyce *Kotlin* napsat výrazně méně kódu než v Javě při dosažení stejného výsledku. [33][36]

3.1.1.2 Výkon a interoperabilita

Z hlediska výkonu **kompilují oba jazyky do stejného bytecode pro JVM**, takže rozdíl v rychlosti běhu aplikací je **minimální**. Výhodou Kotlinu je ale efektivnější práce s pamětí a modernější konstrukce jazyka. *Kotlin* je také plně interoperabilní s knihovnami napsanými v Javě, což usnadňuje přechod a integraci do stávajících projektů. [33][36]

3.1.2 Vývoj pro iOS: Swift a Objective-C

Objective-C, jako nadmnožina **jazyka C**, byl hlavním jazykem pro *iOS* vývoj po mnoho let. Je silně objektově orientovaný, avšak jeho syntaktická struktura je komplikovaná a méně přehledná pro nové vývojáře. **Swift**, představený společností *Apple* v roce 2014, byl navržen jako moderní alternativa s důrazem na jednoduchost, bezpečnost a výkon. [34][35]

3.1.2.1 Syntaxe a klíčové prvky

Swift nabízí čistou a stručnou syntaxi, která připomíná přirozený jazyk. Zavádí pokročilé prvky jako *optionals* (volitelné typy), typovou inferenci, výjimekové zpracování nebo funkcionální konstrukce jako *map*, *filter*, *reduce*. *Objective-C* naopak pracuje s oddělenou implementací a deklarací tříd a používá složitější zápis. [34][35]

3.1.2.2 Správa paměti a výkon

Objective-C vyžaduje ruční správu paměti nebo používání automatického referenčního počítání (*ARC*) omezeného na *Cocoa API*. **Swift** využívá *ARC* komplexněji a bezpečněji, což vede ke snížení paměťových úniků a zjednodušení vývoje. Aplikace psané ve **Swiftu** mají navíc podle benchmarků lepší výkon, zejména při zpracování náročných výpočtů. [34][35]

3.1.3 Shrnutí

Na základě výše uvedených poznatků lze konstatovat, že nativní vývoj představuje robustní a osvědčený přístup, který umožňuje maximální využití potenciálu dané platformy. **Kotlin** a **Swift** se ukazují jako moderní a efektivní jazyky s důrazem na bezpečnost, přehlednost kódu a snadnější údržbu aplikací. Díky své syntaktické úspornosti a pokročilým funkcím nahrazují své starší protějšky, **Java** a **Objective-C**, které však nadále nacházejí uplatnění v některých projektech. Využití nativních nástrojů umožňuje vývojářům vytvářet aplikace s

vysokým výkonem, kvalitní správou paměti a hlubokou integrací se systémem, což je klíčové především u složitějších nebo citlivějších aplikací.

3.2 Multiplatformní kompilované frameworky (Flutter, React Native, Lynx)

Multiplatformní frameworky přináší moderní přístup k vývoji mobilních aplikací, který umožňuje psaní **sdíleného kódu pro více platforem** (např. *Android* a *iOS*) **z jednoho kódového základu**. Tento přístup přináší úsporu času, nákladů a zajišťuje konzistentní uživatelský zážitek napříč různými zařízeními. [38]

3.2.1 Flutter

Flutter je **open-source framework** vyvinutý společností *Google*, který používá programovací jazyk **Dart**. *Flutter* umožňuje vývoj aplikací pro *iOS*, *Android*, *web*, *Windows*, *macOS* i *Linux* z jednoho společného kódu. Na rozdíl od jiných frameworků vykresluje *UI* pomocí vlastního enginu (*Skia*), což zajišťuje konzistentní výstup napříč platformami. [37]

Velkou výhodou **Flutteru** je tzv. „*hot reload*“, který umožňuje okamžitou vizualizaci změn v kódu bez restartování aplikace. Komponenty v aplikaci jsou tvořeny pomocí widgetů, které lze volně kombinovat a upravovat. Vývojáři tak mají vysokou flexibilitu při návrhu *UI*. [37]

3.2.2 React Native

React Native je framework vyvinutý společností *Meta*, který umožňuje vytvoření mobilních aplikací pro *Android* a *iOS* pomocí jazyka **JavaScript** a knihovny **React**. Na rozdíl od hybridních frameworků typu *Cordova* používá nativní komponenty a vykreslovací engine, což zajišťuje lepší výkon a „nativní“ vzhled aplikace. [38]

React Native si získal popularitu díky možnosti znovupoužití velké části kódu a schopnosti vytvářet aplikace pro více platforem bez nutnosti učit se specifický jazyk jako **Swift** či **Kotlin**. Systém komunikace mezi JavaScriptovou a nativní vrstvou zajišťuje tzv. **bridge**. [38]

Mezi známé aplikace postavené na **React Native** patří např. *Facebook*, *Instagram*, *Skype* nebo *Uber Eats*. Výhodou je i široká komunita a dostupnost mnoha knihoven a komponent. [38]

3.2.3 Lynx

Lynx je nový multiplatformní framework vyvinutý společností *ByteDance* (*TikTok*), který podporuje vývoj aplikací pro Android, *iOS* a *web*. Je napsaný v jazyce *C++* a využívá vlastní *JavaScript engine* **PrimJS** založený na **QuickJS**. Cílem jazyka *Lynx* je dosáhnout vyššího výkonu pomocí **dvouvrstvé architektury**, kde je oddělena *UI* logika a úkoly na pozadí. [39]

Lynx podporuje **CSS stylování**, včetně animací, proměných a maskování, což výborně zpřístupňuje webovým vývojářům přechod do mobilního světa. Podporuje *React* i další *JavaScript* frameworky (např. *Vue*, *Svelte*), což z něj dělá velmi flexibilní platformu. [39]

Vzhledem k tomu, že **Lynx** je nový a jeho ekosystém je ještě v rané fázi, nenabízí zatím tolik knihoven jako zavedené frameworky. *ByteDance* však již *Lynx* používá ve svých produktech jako *TikTok Shop* nebo *LIVE*. [39]

3.2.4 Shrnutí

Dle dostupných odborných zdrojů představují multiplatformní frameworky moderní a efektivní přístup k vývoji mobilních aplikací. Přinášejí výhody zejména v oblasti sdíleného kódu, zrychlení vývoje a snížení nákladů. **Flutter** vyniká svou flexibilitou při návrhu uživatelského rozhraní a konzistencí napříč platformami díky vlastnímu vykreslovacímu enginu. **React Native** je osvědčenou technologií s rozsáhlou komunitní podporou a dobrou integrací s nativními komponentami, což ho činí vhodným pro rozsáhlejší produkční projekty. **Lynx** se profiluje jako výkonná alternativa s moderní architekturou a silným důrazem na rozšiřitelnost, i když jeho ekosystém je zatím ve fázi růstu.

Celkově lze konstatovat, že každý z uvedených frameworků má své specifické výhody a je vhodný pro jiné typy projektů v závislosti na prioritách zadavatele i vývojového týmu.

3.3 Hybridní řešení (Ionic, Cordova)

Hybridní vývoj mobilních aplikací umožňuje využití webových technologií (HTML, CSS, JavaScript) pro tvorbu aplikací, které běží na více platformách a zároveň mají přístup k

nativním funkcím zařízení. Dvěma nejznámějšími zástupci tohoto přístupu jsou **Apache Cordova** a **Ionic Framework**.

3.3.1 Apache Cordova

Apache Cordova je **open-source framework**, který umožňuje vývoj mobilních aplikací pomocí **webových technologií**. Aplikace se balí do nativní *WebView* komponenty, která zobrazuje *HTML/CSS/JS* rozhraní a komunikuje s nativními *API* prostřednictvím tzv. Cordova pluginů. [40]

Tento přístup umožňuje vývoj jediné aplikace, která běží na **Androidu**, **iOS** i dalších platformách. Vývojářům stačí znalost webových technologií, což výrazně snižuje nároky na učení nových jazyků jako *Swift* nebo *Java*. [40]

3.3.2 Ionic

Ionic je framework postavený nad **Cordovou**, který umožňuje vývoj mobilních aplikací s využitím frameworků *Angular*, *React* nebo *Vue*, *HTML* a *CSS*. Výsledná aplikace běží stejně jako u **Apache Cordova** ve *WebView*, ale přidává moderní *UI* komponenty a vývojářské nástroje. [41][42]

Ionic umožňuje vývoj aplikací pro **Android**, **iOS**, **web** a další platformy z jednoho kódu. Integruje knihovny komponent, které vypadají jako nativní prvky, a nabízí širokou škálu pluginů pro přístup k funkcím zařízení. [41][42]

3.4 Komparativní analýza

S rostoucí nabídkou nástrojů pro vývoj mobilních aplikací je nezbytné porovnat jednotlivé přístupy z hlediska jejich praktických vlastností. Tato část nabízí souhrnné srovnání klíčových aspektů – od výkonu přes náklady až po bezpečnost.

3.4.1 Výkon a uživatelský zážitek

Jedním z hlavních kritérií při výběru technologie pro vývoj aplikace je její schopnost zajistit rychlou odezvu, plynulost a komfortní používání. Výkon úzce souvisí nejen s technickým řešením na úrovni kódu, ale také s celkovou architekturou aplikace a způsobem, jakým využívá zdroje zařízení.

3.4.1.1 *Nativní aplikace*

Nativní aplikace jsou známé **vysokou výkonností**, protože jsou optimalizovány přímo pro konkrétní platformu. Poskytují plynulou a rychlou odezvu, hladké animace a přirozený uživatelský zážitek díky přímé integraci s hardwarem zařízení a systémovými *API*. Tyto aplikace se vyznačují i robustní *offline* funkcionalitou. [45][46]

3.4.1.2 *Cross-platformní aplikace*

Cross-platformní aplikace se snaží poskytnout výkon blízký nativním aplikacím díky **kompilaci kódu** přímo do nativního prostředí. Přesto mohou mít omezení při práci s náročnější grafikou či pokročilými animacemi. Uživatelský zážitek bývá konzistentnější než u hybridních aplikací, ale stále nemusí dosahovat úplné nativní úrovně. [45][46]

3.4.1.3 *Hybridní aplikace*

Hybridní aplikace mají **výkon** mírně **nižší** kvůli použití *WebView* komponenty. Uživatelský zážitek může být méně plynulý, zejména při náročných grafických operacích nebo složitých animacích, což může vést k pomalejšímu načítání či nižší responzivnosti. [45][46]

3.4.2 **Náklady na vývoj a údržbu**

Finanční a časové náklady hrají zásadní roli při rozhodování o technologii pro vývoj mobilní aplikace. Každý přístup – **nativní, cross-platformní nebo hybridní** – s sebou nese rozdílné požadavky na rozpočet, lidské zdroje a dlouhodobou správu. V následujícím srovnání jsou shrnuty hlavní výhody a nevýhody jednotlivých řešení z hlediska jejich ekonomické náročnosti, jak při prvotním vývoji, tak při následné údržbě a aktualizacích.

3.4.2.1 *Nativní aplikace*

Nativní aplikace mají **nejvyšší náklady**, protože **vyžadují vývoj a údržbu samostatných kódových bází** pro každou platformu zvlášť (*iOS* a *Android*). Vývoj je tedy delší, náročnější na lidské zdroje a finanční prostředky. Počáteční náklady se typicky pohybují od **50 000** do více než **250 000** dolarů. [43][44]

3.4.2.2 *Cross-platformní aplikace*

Cross-platformní řešení mají nižší náklady než nativní vývoj. Díky jednotné kódové bázi jsou rychlejší a levnější jak při vývoji, tak při následné údržbě a aktualizacích. Typicky umožňují úspory nákladů o **30–50 %** oproti nativnímu vývoji. [44][46]

3.4.2.3 *Hybridní aplikace*

Hybridní aplikace mají podobně jako cross-platformí přístup výrazně nižší počáteční náklady, obvykle mezi **20 000** a **100 000** dolarů, protože umožňují vývoj jedné kódové báze pro více platforem. To vede k rychlejšímu vývoji, sníženým nákladům a jednodušší údržbě v dlouhodobém horizontu. Na druhou stranu mohou být **potřeba častější aktualizace z důvodu kompatibility** s novými verzemi zařízení a platforem. [43][45]

3.4.3 **Přístup k hardwaru, publikace, aktualizace**

Důležitým aspektem při výběru vývojového přístupu je možnost využití funkcí zařízení a celková správa aplikace po jejím uvedení. Rozdíly mezi jednotlivými přístupy se promítají do toho, jak snadno lze aplikaci publikovat, jak rychle reaguje na změny či aktualizace, a jak hluboký přístup mají vývojáři k nativním funkcím operačního systému.

3.4.3.1 *Nativní aplikace*

Nativní aplikace poskytují nejlepší přístup k hardwarovým funkcím zařízení, včetně GPS, kamery, senzorů, biometrických funkcí a dalšího specifického hardwaru. Publikace a aktualizace aplikací však může být náročnější kvůli potřebě oddělené správy pro každou platformu, což může prodlužovat čas uvedení nových funkcí. [43][46]

3.4.3.2 *Cross-platformní aplikace*

Cross-platformní aplikace disponují **velmi dobrým přístupem** k hardwarovým funkcím zařízení, který je **blízký nativním aplikacím**. Díky přímé kompilaci do nativního kódu mohou tyto aplikace efektivně využívat platformová *API*, což umožňuje kvalitní integraci s hardwarem zařízení. Publikace a aktualizace cross-platformních aplikací jsou jednodušší než v případě nativních aplikací, jelikož je možné jedním krokem distribuovat aktualizace napříč všemi podporovanými platformami. To výrazně zrychluje proces uvádění nových verzí na trh. [45][46]

3.4.3.3 *Hybridní aplikace*

Hybridní aplikace mají omezenější přístup k hardwarovým funkcím zařízení ve srovnání s nativními i cross-platformními aplikacemi. Přístup ke klíčovým funkcím (kamera, GPS, kontakty apod.) probíhá pomocí speciálních pluginů, což může znamenat omezení v pokročilejších nebo specifických případech použití. Výhodou hybridních aplikací je snadná publikace a jednoduché aktualizace díky jediné webové kódové bázi, což umožňuje rychlou distribuci změn napříč všemi platformami. [45][46]

3.4.4 **Bezpečnost a správa dat**

Zajištění bezpečnosti a efektivní správa uživatelských dat představují zásadní faktory zejména v aplikacích pracujících s citlivými informacemi. Zvolený vývojový přístup ovlivňuje míru ochrany dat, možnosti šifrování, práci s autentizací i odolnost vůči zranitelnostem. Každá technologie přináší jinou úroveň zabezpečení i specifické výzvy.

3.4.4.1 *Nativní aplikace*

Nativní aplikace nabízejí nejvyšší úroveň bezpečnosti díky možnosti využití specifických bezpečnostních funkcí platformy, například biometrické autentizace nebo pokročilého šifrování dat. To je důležité zejména pro aplikace, které manipulují s citlivými uživatelskými údaji nebo finančními transakcemi. [45][46]

3.4.4.2 *Cross-platformní aplikace*

Cross-platformní aplikace nabízejí solidní bezpečnost, ale závisí na kvalitě použitého frameworku a třetích stran. Jelikož jsou tyto aplikace kompilovány přímo do nativního prostředí, umožňují efektivní implementaci bezpečnostních opatření, nicméně případné zranitelnosti v frameworku či závislostech mohou představovat bezpečnostní riziko. [45][46]

3.4.4.3 *Hybridní aplikace*

Hybridní aplikace mohou být náchylnější k bezpečnostním problémům kvůli použití **webových technologií**, které mají inherentní bezpečnostní slabiny. Zabezpečení hybridních aplikací tak vyžaduje pečlivější správu bezpečnostních opatření jak z webového, tak z mobilního hlediska. [45][46]

3.4.5 Závěr komparativní analýzy

Na základě prostudovaných zdrojů bylo provedeno srovnání tří hlavních přístupů k vývoji mobilních aplikací – **nativního, cross-platformního a hybridního**. Toto porovnání vychází z dostupné odborné literatury a analytických článků, přičemž zohledňuje faktory jako výkon, náklady, přístup k hardwaru, bezpečnost a správu dat.

Z přehledu vyplývá, že **nativní přístup** nabízí nejvyšší úroveň výkonu, zabezpečení a integrace s funkcemi zařízení, a je tak vhodný pro náročnější projekty. **Cross-platformní** řešení představují kompromis mezi kvalitou a efektivitou vývoje, s možností snížení nákladů a rychlejší implementace. **Hybridní přístup** pak umožňuje nejrychlejší uvedení na trh a nejnižší vstupní náklady, avšak s určitými limity z hlediska výkonu a dostupnosti funkcí. Výběr konkrétní technologie by proto měl být vždy podmíněn specifiky projektu, jeho rozpočtem, cílovou skupinou i dlouhodobými požadavky na údržbu a rozvoj.

4 UI/UX DESIGN

Při vývoji moderních aplikací hrají klíčovou roli dvě úzce propojené disciplíny: uživatelské rozhraní (*UI*) a uživatelský zážitek (*UX*). *UI* se zaměřuje na vizuální a interakční stránku rozhraní – tedy to, co uživatel vidí a jak s aplikací přímo komunikuje. *UX* se naproti tomu soustředí na celkový dojem z používání produktu, včetně jeho použitelnosti, přístupnosti a emočního dopadu. Společným cílem obou přístupů je vytvořit řešení, které je nejen funkční, ale zároveň intuitivní, esteticky přitažlivé a přizpůsobené reálným potřebám uživatelů. Kombinace kvalitního *UI* a promyšleného *UX* významně zvyšuje úspěšnost digitálních produktů a podporuje dlouhodobou spokojenost uživatelů.

4.1 Uživatelské rozhraní (UI)

Kvalita uživatelského rozhraní (*UI*) přímo ovlivňuje použitelnost, užitečnost a atraktivitu aplikací. Dobré uživatelské rozhraní by mělo splňovat soubor osvědčených pravidel a principů, mezi nejznámější patří deset heuristik od *Jakoba Nielsena* a *Rolfa Molicha*. Tyto principy slouží jako základní vodítko při navrhování a vyhodnocování *UI*, a jejich cílem je vytvářet intuitivní, efektivní a přívětivé rozhraní. [47][48]

4.1.1 Viditelnost stavu systému

Základním požadavkem na kvalitní uživatelské rozhraní je, aby uživatelé vždy jasně věděli, co se v systému děje. Systém by měl poskytovat včasnou a zřetelnou zpětnou vazbu na provedené akce. Uživatelé tak mají pocit kontroly nad děním v aplikaci, což buduje důvěru a snižuje nejistotu. [47][48]

4.1.2 Systém odpovídající reálnému světu

Návrh rozhraní by měl mluvit jazykem uživatelů, tedy využívat jim známé výrazy, pojmy a konvence z reálného světa. Informace by měla být uspořádána logicky a přirozeně podle očekávání uživatelů. Správné využití známých symbolů a terminologie výrazně snižuje kognitivní zátěž a usnadňuje pochopení fungování aplikace. [47][48]

4.1.3 Uživatelská kontrola a svoboda

Uživatelé často provádějí nechtěné akce, proto musí mít možnost snadno se vrátit zpět nebo akci zrušit. Přítomnost jasně označených možností „zpět“, „zrušit“ nebo „obnovit“ poskytuje

pocit kontroly a svobody, čímž snižuje frustraci a zvyšuje celkovou spokojenost s aplikací. [47][48]

4.1.4 Konzistence a standardy

Rozhraní by mělo být konzistentní jak interně v rámci jedné aplikace, tak externě vůči jiným běžně používaným aplikacím a platformám. Uživatelé očekávají, že podobné prvky a akce budou fungovat stejně napříč různými systémy. Dodržování konzistence snižuje kognitivní zátěž a podporuje intuitivní ovládání nového rozhraní. [47][48]

4.1.5 Prevence chyb

Ideální návrh aplikace minimalizuje riziko vzniku chyb. Důležité je eliminovat příležitosti k chybám předem, případně varovat uživatele a požádat ho o potvrzení, než provede kritickou akci. Prevence chyb je účinnější než jejich následná korekce, neboť přispívá k úspoře času a zároveň minimalizuje riziko frustrace. [47][48]

4.1.6 Rozpoznání místo vybavování

Rozhraní by mělo minimalizovat potřebu uživatelů uchovávat si informace v paměti. Všechny důležité prvky, volby a akce by měly být viditelné a snadno rozpoznatelné. Člověk je schopen snáze rozpoznat známou položku než si ji zpětně vybavit z paměti. Z tohoto důvodu je vhodné využívat vizuální nápovědu a jednoznačné označení funkcí a možností. [47][48]

4.1.7 Flexibilita a efektivita použití

Dobré rozhraní poskytuje různé možnosti provedení stejné akce, aby vyhovělo potřebám různě zkušených uživatelů. Zatímco novým uživatelům vystačí základní interakce, pokročilí uživatelé ocení zkratky, gesta nebo jiné rychlé způsoby ovládání. Personalizace a možnost upravit si rozhraní podle vlastních potřeb zvyšuje efektivitu a uživatelskou spokojenost. [47][48]

4.1.8 Estetický a minimalistický design

Rozhraní by mělo obsahovat pouze informace a prvky, které jsou nezbytně nutné pro splnění uživatelských cílů. Přebytečné informace „soutěží“ o pozornost a komplikují ovládání.

Minimalistický design umožňuje rychleji se orientovat v obsahu a zaměřit se na důležité informace. [47][48]

4.1.9 Pomoc uživatelům rozpoznat, diagnostikovat a řešit chyby

Pokud již k chybě dojde, systém by měl uživateli jasně sdělit, v čem problém spočívá, jaké byly příčiny a jak problém vyřešit. Chybová hlášení by měla být formulována srozumitelně, bez technického žargonu a měla by obsahovat konkrétní doporučení k nápravě. [47][48]

4.1.10 Náповěda a dokumentace

Ideální uživatelské rozhraní by nemělo potřebovat rozsáhlou dokumentaci. Přesto však existují situace, v nichž je vhodné nabídnout uživateli snadno dostupnou nápovědu, stručnou dokumentaci nebo interaktivního průvodce. Taková dokumentace by měla být snadno dohledatelná, relevantní k aktuálnímu úkolu a měla by jasně a jednoduše uživatele provést řešením problému. [47][48]

4.2 Uživatelský zážitek (UX)

Při návrhu moderních webových a mobilních aplikací již nestačí zaměřit se pouze na funkčnost – stále větší důraz je kladen na samotný zážitek uživatele při interakci s produktem. Právě **uživatelský zážitek (UX)** představuje komplexní souhrn dojmů, které uživatel vnímá během používání aplikace, a stává se zásadním faktorem ovlivňujícím úspěch projektu. Následující podkapitoly se věnují vybraným principům *UX designu*, které mají významný dopad na kvalitu, použitelnost a celkovou přívětivost digitálních produktů.

4.2.1 Náповěda a dokumentace

UX design (*User Experience design*) je proces, který návrháři využívají k vytváření kvalitnějších, smysluplnějších a atraktivnějších zážitků pro uživatele produktů, služeb či interaktivních systémů. Cílem *UX* je maximalizovat snadnost použití, užitečnost a celkový uživatelský dojem, což vede ke zvýšenému zájmu, vyšší loajalitě ke značce a zlepšené konverzi. *UX design* je dynamický obor, který se průběžně přizpůsobuje technologickým změnám a potřebám uživatelů. [49][50]

4.2.2 Zaměření uživatele

Nejdůležitějším principem *UX designu* je orientace na uživatele. Tento přístup začíná důkladným pochopením **potřeb, cílů a frustrací** uživatelů prostřednictvím výzkumu a analýzy cílových skupin. *UX* návrháři by měli vycházet z reálných potřeb uživatelů, nikoliv z vlastních domněnek. Uživatelé by měli být zapojeni v každé fázi designového procesu, což pomáhá včas identifikovat problémy použitelnosti a snižuje celkové náklady na vývoj. [49][51]

4.2.3 Použitelnost a přístupnost

Klíčovým aspektem *UX designu* je použitelnost, tedy schopnost uživatelů jednoduše, rychle a efektivně dosáhnout svých cílů. Produkty by měly být nejen použitelné, ale také užitečné, tedy pomáhat **efektivně splnit požadované úkoly**. Zároveň by měly být produkty pravidelně využívány. *UX design* také klade velký důraz na přístupnost. To znamená, že by produkty měly být dostupné i pro uživatele s různými druhy postižení, včetně fyzických, vizuálních, sluchových či kognitivních omezení. Dodržování standardů přístupnosti, jako jsou **Web Content Accessibility Guidelines (WCAG)**, zajišťuje, že produkty mohou používat všichni uživatelé bez rozdílu. [49][50][51]

4.2.4 Konzistence a hierarchie

Konzistentní *UX design* usnadňuje orientaci a snižuje jejich kognitivní zátěž. Používání známých prvků a standardních interakcí pomáhá uživatelům snadno se orientovat v produktu a rychle jej ovládat. Konzistence nejenže zvyšuje použitelnost, ale také posiluje důvěryhodnost a profesionální vzhled produktu. Podobně důležitá je jasná vizuální hierarchie, která vede oko od nejdůležitějších prvků k těm méně důležitým. Dobře navržená vizuální hierarchie usnadňuje orientaci v obsahu a snižuje riziko frustrace. [49][50][51]

4.2.5 Kontrola a svoboda uživatele

Uživatelé by měli mít možnost kontrolovat interakce s produktem. *UX design* by měl nabízet možnosti, jak jednoduše napravit chyby nebo změnit rozhodnutí. To zahrnuje například funkce „zpět“, „zrušit“ nebo „vrátit“, které umožní uživatelům bezpečně a bez stresu experimentovat. Tento princip významně přispívá k pozitivnímu zážitku uživatelů a jejich spokojenosti s produktem. [49][50][51]

4.2.6 Minimalizace kognitivní zátěže

Dobře navržené *UX* minimalizuje kognitivní zátěž uživatelů, tedy množství mentální energie, kterou potřebuje k dokončení úkolu. To znamená prezentovat pouze relevantní informace a používat jasné, jednoduché interakce. Redukce kognitivní zátěže vede k lepšímu uživatelskému prožitku a vyšší pravděpodobnosti, že uživatel úkol dokončí. [51]

4.2.7 Testování a zpětná vazba

Testování použitelnosti a získávání zpětné vazby od uživatelů jsou klíčové komponenty efektivního *UX designu*. Pravidelné testování pomáhá včas odhalit a vyřešit problémy, čímž se snižují náklady na vývoj a zlepšuje celková kvalita produktu. Získaná zpětná vazba by měla být pravidelně implementována do procesu návrhu, čímž se produkt stává více uživatelsky orientovaným a efektivnějším. [50][51]

4.2.8 Design založený na kontextu

UX design by měl reflektovat kontext, ve kterém je produkt používán. To zahrnuje prostředí, zařízení a emoční stav uživatele během interakce s produktem. Kontextový přístup pomáhá vytvářet relevantní produkty, které odpovídají skutečným potřebám a očekáváním. Správně aplikovaný design na základě kontextu výrazně zlepšuje celkovou uživatelskou zkušenost. [51]

II PRAKTICKÁ ČÁST

5 POPIS APLIKACE A JEJÍ ÚČEL

Aplikace je navržena jako komplexní nástroj pro správu provozních dat strojů a zajištění efektivního provozu. Poskytuje uživatelům přehledné zobrazení dat, nahrazuje tradiční provozní knihu a nabízí širokou škálu funkcí, které usnadňují každodenní práci.

Hlavním cílem aplikace je zlepšit sledování výkonu strojů, zvýšit efektivitu práce zaměstnanců a zajistit bezpečné ukládání a snadnou dostupnost všech potřebných informací.

5.1 Hlavní funkce aplikace

Jádrem každé softwarové aplikace je její funkční výbava, která určuje, jak efektivně dokáže naplnit očekávání svých uživatelů. Tato část přibližuje klíčové funkce vyvinuté aplikace, které byly navrženy s ohledem na potřeby provozního prostředí. Představené moduly se zaměřují na sběr a zpracování dat, digitalizaci provozních procesů, sledování efektivity práce, přístup k technické dokumentaci a další podpůrné mechanismy, které přispívají ke zjednodušení každodenního provozu.

5.1.1 Sběr a zpracování dat

Aplikace sbírá data přímo z jednotlivých strojů a následně je zpracovává do podoby, která je snadno interpretovatelná. Data jsou zobrazována pomocí:

- **Grafů:** například vizualizace poměrů provozních režimů stroje (zapnutý, vypnutý, v provozu apod.).
- **Tabulek:** přehledné uspořádání údajů pro snadné vyhledávání a analýzu.
- **UI prvků:** přizpůsobené rozhraní pro rychlý přístup k datům.

5.1.2 Digitální provozní kniha

Jednou z hlavních funkcí aplikace je nahrazení tradiční provozní knihy digitální verzí. Veškerá provozní data, která by se jinak zapisovala ručně, lze nyní snadno zadat přímo v aplikaci. Tento nástroj umožňuje zaznamenávat informace o spotřebě materiálu, údržbě, opravách a dalších klíčových údajích. Data lze dále exportovat do formátů CSV nebo Excel, což usnadňuje jejich následné zpracování nebo sdílení.

5.1.3 Sledování práce zaměstnanců

Aplikace poskytuje funkci pro zaznamenávání pracovní doby zaměstnanců na jednotlivých zakázkách. Tento systém využívá časové stopky, které umožňují přesně sledovat dobu strávenou na konkrétních úkolech. Tyto záznamy lze následně editovat a zobrazovat na hlavní stránce aplikace, což usnadňuje přehled a analýzu efektivity práce.

5.1.4 Statistiky a provozní data strojů

Uživatelé mohou prostřednictvím aplikace sledovat podrobné statistiky jednotlivých strojů. Mezi dostupnými údaji jsou například informace o motohodinách, tedy o tom, kolik času byl stroj v provozu za určité období, jako je den, týden nebo měsíc. Provozní data jsou zobrazována nejen číselně, ale i graficky, což umožňuje rychlejší pochopení souvislostí. Například poměr času, kdy byl stroj zapnutý či vypnutý, je možné vidět ve formě přehledných grafů.

5.1.5 Ekonomický přehled spotřeby

Součástí aplikace je funkce pro sledování ekonomické efektivity provozu jednotlivých strojů. Každý stroj má vlastní přehledovou stránku, kde jsou automaticky zobrazovány provozní a ekonomické údaje získané z digitální provozní knihy. Na základě zadaných dat, jako je spotřeba materiálu, délka provozu nebo typ prováděných činností, se generují interaktivní grafy a tabulky, které umožňují rychlou vizuální analýzu nákladovosti.

5.1.6 Práce s manuály

Další důležitou funkcí aplikace je správa manuálů. Uživatelé mají možnost si stahovat manuály pro jednotlivé stroje ve formátu PDF, případně přidávat a aktualizovat vlastní dokumentaci. Tento systém zajišťuje snadný přístup k důležitým informacím a umožňuje kategorizaci dokumentů podle potřeby.

5.1.7 Přístup do systému

Přístup do aplikace je zabezpečen prostřednictvím přihlašování pomocí emailu a hesla. Registrace nových uživatelů není povolena, což zajišťuje vyšší kontrolu nad tím, kdo má přístup k systému.

6 FUNKCIONÁLNÍ A NEFUNKCIONÁLNÍ POŽADAVKY NA APLIKACI PRO MONITORING TRYSKACÍCH ZAŘÍZENÍ

Analýza požadavků je klíčovým krokem v každém projektu vývoje softwaru nebo systému. Kvalitní definice požadavků určuje, zda bude systém úspěšně splňovat očekávání uživatelů a obchodní cíle. Tyto požadavky lze rozdělit do dvou hlavních kategorií: **funkcionální** a **nefunkcionální**. Každá z těchto kategorií má svůj specifický účel a hraje jedinečnou roli v procesu návrhu a vývoje systému. [52]

6.1 Funkcionální požadavky

Funkcionální požadavky popisují, co systém má dělat, tedy konkrétní funkce, operace nebo chování systému. Jsou zaměřené na potřeby uživatelů a zahrnují vše, co musí být v systému přítomno, aby byl funkční. [52]

6.1.1 Charakteristika funkcionalních požadavků

- Specifikují vstupy, operace a výstupy systému. [52]
- Jsou snadno měřitelné prostřednictvím výsledků nebo testů. [52]
- Jsou přímo viditelné ve finálním produktu. [52]

6.2 Nefunkcionální požadavky

Nefunkcionální požadavky definují, jakým způsobem má systém splnit své úkoly. Zaměřují se na kvalitu systému, jako jsou výkonnost, bezpečnost, škálovatelnost nebo použitelnost. [52]

6.2.1 Charakteristika nefunkcionalních požadavků

- Určují atributy systému, nikoli jeho konkrétní funkce. [52]
- Často se měří prostřednictvím benchmarků nebo SLA (dohod o úrovni služeb). [52]
- Přispívají k celkové spolehlivosti, efektivitě a uživatelské spokojenosti. [52]
- Zaměřují se na atributy systému, jako jsou škálovatelnost, bezpečnost, rychlost odezvy, spolehlivost a snadná udržitelnost. [52]

6.3 Požadavky na systém

Požadavky na funkcionalitu a podobu aplikace byly postupně upřesňovány během několika konzultací se zadavatelem. Tyto konzultace probíhaly osobně, či online, a jejich cílem bylo co nejpresnější definování požadavků již v rané fázi vývoje tak, aby nedocházelo k jejich pozdějším úpravám a bylo zajištěno, že výsledné řešení bude odpovídat očekáváním zadavatele.

6.4 Funkcionální požadavky na systém

ID požadavku	Požadavek
F01	Umožnit uživateli přihlášení do aplikace.
F02	Zobrazení provozních dat technologie dle režimů (off, stand-by, operation, blow-off) a módů (manual/automat).
F03	Možnost vložení zakázky a její parametrizace (např. provozní tlak, průměr trysky, typ trysky, typ abraziva).
F04	Automatické snímání provozního tlaku.
F05	Možnost ručního zadání parametrů (např. průměr trysky, typ trysky, typ abraziva, cena).
F06	Export dat do CSV pro další zpracování, řazení a vyhledávání.
F07	Vytvoření provozní knihy obsahující provozní data, spotřebu materiálu, údržbu, opravy, a další informace.
F08	Možnost doplnění dat do provozní knihy (např. abraziva, spotřeba, údržba, opravy).
F09	Záznam revizí včetně dat, platnosti, odpovědné osoby a připojení dokumentů (např. PDF).
F10	Implementace upozornění (ALERT) s možností správy (vyřešeno/nevyřešeno) a uložení dat do provozní knihy.
F11	Možnost exportu záznamů revizí a upozornění do CSV.
F12	Vložení manuálu včetně obsahu, nastavení, náhradních dílů, a dokumentace ve formátu PDF.
F13	Možnost aktualizace dokumentace a řazení dokumentů podle kategorií.
F14	Zobrazení a export provozní ekonomiky (např. náklady na provoz, spotřebu, opravy, a doby výměny trysky).
F15	Umožnit práci s užitečnými informacemi (např. kontakty, webové odkazy, kalkulačky).
F16	Umožnit záznam pracovní doby zaměstnance na zakázce prostřednictvím časovače. (Zaměstnanec může spustit časovač a zaznamenat dobu strávenou prací na konkrétní zakázce.)
F17	Umožnit filtrování a export provozní knihy jako celku s řazením dle oddílů vstupních údajů nebo podle časového období.
F18	Poskytnout možnost přidávání obrázků nebo PDF souborů k záznamům v provozní knize.
F19	Zajistit funkci odhlášení uživatele z aplikace.

F20	Zobrazit činnost stroje v grafickém formátu, například prostřednictvím Gaussova diagramu.
F21	Umožnit výběr a změnu pracovního stroje z nabídky dostupných zařízení. (Vhodné pro firmy s více stroji.)
F22	Poskytnout možnost editace času a data u záznamů o vykonané práci. (Například úprava času, kdy byla práce na stroji prováděna.)
F23	Zajistit zobrazení statistik stroje podle různých časových období (denní, týdenní, měsíční).
F24	Umožnit zobrazení statistik stroje k určenému datu.
F25	Prezentovat statistiky v grafickém formátu, například prostřednictvím koláčových diagramů.
F26	Zajistit možnost stažení manuálů ve formátu PDF přímo do zařízení uživatele. (Manuály budou dostupné ke stažení jako PDF soubory.)
F27	Vypočítat a zobrazit provozní náklady (např. spotřeba abraziva, opravy, údržba) s možností přiřazení cen z ceníku nebo fakturace.
F28	Implementovat upozornění (ALERT) na údržbu s možností konfigurace předstihu
F29	Umožnit záznam revizí včetně detailů (co, kdy, platnost, kdo zadal)
F30	Umožnit přidávání, aktualizaci a kategorizaci různých typů manuálů (elektro schémata, pneu schémata, výchozí revize, CE prohlášení) ve formátu PDF.

Tabulka 1: Funcionální požadavky

6.5 Nefunkcionální požadavky na systém

ID požadavku	Požadavek
NF01	Aplikace musí být modulární, aby umožňovala postupnou implementaci a rozšiřování.
NF02	Použití přehledného uživatelského rozhraní vhodného pro intuitivní práci.
NF03	Zajištění kompatibility s různými zařízeními (mobilní, tablet).
NF04	Data musí být uložena bezpečně a zálohována v souladu s GDPR.
NF05	Systém musí být schopen zpracovávat a zobrazovat upozornění v reálném čase bez znatelného zpoždění.
NF06	Aplikace musí umožnit export dat ve formátu CSV.
NF07	Systém musí být škálovatelný pro budoucí rozšíření funkcionalit.
NF08	Zajištění vysoké dostupnosti aplikace (např. SLA 99,5 %).
NF09	Responzivní design pro různé velikosti obrazovek.
NF10	Možnost snadné integrace s již existujícími daty a aplikacemi.
NF11	Podpora více uživatelských rolí (administrátor, uživatel).

NF12	Optimalizace výkonu pro rychlé zpracování dat a generování exportů.
NF13	Omezení přístupu k citlivým datům pomocí role-based access control (RBAC).
NF14	Automatické notifikace a připomínky uživatelům.
NF15	Možnost škálování úložiště dat.
NF16	Minimalizace doby odezvy aplikace na uživatelské akce.

Tabulka 2: Nefunkcionální požadavky

7 TECHNOLOGICKÁ ARCHITEKTURA PROJEKTU

Aplikace je postavena na **klient-server** architektuře, kde klientská část komunikuje se serverem prostřednictvím *API*. Pro vývoj byla využita platforma **Ionic** v kombinaci s frameworkem **Vue.js**, konkrétně s využitím **Composition API**. Pro testování *API* volání sloužil **JSON Dummy Server**, který poskytl jednoduché a flexibilní prostředí pro simulaci datových operací.

7.1 Architektura klient-server

Klient-server architektura je síťový model, který umožňuje komunikaci a výměnu dat mezi klienty a servery. Tento model rozděluje systém na dvě hlavní části: **klienta**, který odesílá požadavky, a **server**, který je zpracovává a poskytuje odpovědi. Komunikace probíhá prostřednictvím předem definovaných protokolů, jako je *HTTP* nebo *FTP*. [53]

7.1.1 Důvod zavedení klient-server architektury

Pro tento projekt se architektura klient-server ukázala jako optimální volba, protože umožňuje snadnou škálovatelnost, modularitu aplikace a přehledné oddělení klientské a serverové části. Zároveň je *back-end* část systému již částečně hotová, což umožňuje napojit nový *front-end* model přímo na stávající *back-end* bez nutnosti zásadních úprav základní logiky. Tím se zjednodušuje vývoj a zkracuje čas potřebný na implementaci. [53]

7.1.2 Porovnání s peer-to-peer

V **klient-server** architektuře existuje centrální server, který spravuje a poskytuje zdroje klientům. To umožňuje centralizovanou správu a lepší zabezpečení dat. Naopak v *peer-to-peer* (*P2P*) architektuře fungují všechny uzly jako rovnocenní partneři, kteří mohou současně poskytovat i přijímat služby. *P2P* sítě jsou často levnější na implementaci a nevyžadují centrální server, což může zvýšit jejich odolnost vůči výpadkům.

Peer-to-peer (*P2P*) architektura by pro tuto aplikaci nebyla vhodná, protože vyžaduje centralizovanou správu a bezpečné ukládání dat, jakou je provozní knihy, data získané ze strojů, či záznamy zaměstnanců, což *P2P* model nezajišťuje. Navíc aplikace klade důraz na kontrolu přístupu, zabezpečení v souladu s *GDPR* a spolehlivost infrastruktury, kterou klient-server model nabízí díky centrálnímu serveru. *P2P* by přineslo rizika spojená s nekonzistencí dat, nižší bezpečností a nespolehlivým fungováním při výpadcích uzlů.

Klient-server model také usnadňuje implementaci a testování díky využití standardizovaného *JSON API*. [54]

7.2 Popis způsobu komunikace s API

API volání jsou mechanismy, které umožňují aplikacím vzájemně komunikovat a vyměňovat si data prostřednictvím předem definovaných pravidel. *API* (*Application Programming Interface*) funguje jako rozhraní, které poskytuje přístup k funkcionalitám nebo datům jiných systémů, aniž by bylo nutné znát detaily jejich interní implementace. [55]

7.2.1 Jak funguje volání API

API volání pracuje na principu žádostí a odpovědí:

1. **Žádost:** Klientská aplikace odešle požadavek na *API* server prostřednictvím protokolu, jako je *HTTP* (*GET*, *PUT*, *POST*, *DELETE*). [55]
2. **Zpracování:** Server požadavek zpracuje podle definovaných pravidel, například načte data z databáze nebo provede určité výpočty. [55]
3. **Odpověď:** Server vrátí odpověď zpět klientovi ve formátu, který obě strany podporují. [55]

7.2.2 Formát dat

Pro *API* volání byl vybrán *JSON* (*JavaScript Object Notation*) jako formát pro výměnu dat. *JSON* je lehký, snadno čitelný formát, který umožňuje strukturované zobrazení dat, což usnadňuje jejich interpretaci na klientské straně.

Například odpověď serveru může vypadat takto:

```
{
  "id": "312313",
  "orderId": "453-A",
  "orderName": "Objednavka 1",
  "startTime": "2024-11-04T14:05:23",
  "endTime": "2024-11-04T17:35:51"
}
```

Kód 4: Ukázka JSON formátu

8 NÁVRH APLIKACE

Na návrhu moderní aplikace je důležité nejen zajistit její funkčnost, ale také přívětivý vzhled a snadné používání. Návrh aplikace byl na začátku zpracováván v programu *Figma*, kde byly vytvořeny základní prvky rozhraní, jako je struktura hlavních obrazovek a navigační systém. Zbylé části aplikace byly navrhovány prostřednictvím prototypování, což umožnilo rychlé iterace a efektivní přechod od konceptu k implementaci. Tato metoda přispěla ke zkrácení doby návrhu a usnadnila testování.

8.1 Grafický styl a barevná identita aplikace

Barevná identita společnosti byla definována světle modrým odstínem #007DB1, z něhož byla následně odvozena celá paleta. Primární barva byla určena na základě firemního loga a byla upřesněna během konzultací s klientem. Doplnkové odstíny byly vybírány tak, aby s touto barvou harmonizovaly, a zároveň byla zachována dostatečná kontrastní čitelnost i základní principy *UX*. Při tvorbě palety bylo postupováno metodicky: nejprve byly stanoveny světlé a tmavé varianty primární barvy pro zvýraznění interaktivních prvků, poté byly přidány neutrální tóny pro pozadí a text. Celkový grafický styl aplikace byl tak utvářen tak, aby výsledná vizuální konzistence podporovala firemní identitu a usnadňovala uživatelům orientaci v rozhraní.



Obrázek 3: Primární Barva (od nejsvětější po nejtmavší)

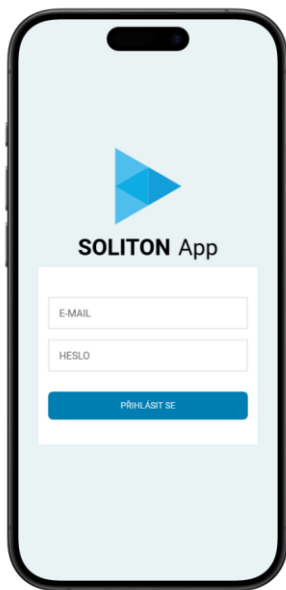
8.2 Návrh obrazovek

Návrh uživatelských obrazovek byl vytvořen s cílem modernizovat stávající aplikaci a přenést její funkčnost na mobilní technologie. Tento proces byl úzce konzultován se zadavatelem, aby bylo zajištěno, že nový design odpovídá jeho požadavkům a očekáváním. Cílem bylo zachovat klíčové funkce a logiku původní aplikace, avšak přizpůsobit je moderním trendům v *UI/UX* designu a potřebám mobilních uživatelů.

Celkový návrh obrazovek byl vytvořen v souladu s vizuální identitou společnosti, což zahrnuje použití jejích firemních barev a grafických prvků.

8.2.1 Přihlašovací obrazovka

Tato obrazovka představuje jednoduché a přehledné přihlašovací rozhraní navržené s ohledem na minimalistický a intuitivní design. Uživatelé se mohou přihlásit pomocí své e-



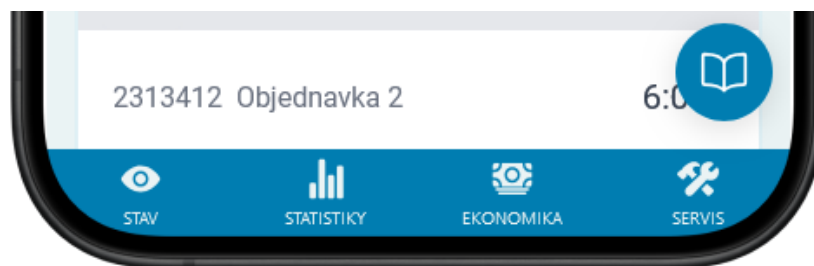
Obrázek 4: Návrh přihlašovací obrazovky

mailové adresy a hesla, přičemž aplikace neumožňuje registraci nových účtů ani resetování hesla.

Design přihlašovací obrazovky byl vytvořen v barvách společnosti, což přispívá k vizuální konzistenci.

8.2.2 Navigace v aplikaci

Navigační systém aplikace byl navržen tak, aby umožňoval snadnou orientaci uživatelů mezi hlavními a doplňkovými funkcemi. Pro základní ovládání slouží spodní navigační lišta, která obsahuje hlavní části aplikace, jako je **Stav**, **Statistiky**, **Ekonomika** a **Servis**.



Obrázek 5: Návrh navigace (Spodní lišta)

Důležitou součástí navigace je také „*floating blutton*“, který je umístěn na pravé dolní straně obrazovky. Tento tlačítko poskytuje rychlý přístup k **provozní knize**, která představuje jednu z klíčových funkcí aplikace.

8.2.2.1 Horní lišta

Horní lišta aplikace obsahuje klíčové navigační prvky. Na levé straně je umístěna ikona „hamburger“ menu pro přístup k boční navigaci s doplňkovými funkcemi, jako jsou manuály nebo export dat. Vpravo se nachází ikona zvonečku pro oznámení, která upozorňuje na důležité události, jako například nadcházející servis, a ikona uživatelského profilu pro přístup k nastavením účtu.

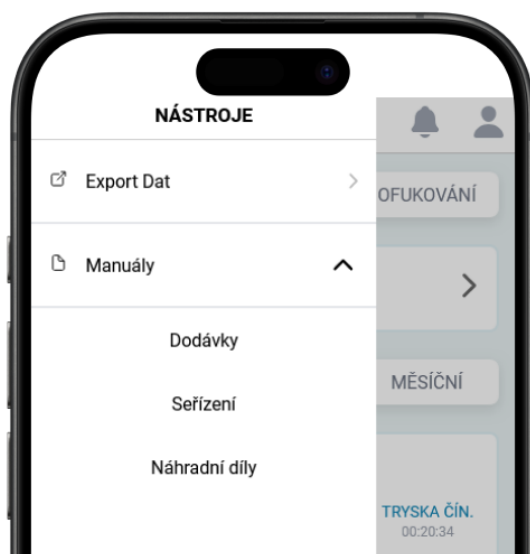


Obrázek 6: Návrh navigace (horní lišta)

Horní lišta zároveň zobrazuje aktuálně vybraný stroj, jehož data si uživatel prohlíží.

8.2.2.2 Rozbalovací lišta

Kvůli větší komplexnosti aplikace bylo potřeba zavést také **sekundární navigaci**, která je dostupná prostřednictvím bočního menu, aktivovaného „*hamburger*“ ikonou.



Obrázek 7: Návrh navigace (rozbalovací lišta)

Toto menu obsahuje doplňkové funkce, například **Export dat** nebo přístup k manuálům. Pro lepší organizaci a přehlednost je sekce „Manuály“ rozdělena do rozbalovací kategorie, jako jsou **Dodávky**, **Seřízení** a **Náhradní díly**. Tento víceúrovňový navigační systém zajišťuje, že aplikace zůstává uživatelsky přívětivá i při větším množství funkcí.

8.2.3 Obrazovka pro oznámení

Stránka oznámení slouží k přehlednému zobrazení všech systémových nebo administrátorem zaslaných zpráv. Tyto informace jsou klíčové pro uživatele, kteří díky nim mohou rychle reagovat na potřebné úkony či aktualizace.

8.2.3.1 Identifikátor nevyřízených oznámení

Číslo zobrazené u ikonky zvonečku v pravé horní části obrazovky indikuje počet nevyřízených (nepřečtených) oznámení. Tento vizuální prvek umožňuje uživateli ihned zjistit, kolik nových informací čeká na jeho pozornost.

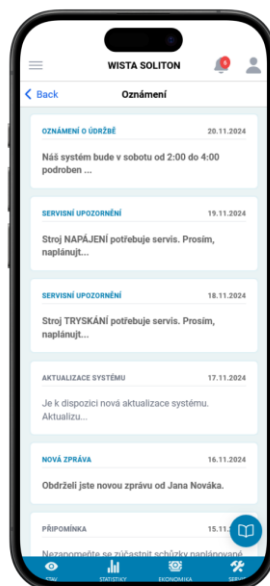


Obrázek 8: Návrh oznámení

8.2.3.2 Stránka s listem oznámení

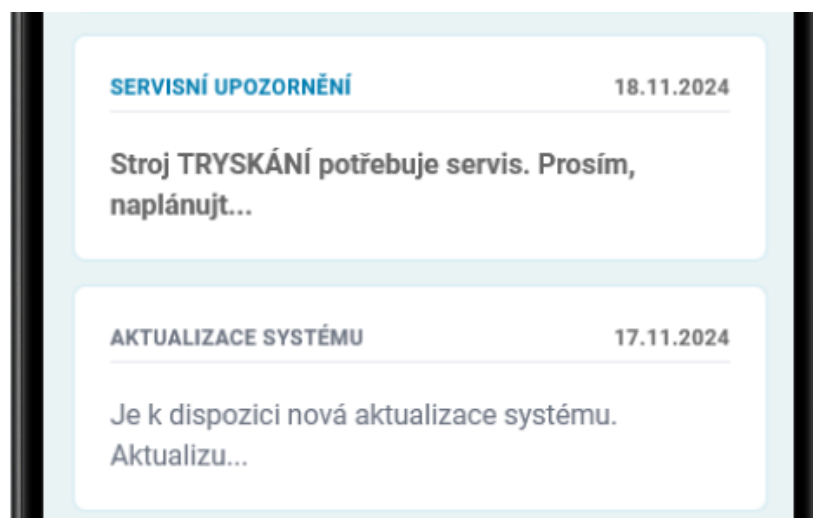
Každé oznámení je na stránce zobrazeno s názvem, datem přijetí a částí textu zprávy.

Pokud je text delší, automaticky se zkrátí a nahradí znakem „...“, což zajišťuje přehlednost i při větším počtu oznámení. Uživatel může zobrazit detail oznámení kliknutím na něj.



Obrázek 9: Návrh listu oznámení

8.2.3.3 Status oznámení



Obrázek 10: Návrh statusu oznámení

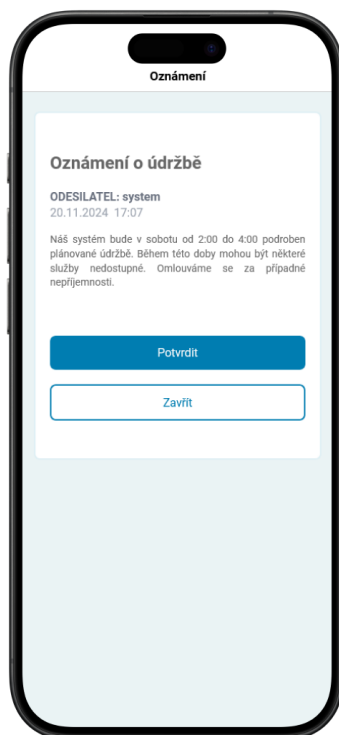
Oznámení mají dvě odlišné vizuální podoby podle svého stavu:

- **Nepřečtená oznámení** mají název zvýrazněný modrou barvou a jejich text je tučně zvýrazněn, aby byla okamžitě zřejmá jejich priorita. Například oznámení "SERVISNÍ UPOZORNĚNÍ" má zvýrazněný název i text.

- **Přečtená oznámení** již mají název zobrazený v šedé barvě a text je ve standardní podobě. Tato změna indikuje, že uživatel již oznámení zpracoval.

8.2.3.4 Detail Oznámení

Po kliknutí na oznámení se uživateli zobrazí detailní informace. Tato obrazovka poskytuje kompletní přehled o konkrétním oznámení, včetně jeho obsahu, data a času odeslání a identity odesílatele.



Obrázek 11: Návrh detailu oznámení

V horní části obrazovky je zobrazen název oznámení spolu s datem a časem jeho odeslání. Pod těmito informacemi se nachází textová část obsahující podrobnosti oznámení, které uživatel potřebuje k tomu, aby mohl adekvátně reagovat.

V dolní části obrazovky jsou k dispozici dvě tlačítka:

- **Potvrdit:** Po stisknutí tohoto tlačítka se oznámení označí jako vyřízené a zmizí z nevyřízených oznámení indikovaných číslem u ikonky zvonečku.
- **Zavřít:** Toto tlačítko umožňuje uživateli opustit detail oznámení bez toho, aby bylo považováno za vyřízené. Notifikace tak zůstane v seznamu nevyřízených.

Tento systém interakce je navržen tak, aby uživateli poskytl maximální flexibilitu a přehled při práci s oznámeními.

8.2.4 Obrazovka stavu stroje

Jednou z klíčových částí aplikace je obrazovka, která zobrazuje **aktuální stav stroje** a jeho nedávnou aktivitu. Tato obrazovka byla navržena tak, aby uživateli poskytla rychlý přehled o provozním stavu jednotlivých částí stroje, jeho využití v průběhu času a historii prováděných zakázek.



Obrázek 12: Návrh úvodní obrazovky (Status stroje)

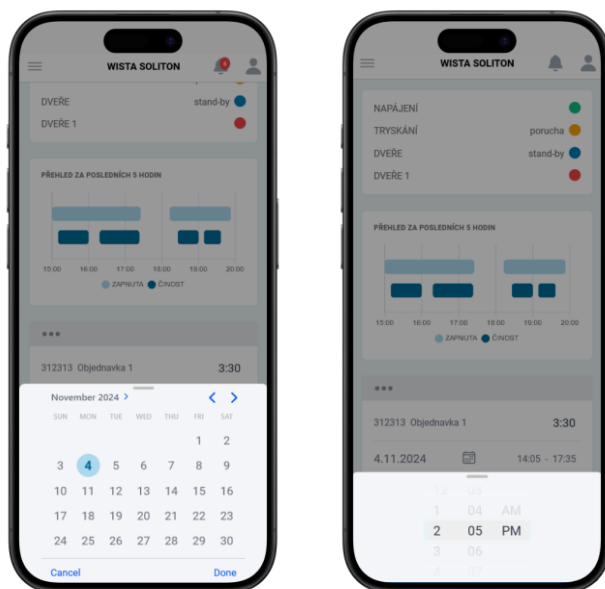
Horní část obrazovky obsahuje indikátory, které vizuálně znázorňují stav jednotlivých komponent stroje. Každý stav je reprezentován barevným symbolem. Zelená barva signalizuje, že stroj nebo jeho část je zapnutá a funkční, zatímco červená značí vypnutí. Další možné stavy, jako například „porucha“ nebo „stand-by“, jsou jasně označeny příslušnými barvami a textovým popisem, což uživateli umožňuje rychle porozumět aktuální situaci.

Pod touto sekci se nachází časový přehled provozu stroje za posledních pět hodin. Tento přehled je zobrazen prostřednictvím grafu, který vizuálně ukazuje, kdy byl stroj zapnut a kdy byl aktivně využíván. Uživatel tak může snadno analyzovat využití stroje během daného časového období.

V další části obrazovky jsou uvedeny informace o jednotlivých zakázkách, které byly na stroji realizovány. Pro každou zakázku jsou zobrazeny detaily, jako je její číslo, datum realizace, přesný časový úsek práce a celková doba trvání. Tato sekce slouží ke snadnému sledování historie zakázek a jejich efektivity.

8.2.4.1 Editace časového úseku práce v zakázce

Součástí obrazovky stavu stroje je také sekce, která umožňuje správu a editaci údajů o jednotlivých zakázkách vykonávaných na stroji. Tato funkcionality je navržena tak, aby uživatelé mohli jednoduše upravit informace o zakázkách, včetně data, kdy byly realizovány, a časového úseku jejich trvání.

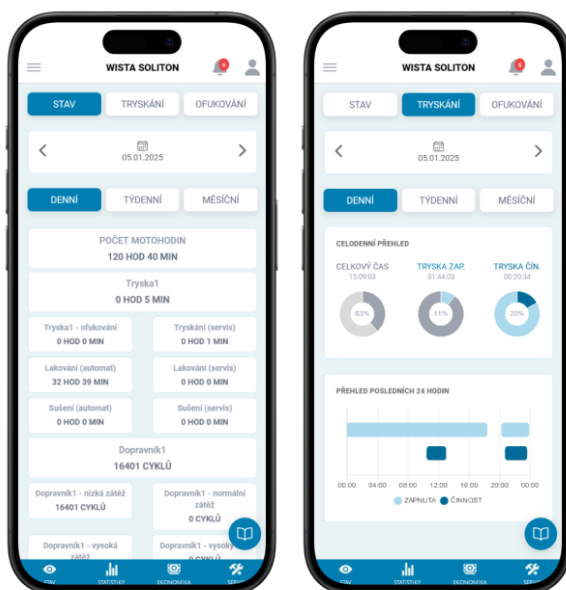


Obrázek 13: Návrh editace času

Při editaci zakázky je možné upravit konkrétní datum pomocí přehledného kalendáře, který umožňuje rychlé a intuitivní výběry. Stejně tak lze detailně nastavit přesné časové rozmezí zakázky. Rozhraní pro výběr času je navrženo tak, aby uživateli poskytovalo co nejvyšší komfort při zadávání požadovaných hodnot.

8.2.5 Obrazovka statistiky stroje

Obrazovka statistik poskytuje detailní přehled o provozu stroje a jednotlivých činnostech, které na něm byly vykonávány. Uživatel má možnost přepínat mezi různými sekcemi, které reprezentují různé technologie, jako například „Stav“ (celkový přehled o stavu stroje), „Tryskání“ a „Ofukování“. Tyto sekce jsou organizovány v horní části obrazovky.



Obrázek 14: Návrh obrazovky pro statistiky stroje

8.2.5.1 Výběr časového období pro zobrazení dat

Jednou z klíčových funkcí je možnost zvolit časové období, pro které se mají data zobrazit. Uživatel si může vybrat mezi denním, týdenním a měsíčním přehledem, přičemž má také možnost upravit konkrétní datum začátku vybraného období.

8.2.5.2 Grafické zobrazení statistických dat

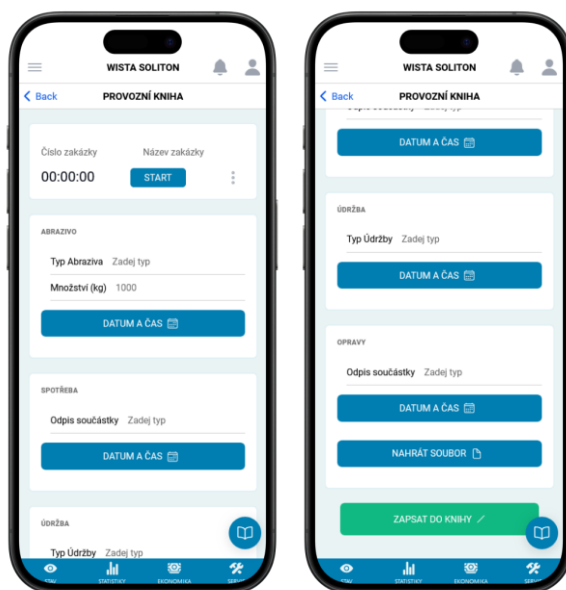
V jednotlivých sekcích jsou zobrazena data o tom, jak dlouho byla daná technologie v provozu. Například u sekce „Tryskání“ je možné vidět, kolik času strávil stroj zapnutý a kolik času byla technologie aktivně využívána. Tyto informace jsou vizualizovány pomocí grafů a číselných údajů, které poskytují přehledný pohled na provozní efektivitu stroje.

V dolní části sekce je umístěn graf, který zobrazuje podrobnější a vizuálně srozumitelný pohled na využití stroje během zvoleného období. Uživatelé tak mohou rychle identifikovat vzory nebo anomálie v provozu stroje a získat lepší představu o jeho výkonu.

8.2.6 Obrazovka provozní knihy

Provozní kniha je jednou z klíčových funkcí aplikace, která slouží k zaznamenávání a správě všech důležitých informací týkajících se strojů a jejich provozu. Tato funkce umožňuje udržovat podrobnou evidenci o aktivitách prováděných na stroji, včetně údržby, oprav, spotřeby materiálů a dalších relevantních údajů. Díky tomuto nástroji mohou

uživatelé snadno sledovat historii provozu a zajistit, že všechny operace jsou pečlivě zaznamenány.



Obrázek 15: Návrh obrazovky pro provozní knihu

8.2.6.1 Časomíra zakázek

V horní části obrazovky provozní knihy se nachází sekce pro správu zakázek. Uživatel může zadat číslo a název zakázky a zahájit měření času stráveného na konkrétní úloze pomocí vestavěného časovače. Tento časovač zajišťuje přesnou evidenci doby trvání jednotlivých zakázek.

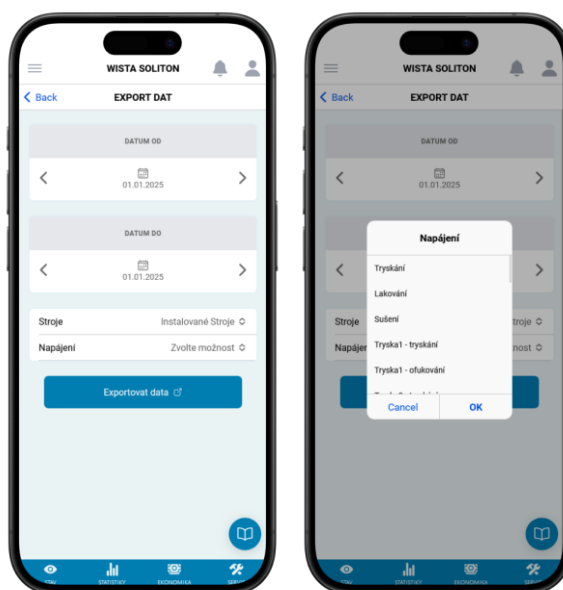
8.2.6.2 Zápis dat

Pod sekci zakázek jsou rozděleny další oblasti zaměřené na specifické činnosti. Například sekce **Abrazivo** umožňuje zaznamenat typ použitého abraziva a jeho množství. V části **Spotřeba** mohou uživatelé evidovat odpisy součástek, zatímco sekce **Údržba** slouží k zadávání typu údržby a připojení dalších podrobností, jako jsou datum a čas. U některých záznamů lze také nahrát soubory, například dokumentaci k opravám nebo jiné relevantní materiály.

8.2.7 Obrazovka exportu dat

Obrazovka pro export dat umožňuje uživatelům vybrat požadované časové období a zobrazit odpovídající data týkající se jednotlivých strojů nebo konkrétních technologických operací.

Uživatel má možnost zadat datum začátku a konce období prostřednictvím přehledného kalendářního výběru.

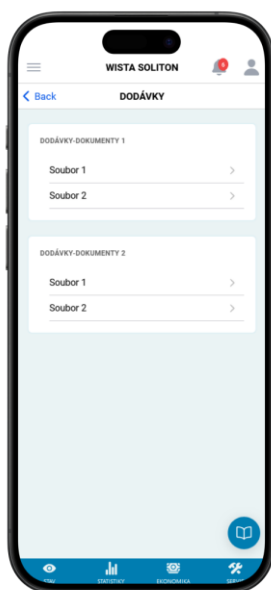


Obrázek 16: Návrh obrazovky pro export dat

Kromě výběru období lze specifikovat také konkrétní stroj a typ operace, například tryskání, lakování, sušení nebo další dostupné možnosti. Tyto volby jsou přehledně uspořádány ve formě rozbalovacích nabídek, které umožňují snadnou a intuitivní selekci.

8.2.8 Obrazovka pro seznam manuálů

Stránka s manuály slouží jako centrální místo pro přístup k dokumentaci a návodům vztahujícím se k různým aspektům systému a jeho údržby. Jak bylo popsáno v části **Rozbalovací lišta**, uživatel má přístup k manuálům prostřednictvím boční navigace, kde jsou rozděleny do přehledných podsekcí, jako jsou například **Dodávky**, **Seřízení** a **Náhradní díly**. Tyto sekce obsahují konkrétní soubory manuálů, které lze stáhnout kliknutím na příslušnou položku.



Obrázek 17: Návrh obrazovky pro manuály

Každý soubor je zobrazen ve formě karty s názvem a symbolem šipky indikující možnost interakce. Kliknutím na konkrétní soubor systém automaticky zahájí jeho stažení, což umožňuje uživateli mít potřebnou dokumentaci snadno dostupnou i offline. Tento design zajišťuje uživatelskou přívětivost a přehlednost, což je klíčové pro rychlý přístup k důležitým informacím při práci se systémem.

9 IMPLEMENTACE APLIKACE

Implementace aplikace byla realizována s využitím moderních technologií, které umožňují efektivní vývoj mobilních a webových aplikací. Jako hlavní frameworky byly zvoleny *Vue.js* a *Ionic*, které poskytují robustní nástroje pro tvorbu responzivních a multiplatformních aplikací. Díky těmto frameworkům bylo možné rychle prototypovat a implementovat klíčové funkce aplikace, přičemž zůstala zachována flexibilita a škálovatelnost pro budoucí rozšíření.

Pro stylování aplikace byl zvolen CSS framework *Tailwind*, který výrazně urychlil práci s návrhem uživatelského rozhraní díky svým užitečným třídám a možnostem snadné konfigurace. Tailwind také umožnil konzistentní aplikaci barevné palety a stylu odpovídající identitě společnosti.

Vzhledem k tomu, že *back-endová* část aplikace není v současnosti plně funkční, byl použit *dummy JSON server* pro simulaci *API* požadavků. Tento přístup umožnil vývojářům testovat a ladit aplikační logiku, jako by již existoval plně funkční *back-end*, což výrazně usnadnilo proces implementace.

9.1 Souborová struktura

```
src/  
├── API/  
├── components/  
├── constants/  
├── dummyJson/  
├── router/  
├── stores/  
├── theme/  
├── utils/  
├── views/  
├── App.vue  
├── main.ts  
├── tailwind.css  
└── vite-env.d.ts
```

Kód 5: Souborová struktura

Struktura tohoto projektu byla navržena tak, aby umožnila přehlednou organizaci kódu a usnadnila jeho údržbu. Každý adresář má jasně definovanou odpovědnost, což přispívá k modularitě a opakovatelné použitelnosti jednotlivých částí.

9.1.1 API

Obsahuje logiku pro komunikaci s *back-end* prostřednictvím *API*. Slouží jako centrální bod, který zajišťuje odesílání a přijímání požadavků (např. *GET*, *POST*). Tato část odděluje přímou práci s *API* od zbytku aplikace, čímž zvyšuje čitelnost a testovatelnost kódu.

Příklad volání API:

```
export const GETMachineStateData = async () => {
  let data;
  try {
    const response = await fetch('http://localhost:8000/machines1');
    if (!response.ok) {
      console.error(`HTTP error! status: ${response.status}`);
    }
    data = await response.json();
  } catch (err) {
    console.error('Error:', err); // Log error to console
    return null;
  }
  return data;
};
```

Kód 6: Ukázka API volání

9.1.2 Components

Tato složka zahrnuje opakovaně použitelné komponenty, které jsou obecné a nezávislé na konkrétních stránkách aplikace. Příkladem mohou být **tlačítka**, **formulářové prvky**, **modální okna** nebo **tabulky**.

9.1.3 Constants

Složka **constants** slouží k ukládání hodnot, které jsou využívány ve více částech aplikace. Tyto hodnoty mohou zahrnovat například textové řetězce, které simulují **enumerátory** (*enums*). V prostředí *JavaScript*, kde nejsou **enumerátory** přímo definovány jako jazyková konstrukce, se místo nich často používají objekty. Tento přístup umožňuje zvýšit čitelnost a konzistenci kódu tím, že se předchází opakování stejných řetězců na různých místech projektu.

```
export const timeSpan = {
  DAILY: 'DENNÍ',
  WEEKLY: 'TÝDENNÍ',
  MONTHLY: 'MĚSÍČNÍ',
  WHOLE: 'CELKOVÝ PŘEHLED',
}
```

Kód 7: Ukázka definovaného objektu

Objekt *timeSpan* slouží k definování tří časových intervalů – **denního**, **týdenního** a **měsíčního**. Jednotlivé klíče, jako například *DAILY*, představují logické názvy, které jsou

konzistentní s implementační logikou aplikace. Hodnoty těchto klíčů (například 'DENNÍ') slouží buď jako **vizuální reprezentace** v uživatelském rozhraní, nebo jako **vstupní hodnoty v rozhodovacích funkcích**, jako je například podmíněná logika ve funkcích typu „if“. Tato struktura přispívá ke zvýšení čitelnosti a přehlednosti kódu.

9.1.4 DummyJson

Simuluje back-endovou logiku pomocí *mock* dat. Tato složka obsahuje vzorová data ve formátu *JSON*, která slouží jako odpovědi na *API* požadavky.

9.1.5 Router

Zajišťuje správu směrování aplikace pomocí routeru **Vue.js**. Obsahuje definice jednotlivých cest (*routes*) a jejich propojení s odpovídajícími komponentami nebo *views*.

9.1.6 Stores

Tato složka obsahuje logiku pro správu stavu aplikace pomocí knihovny *Pinia*. Správa stavu je důležitá pro sdílení dat mezi komponentami a uchování globálního stavu aplikace.

```
export const useStatisticsStore = defineStore('statisticsStore', {
  state: () => ({
    timePeriod: timeSpan.DAILY,
    selectedTechnology: technologyState.TECHSTATE,
    selectedTechnologyState: technologyState.TECHSTATE,
    date: '',
  }),
  getters: {
  },
  actions: {
  }
})
```

Kód 8: Příklad Pinia store

9.1.7 Theme

Složka *theme* obsahuje soubor **variable.css**, který slouží k definování globálních stylů a přizpůsobení designu aplikace. Tento soubor umožňuje jednotné ovládání stylů všech komponent knihovny *Ionic*. Prostřednictvím tohoto souboru lze například nastavovat barevné schéma, fonty, rozměry či další vizuální prvky, což zajišťuje konzistenci vzhledu celé aplikace a usnadňuje budoucí úpravy designu.

9.1.8 Utils

Zahrnuje různé pomocné funkce a nástroje, které usnadňují opakované úlohy v aplikaci. Tyto funkce jsou obecné povahy a nezávislé na konkrétní části aplikace.

9.1.9 Views

Obsahuje jednotlivé pohledy aplikace, které reprezentují celé stránky. Tyto pohledy kombinují různé komponenty dohromady a propojují je s daty a logikou.

9.1.10 Hlavní Soubory

- **App.vue:** Hlavní komponenta aplikace, která je kořenovou částí celé aplikace.
- **Main.ts:** Vstupní bod aplikace, kde je zajištěna inicializace aplikace a její propojení s dalšími knihovnami (např. *router*, *store*).
- **Tailwind.css:** Stylovací soubor obsahující konfiguraci pro framework *Tailwind CSS*.
- **Vite-env.d.ts:** Deklarace typů pro *TypeScript* v prostředí *Vite.js*.

9.2 Navigace a směrování (Routing)

Navigace v aplikaci je realizována pomocí technologie **Vue Router**, konkrétně s využitím `@ionic/vue-router`, který rozšiřuje standardní *Vue Router* o integraci s frameworkem *Ionic*. Tento přístup umožňuje pohodlné řízení směrování mezi jednotlivými obrazovkami (*views*) a zároveň podporuje mobilní přechody a animace.

9.2.1 Statický routing

Základní struktura routingu je definována v souboru `router/index.ts`, kde jsou pomocí objektu `routes` specifikovány jednotlivé cesty (`path`) a komponenty (`component`), které se na dané adrese vykreslují. Například cesta k hlavnímu přehledu ekonomických dat je definována v Kód 9: Statický routing na stránku „*EconomyPage*“.

```
{  
  path: 'EconomyPage',  
  component: () => import('@views/EconomyPage.vue')  
},
```

Kód 9: Statický routing na stránku „*EconomyPage*“

Tato stránka je dostupná pod adresou `http://localhost:8100/tabs/EconomyPage`.

9.2.2 Dynamický routing

Aplikace využívá i **dynamický routing**, tedy přesměrování na základě proměnných parametrů. Typickým příkladem je stránka detailu konkrétní zakázky.

```
{
  path: 'OrderDetailStatisticsPage/:id',
  name: 'OrderDetailStatisticsPage',
  component: () => import('@/views/OrderDetailStatisticsPage.vue')
},
```

Kód 10: Dynamický routing detailu zakázky

Při přechodu na tuto stránku se v *URL* adrese předává parametr *id*, např. `http://localhost:8100/tabs/OrderDetailStatisticsPage/123`, kde 123 je konkrétní identifikátor zakázky.

9.2.3 Navigace pomocí komponent a funkcí

Navigaci mezi jednotlivými částmi aplikace lze provádět buď staticky pomocí komponenty

```
<ion-tab-button>.
<ion-tab-button tab="Economy" href="/tabs/EconomyPage">
  <ion-icon aria-hidden="true" :icon="cash" />
  <ion-label>EKONOMIKA</ion-label>
</ion-tab-button>
```

Kód 11: Navigace pomocí komponenty `<ion-tab-button>`

nebo dynamicky skriptem – např. při kliknutí na kartu nebo tlačítko se zavolá funkce, která programově přesměruje uživatele.

```
export function changeTabs(tabPath) {
  router.push(tabPath)
}
```

Kód 12: Navigace pomocí JavaScriptu

Volání této funkce může být navázáno na událost `@click` z *Vue.js*.

```
<ion-card @click="changeTabs('/tabs/Orders')" class="lg:w-full">
  <ion-card-content>
    ...
  </ion-card-content>
</ion-card>
```

Kód 13: Navigace pomocí události

Tímto způsobem se zajistí přehledné a uživatelsky přívětivé přepínání mezi sekcemi aplikace bez nutnosti znovu načítat celou stránku.

9.2.4 Závěr k směrování

Použití *Vue Routeru* ve spojení s *Ionicem* poskytuje flexibilní a modulární způsob navigace. Struktura směrování včetně podpory parametrizovaných cest umožňuje snadnou

rozšiřitelnost aplikace a přehledné oddělení jednotlivých funkčních celků. *URL* jsou srozumitelné a dobře reflektují hierarchii aplikace, což přispívá k lepší orientaci uživatelů.

9.3 Komponentová architektura

Moderní vývoj frontendových aplikací ve frameworku *Vue* se opírá o komponentový přístup, který podporuje modulárnost, znouvopoužitelnost a lepší udržitelnost kódu. Komponenty představují samostatné části uživatelského rozhraní, které mohou mít vlastní logiku, šablonu i stylování. V této části je popsáno, jak jsou komponenty v rámci projektu strukturovány, jakým způsobem se registrují a jak se následně používají v rámci šablon.

9.3.1 Struktura souborů komponent

Komponenty jsou v projektu rozděleny do tematických složek podle jejich použití. Některé složky odpovídají **konkrétním částem aplikace**, jako jsou např. `viewsComponents/state`, `notifications` nebo `manuals`, kde se nachází komponenty určené pro konkrétní sekce zobrazení. Vedle toho existují složky jako `charts`, `inputs` nebo `buttons`, **které obsahují obecnější komponenty** využitelné napříč více částmi aplikace.

9.3.2 Import a registrace komponenty

Pro použití komponenty v jiné části aplikace je nutné ji nejprve importovat. Importy se zapisují v horní části `.vue` souborů. Například komponenta `ProgressBarComponent` se importuje následovně.

```
import ProgressBarComponent from
"@/components/charts/ProgressBarComponent.vue";
```

Kód 14: Import komponenty `ProgressBarComponent`

9.3.3 Použití komponenty v šabloně

Jakmile je komponenta importována, může být v šabloně použita. Existují dva způsoby zápisu – s pomlčkami nebo s *camelCase* názvem.

```
<ProgressBarComponent />
<progress-bar-component />
```

Kód 15: Ukázka použití komponent v šabloně

Kromě samotné funkčnosti lze komponentě přiřazovat i vizuální vlastnosti prostřednictvím tříd. Například pomocí utility třídy z *Tailwind CSS* `class="lg:w-full"` se zajistí, že komponenta bude mít plnou šířku (`w-full`) při velkém rozlišení (`lg:`). Tímto způsobem lze efektivně stylizovat komponenty zvenku bez nutnosti zasahovat do jejich interní struktury nebo *CSS* stylů.

9.4 Přenos dat mezi vrstvami aplikace

V moderních aplikacích založených na komponentové architektuře je nezbytné zajistit efektivní přenos dat mezi jednotlivými komponentami, vrstvami logiky a zdroji dat. V tomto projektu je datová komunikace realizována různými způsoby v závislosti na úzkém propojení komponent, sdíleném stavu nebo komunikaci s *back-endem*.

9.4.1 Příjem a spotřeba dat v komponentě

Nejzákladnějším způsobem předávání dat mezi komponentami je přes tzv. props. V rodičovské komponentě se definují data jako atributy.

```
<machine-type-modal-component v-if="machineTypeSelectors.length"
                              @modal-closed="modalOpen=false"
                              @selected-machine-id="handleMachineChange"
                              :modal-open="modalOpen"
                              :machine-type-selectors="machineTypeSelectors">
</machine-type-modal-component>
```

Kód 16: Ukázka předání dat do komponenty

V komponentě se pak definují props tímto způsobem.

```
const props = defineProps([
  "modalOpen",
  "machineTypeSelectors"
]);
```

Kód 17: Definování props v komponentě

Nebo v rozšířené podobě, kde je možné nastavit i typy a výchozí hodnoty.

```
const props = defineProps({
  modalOpen: {
    type: Boolean,
    required: true,
    default: false
  },
  machineTypeSelectors: {
    type: Array,
    required: true,
    default: () => []
  }
});
```

Kód 18: Rozšířená podoba props

Pro spotřebu props v šabloně pak může být použité předpona `props.`

```
<div v-if="props.machineTypeSelectors.length">
```

Kód 19: Ukázka použití props v šabloně

Alternativně se dá využít v **šabloně** i bez použití předpony `props.`

```
<div v-if="machineTypeSelectors.length">
```

Kód 20: Ukázka použití props v šabloně 2

Pro spotřebu komponenty v JavaScriptové části musí mít proměnná vždy předponu `props.`

```
const modalTrigger = computed(() => {
  return props.modalOpen;
})
```

Kód 21: Ukázka použití props v JavaScript části

9.4.1.1 Emitování událostí (*emits*)

Komponenty mohou kromě příjmu dat také vysílat události zpět rodiči. Ve *Vue 3* pomocí *Composition API* se události definují následovně.

```
const emit = defineEmits([
  "modalClosed",
  "selectedMachineId"
]);
```

Kód 22: Definování emitovatelných událostí

```
function confirm() {
  emit("selectedMachineId", selectedMachineId.value);
  emit("modalClosed", false);
}
```

Kód 23: Emitování události

V šabloně pak rodič reaguje na tyto události takto.

```
<machine-type-modal-component v-if="machineTypeSelectors.length"
  @modal-closed="modalOpen=false"
  @selected-machine-id="handleMachineChange"
></machine-type-modal-component>
```

Kód 24: Zachytávání emitu v rodičovské komponentě

V komponentě se pak *emit* aktivuje voláním např.: `emit('modalClosed', false)` jak je možno vidět v Kód 23: Emitování události. V rodiči se zachytává pomocí `@modal-closed`.

Emit slouží k jednoduché zpětné komunikaci, např. zavření *modalu* nebo výběr hodnoty. Pokud ale dat proudí více směry a přes větší počet komponent, je vhodnější zvolit centrální správu stavu.

9.4.1.2 Předávání složitějších struktur

Při větším množství dat nebo složitější struktuře je výhodnější předat do komponenty celý objekt stejným způsobem jako kdyby se předávala proměnná.

9.4.2 Vnořování komponent a prop-drilling vs. provide/inject

Při vývoji složitějších aplikací často nastává situace, kdy je potřeba předat data mezi komponentami, které spolu nejsou přímo propojené. *Vue* nabízí více způsobů, jak tento přenos zrealizovat – od klasického **prop-drillingu** přes mechanismus **provide/inject**, až po využití knihovny pro správu stavu jako je **Pinia**.

9.4.2.1 Prop-drilling

Prop-drilling znamená předávání **props** skrze několik úrovní komponent, i když je využívá až komponenta hluboko ve stromu.

```
<!-- ParentComponent.vue -->
<MiddleComponent :title="title" />

<!-- MiddleComponent.vue -->
<ChildComponent :title="title" />

<!-- ChildComponent.vue -->
<p>{{ title }}</p>
```

Kód 25: Ukázka prop-drilling

Tento přístup je vhodný pro menší komponenty, ale při větším rozsahu je přehlednější použít jiné řešení jako **Pinia** nebo *provide/inject*.

9.4.2.2 Provide / Inject

Alternativou k prop-drillingu je *provide/inject*, které umožňuje předat hodnotu přímo mezi rodičem a vzdáleným potomkem.

```
// Parent.vue
provide('themeColor', 'blue')

// DeepChild.vue
const themeColor = inject('themeColor')
```

Kód 26: Ukázka Provide / Inject

Tento mechanismus se v projektu nevyužíval, ale je vhodný například pro globální konfigurace nebo designové kontexty.

V rámci tohoto projektu byl *prop-drilling* využit pouze výjimečně, většina zanořených předávání dat probíhala přes *store* pomocí knihovny **Pinia**.

9.4.3 Centrální správa stavu pomocí Pinia

Pinia je moderní knihovna pro správu stavu, která nahradila *Vuex* jako *oficiální state management* nástroj pro **Vue 3**. V tomto projektu slouží ke sdílení ukládání hodnot mezi různými komponentami bez nutnosti zanořeného *prop-drillingu* či nadměrného použití *emitů*.

```
import {defineStore} from 'pinia';

export const useOperationsBookStore = defineStore('operationsBookStore',
{
  state: () => ({
    orderName: '',
    orderNumber: '',
    workTimerStartTime: '',
    workTimerEndTime: '',
    abrazeType: '',
    abrazePrice: null,
    ...
  })
})
```

Kód 27: Ukázka Pinia Store

Aby bylo možné hodnoty ze *store* používat v komponentě, je potřeba *store* nejdříve inicializovat.

```
import {useOperationsBookStore} from
"@/stores/useOperationsBookStore.js";

const operationStore = useOperationsBookStore()
```

Kód 28: Inicializace store v kódu

Použití v komponentě pak probíhá například pomocí `v-model`.

```
<ion-input label="Cena (kč/kg)" type="number" placeholder="1000"
  v-model="operationStore.abrazePrice" min="0"></ion-input>
```

Kód 29: Změna hodnoty v Pinia store pomocí `v-model`

Nebo zobrazováním hodnoty.

```
<div v-show="operationStore.serviceType !== null" class="my-auto ml-auto
text-zinc-400">{{ operationStore.abrazeType }}</div>
```

Kód 30: Zobrazení hodnoty z Pinia store

Pinia tak umožňuje efektivní sdílení stavu napříč celou aplikací a redukuje potřebu mezi komponentní komunikace.

9.4.4 Mock back-end – JSON Server

Pro účely testování byl použit JSON Server, jednoduchý **mock back-end**, který byl spuštěn lokálně na adrese `localhost:8000`. Poskytoval statické odpovědi z *JSON* souboru.

```
"users": [
  {
    "id": 1,
    "email": "test.maill@email.com",
    "name": "Jan Novák",
    "company": "Wista s.r.o",
    "password": "secret123",
    "avatar": ""
  },
  ...
],
```

Kód 31: Ukázka JSON struktury

Tento server simuloval odpovědi *REST API*, což usnadnilo vývoj bez potřeby plně funkční *back-end* části aplikace.

9.4.5 Komunikace front-endu s JSON Serverem / REST API

Všechny API volání jsou centralizována v samostatných souborech (např. `API/APImachines.js`). Tato struktura usnadňuje úpravy, ladění a opětovné použití *API* funkcí.

```
export const GETTechnologyStateData = async () => {
  let data
  try {
    const response = await
    fetch('http://localhost:8000/machine1StateData')
    if (!response.ok) {
      console.error(`HTTP error! status: ${response.status}`);
    }
    data = await response.json();
  } catch (err) {
    console.error('Error:', err); // Log error to console
    return null;
  }
  return data;
}
```

Kód 32: Ukázka GET requestu

Dynamické parametry lze jednoduše předat pomocí šablonového řetězce.

```
export const GETOrderDetails = async (orderId) => {
  let data
  try {
    const response = await
    fetch(`http://localhost:8000/orderDetailsInfo?id=${orderId}`)
  }
```

```
    if (!response.ok) {  
        console.error(`HTTP error! status: ${response.status}`);  
    }  
    data = await response.json();  
} catch (err) {  
    console.error('Error:', err); // Log error to console  
    return null;  
}  
return data;  
}
```

Kód 33: Ukázka GET requestu s dynamickým parametrem

9.5 Přihlášení uživatele a autentizace

V rámci vývoje bylo zapotřebí implementovat mechanismus pro přihlášení uživatele a ochranu jednotlivých částí aplikace. Pro účely testování byl zvolen nástroj `json-server-auth`, který poskytuje jednoduché, ale funkční *JWT* ověřování uživatelů pro vývojová prostředí. Celý proces autentizace zahrnuje **přihlášení uživatelů, uchování a práci s JWT tokenem, ochranu routingu a zabezpečení API požadavků**.

9.5.1 Inicializace a konfigurace `json-server-auth`

Pro spuštění serveru s autentizační vrstvou byl využit balíček `json-server-auth`. Tento nástroj staví na knihovně `json-server` a přidává *JWT* autentizaci, kterou lze použít při prototypování nebo vývoji aplikací.

V rámci projektu se server spouští následujícím příkazem, který je zapsán ve `package.json` a spouští se v konzoli pomocí příkazu `npm run start`.

```
"start": "json-server-auth -w src/dummyJson/DummyData.json --port 8000"
```

Kód 34: Skript pro spuštění JSON serveru s auth

Základní datový soubor (v tomto případě `DummyData.json`) musí obsahovat kolekci `users`, protože `json-server-auth` automaticky očekává, že uživatelé budou v této sekci.

```
"users": []
```

Kód 35: JSON objekt v `DummyData.json`

Pro úspěšnou registraci a přihlášení je nezbytné, aby uživatel měl vyplněné pole `email` a `password`. [56]

9.5.2 Získání JWT tokenu a hashování hesel

Po odeslání přihlašovacích údajů pomocí POST požadavku na `/login` endpoint dojde k ověření identity uživatele. Pokud je autentizace úspěšná, server odpoví objektem obsahujícím `accessToken` a informace o uživateli.

```
{
  "accessToken": "xxx.xxx.xxx",
  "user": { "id": 1, "email": "..."}
}
```

Kód 36: AccessToken

Tento token je následně uložen do `localStorage` klienta pro další práci.

```
localStorage.setItem('session', JSON.stringify({token, user}));
```

Kód 37: Vložení accessTokenu do localStorage

Kód 38: Login request ukazuje implementaci funkce `login`, která zajišťuje komunikaci s *JSON* serverem

```
export const login = async (email, password) => {
  try {
    const response = await fetch(`${API_URL}login`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({email, password}),
    });

    if (!response.ok) {
      console.log("Login Failed")
      return null
    }
    const {accessToken: token, user} = await response.json();

    localStorage.setItem('session', JSON.stringify({token, user}));

    return {token, user};
  } catch (err) {
    console.error('Login failed:', err);
    return {ok: false, token: null};
  }
};
```

Kód 38: Login request

Token slouží jako klíč pro budoucí autorizované požadavky na server. Hesla v databázi jsou uložena hashovaně pomocí algoritmu `bcrypt`, což je patrné např. z pole `password`, které obsahuje hashovaný řetězec místo běžného hesla. [56]

9.5.2.1 Odhlášení uživatele

Odhlášení je nedílnou součástí autentizační logiky. V tomto projektu se odhlášení provádí odstraněním relace ze `localStorage` a vynulováním všech souvisejících údajů o uživateli a tokenu. Následně je uživatel přesměrován zpět na přihlašovací obrazovku.

Funkce `logout()` je definována ve `useUserStore` a zajišťuje kompletní ukončení relace.

```
logout() {  
  localStorage.removeItem('session');  
  this.user = null;  
  this.token = null;  
  changeTabs('/login/LoginPage')  
}
```

Kód 39: Metoda pro odhlášení uživatele

9.5.3 Obnova relace a práce s tokenem na front-endu

Aby uživatel nemusel být znovu přihlášen při každém načtení stránky, token a informace o uživateli jsou ukládány do `localStorage`. Následně je pomocí metody `hydrate()` v rámci `useUserStore` stav znovu načten.

```
hydrate() {  
  if (this.token) return  
  const sess = restoreSession();  
  if (!sess || !isTokenValid(sess.token)) return;  
  
  this.user = sess.user;  
  this.token = sess.token;  
},
```

Kód 40: Metoda `hydrate` pro vložení tokenu a informací z aktuální session

Funkce `restoreSession()` provádí načtení relace ze `localStorage` a převod do objektu.

```
export function restoreSession() {  
  try {  
    const raw = localStorage.getItem('session');  
    return raw ? JSON.parse(raw) : null;  
  } catch (_) {  
    return null;  
  }  
}
```

Kód 41: Funkce pro obnovení aktuální session z `localStorage`

9.5.3.1 Validace platnosti tokenu

Pro ověření, zda je uložený *JWT* token stále platný a neexpiroval, se používá následující funkce `isTokenValid`. Ta dekoduje druhou část *JWT* (tzv. *payload*), získá z ní čas vypršení platnosti (`exp`) a porovná ho s aktuálním časem.

```
export function isTokenValid(token) {
  try {
    const payload = JSON.parse(atob(token.split('.')[1]));
    return payload.exp * 1000 > Date.now();
  } catch (_) {
    return false;
  }
}
```

Kód 42: Funkce pro validace tokenu

9.5.4 Chráněný routing ve Vue Routeru

Přístup k určitým stránkám aplikace je omezen pouze na přihlášené uživatele. To se řeší pomocí meta vlastnosti `requiresAuth` v definici rout, jak je znázorněno v ****

```
{
  path: '/tabs/',
  component: TabsPage,
  meta: { requiresAuth: true },
  ...
}
```

Kód 43: Pole `requiresAuth` u routingu

Před každou změnou routy se volá funkce `router.beforeEach`, která ověřuje přihlášení uživatele a případně ho přesměruje na login stránku.

```
router.beforeEach((to) => {
  const store = useUserStore()

  store.hydrate()

  if (to.matched.some(r => r.meta.requiresAuth) && !store.isLoggedIn) {
    return { path: '/login/LoginPage', query: { redirect: to.fullPath } }
  }

  if (to.path.startsWith('/login') && store.isLoggedIn) {
    return { path: '/tabs/StatePage' }
  }

  return true
})
```

Kód 44: Kontrola platného tokenu při každé změně aktuální URL

9.5.5 Ochrana API

Všechna *API* volání, která vyžadují autentizaci, jsou zabezpečena pomocí funkce `fetchWithAuth`, která automaticky připojí JWT token do hlavičky `Authorization`. V případě neplatného tokenu dojde k odhlášení uživatele.

```
export async function fetchWithAuth(url, options = {}) {
  const store = useUserStore()
  const headers = { ...(options.headers ?? {}) }

  if (store.token) headers.Authorization = `Bearer ${store.token}`

  const response = await fetch(url, { ...options, headers })

  if (response.status === 401 || response.status === 403) {
    store.logout()
    throw new Error('Unauthorized')
  }

  return response
}
```

Kód 45: Chráněný fetch, který kontroluje, zda token je validní

9.5.6 Přihlašovací formulář

Součástí autentizačního procesu je jednoduchý přihlašovací formulář, který slouží jako vstupní brána do zabezpečené části aplikace. Formulář je implementován v komponentě **LoginPage.vue** a využívá knihovnu *Ionic* pro strukturování vstupních polí a rozvržení uživatelského rozhraní.

```
<form @submit.prevent="handleLogin" class="w-5/6">
  <ion-card class="py-10 px-4">
    <ion-list>
      <ion-item lines="none" class="mb-4 border-2">
        <ion-input
          type="email"
          placeholder="E-MAIL"
          v-model="email"
          shape="round"
        />
      </ion-item>

      <ion-item lines="none" class="border-2">
        <ion-input
          type="password"
          placeholder="HESLO"
          v-model="password"
        />
      </ion-item>
    </ion-list>
    <ButtonComponent type="submit"
      class="mt-6"
      :label="'PŘIHLÁSIT SE'"
      :button-type="'primary'"
```

```
      :button-setting="'primary'"
      :button-top-padding="3"
    />
  </ion-card>
</form>
```

Kód 46: Formulář pro přihlášení uživatele

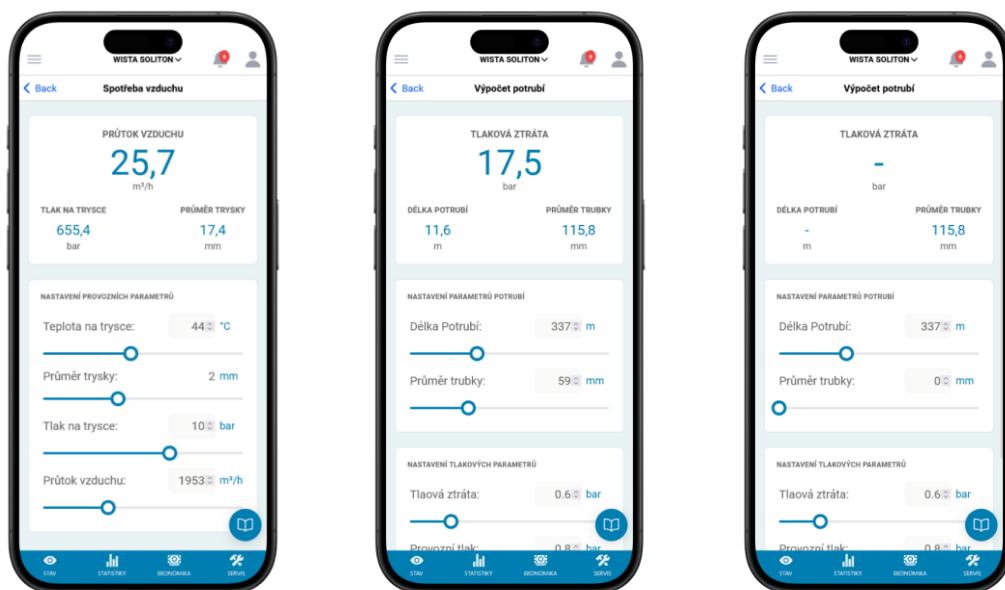
Po zadání e-mailu a hesla do vstupních polí, které jsou propojeny pomocí *v-model* se stavovými proměnnými, se uživatel autentizuje voláním asynchronní funkce `handleLogin()`. Tato funkce předá údaje na *back-end* a v případě úspěšného přihlášení uloží token a informace o uživateli do **Pinia store**. Následně je uživatel automaticky přesměrován na hlavní stránku aplikace (*/tabs/StatePage*).

9.5.7 Shrnutí přihlášení uživatele a autentizace

Celý autentizační mechanismus je vhodný pro testovací nebo vývojová prostředí, kde je zapotřebí rychle otestovat ochranu rout nebo zabezpečení přístupu k *API* bez nutnosti reálné *back-end* části aplikace. *Json-server-auth* poskytuje flexibilní a srozumitelné řešení na základě *JWT* tokenů a simulovaných dat. [56]

9.6 Detailní rozbor obrazovky „Kalkulátory“

Součástí aplikace jsou také praktické technické kalkulátory, které slouží k okamžitému výpočtu provozních parametrů na základě hodnot zadaných uživatelem. Tyto nástroje umožňují rychlou orientaci v klíčových veličinách, jako je průtok vzduchu, tlak na trysce nebo průměr trysky, a významně tak přispívají k efektivitě práce operátorů a přesnému nastavení výrobního procesu.



Obrázek 18: Obrazovka kalkulátorů

Výpočty jsou prováděny v reálném čase – jakmile uživatel změní vstupní hodnoty pomocí posuvníků (*range sliders*) nebo přímého zápisu, výsledné hodnoty se okamžitě aktualizují. Přehledně zobrazené výsledky včetně jednotek jsou navrženy s důrazem na čitelnost. Pokud některý z požadovaných vstupů chybí nebo je neplatný, výstupní hodnota se nahradí znakem „–“, čímž je uživatel intuitivně upozorněn na potřebu doplnění údajů.

9.6.1 Předávání dat a výpočty

Vstupní hodnoty jsou zpracovávány pomocí *Vue* *ref* proměnných a *@emit* událostí z komponent *RangeInputComponent* a *CustomSteppedRangeInputComponent*. Každý vstupní parametr (např. teplota, tlak, průměr) volá specifickou *handler* funkci (*handleEmitValuesNozzlePressure*, *handleEmitValuesAirFlow* aj.), která následně spouští přepočítání všech závislých hodnot pomocí matematických funkcí.

```
function handleEmitValuesNozzleDiameter(value) {
  nozzleDiameter.value = value
  airFlowResult.value = computeTeoreticalAirFlow(
    nozzlePressure.value,
    nozzleDiameter.value,
    compressedAirTemp.value
  )
  nozzlePressureResult.value = computeTeoreticalNozzlePressure(
    airFlow.value,
    nozzleDiameter.value,
    compressedAirTemp.value
  )
}
```

Kód 47: Ukázka handler funkce

9.6.2 Výpočtové funkce

Průtok vzduchu je vypočten na základě **Bernoulliho rovnice** s korekcemi pro stlačený vzduch.

```
const res =
  ((nozzlePressure + 1) / 1) *
  (innerDiameter / Math.pow(1000, 2)) *
  Math.pow(2 / 2.4, 1 / 0.4) *
  Math.sqrt(2 * 9.81 * 29.27 * (273.13 + compressedAirTemp) * (1.4 /
    2.4)) * 60 * 60;
```

Kód 48: Kód pro výpočet Bernoulliho rovnice

Tlak na trysce pro daný průtok se počítá inverzně z výše uvedeného vztahu.

```
const resultBar = (ambientPressureBar * airFlow) / denominator -
  ambientPressureBar;
```

Kód 49: Výpočet tlaku na trysce

Průměr trysky se dopočítá z daného průtoku a tlaku převodem průřezu na průměr.

```
const innerNozzleDia = numerator / denominator;
const result = Math.sqrt((4 * innerNozzleDia) / Math.PI);
```

Kód 50: Výpočet průměru trysky

Díky těmto funkcím si uživatel může například zadat požadovaný průtok a aplikace mu doporučí odpovídající tlak nebo průměr trysky.

9.6.3 Komponenty a architektura

Každý vstupní parametr je reprezentován vlastní komponentou s oddělenou logikou pro práci s hodnotou, validací vstupu a zobrazení jednotky.

```
<ion-input
  v-model="displayValue"
  type="number"
  lang="en"
  class="text-right w-full text-xl"
  @ionInput="updateIndex($event)"
/>
<ion-range v-model="index" :min="0" :max="steps.length - 1" :step="1"
: value="steps[0]" :ticks="props.ticks"
: snaps="props.snaps" @ionInput="updateDisplay"></ion-range>
```

Kód 51: Input pro kalkulačtor

Tato komponenta umožňuje kombinaci plynulého posuvu i ručního zadání hodnoty s přesnou zpětnou vazbou.

Komponenty jako `CalculatedDataResultIndicatorComponent` nebo `SmallerCalculatedDataResultIndicatorComponent` slouží k vizuálnímu zobrazení vypočtených hodnot ve formátu s velkými číslicemi a popisky.

```
<CalculatedDataResultIndicator
  :header="'Průtok vzduchu'"
  :unit="'m³/h'"
  :value="airFlowResult"
/>
```

Kód 52: Komponenta pro zobrazení výsledků kalkulací

Kalkulátory představují významnou podporu pro provozní rozhodování a přispívají k optimalizaci výrobního procesu i bez nutnosti používat externí softwarové nástroje.

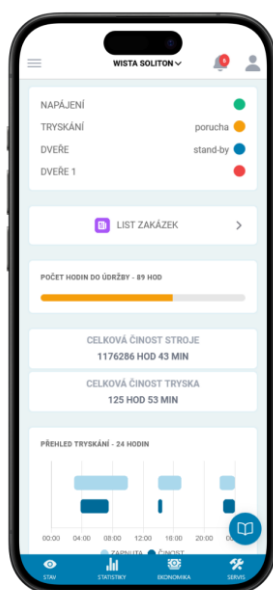
10 FINÁLNÍ VERZE APLIKACE

Tato kapitola představuje výslednou podobu progresivní webové aplikace pro monitoring strojů. Popisuje hlavní funkční části systému, včetně dashboardu, statistik, ekonomiky a provozní knihy.

Jednotlivé obrazovky byly během vývoje upravovány na základě zpětné vazby, s důrazem na přehlednost, funkčnost a uživatelskou přívětivost. Kapitola slouží jako přehled implementovaných řešení a změn oproti původnímu návrhu.

10.1 Stav aplikace

Stav aplikace („Dashboard“) tvoří výchozí obrazovku systému a slouží jako přehledný kontrolní bod nad aktuálním provozním stavem zvoleného stroje. Během vývoje doznal dashboard několika úprav oproti původnímu návrhu. Na základě zpětné vazby a nových požadavků byla doplněna zkratka pro rychlý přístup k seznamu zakázek a také vizuální progress bar indikující stav klíčových částí zařízení.



Obrázek 19: Finální obrazovka „Stavu“

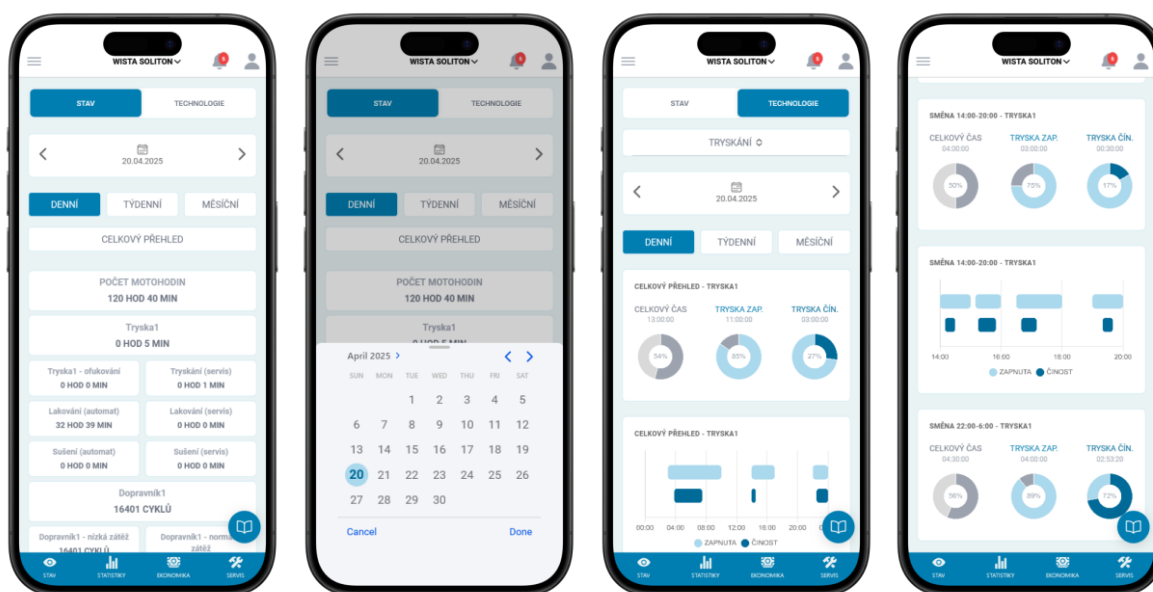
Do obrazovky byly rovněž integrovány nové datové prvky, které odrážejí klíčové provozní ukazatele požadované zadavatelem. Namísto původně zamýšleného zobrazení zakázek formou samostatných karet byl zaveden jednotný seznam v rámci provozní knihy, který je dostupný právě skrze zmíněnou zkratku. Pro zvýšení přehlednosti byly přidány dva Ganttovy diagramy – jeden z diagramů zachycuje detailní průběh tryskání za posledních 24 hodin, zatímco druhý poskytuje celkový přehled provozní činnosti stroje ve stejném

časovém úseku. Tyto vizualizace umožňují uživateli okamžitou orientaci v časovém rozvrhu provozu jednotlivých komponent a napomáhají včasné identifikaci neefektivity či výpadků.

10.2 Statistiky

Sekce Statistiky slouží k podrobnému sledování provozních dat jednotlivých částí stroje v různých časových obdobích. Uživatel zde získává přesný přehled o délce provozu jednotlivých technologických celků, jejich aktivní činnosti a případných nečinnostech. Tato analytická data jsou klíčová pro hodnocení efektivity provozu, identifikaci slabých míst v procesu a optimalizaci plánování výroby.

Z hlediska interakce může uživatel pomocí kalendářového výběru nebo navigačních tlačítek snadno zvolit konkrétní den, případně přepínat mezi denním, týdenním a měsíčním zobrazením. Tato flexibilita umožňuje jak rychlou orientaci v aktuálním výkonu stroje, tak i sledování dlouhodobých trendů a výkyvů.



Obrázek 20: Finální verze obrazovek v sekci „Statistiky“

Oproti původnímu návrhu došlo k zásadní změně ve způsobu zobrazování technologických dat. Původní zobrazení bylo nahrazeno segmentovým přepínačem, který umožňuje přímý výběr konkrétní technologie (např. tryskání). Tato úprava byla realizována především za účelem lepší škálovatelnosti – v budoucnu je díky tomu možné snadno přidat další technologické celky bez nutnosti zásahu do struktury rozhraní. Zároveň se tím výrazně zlepšila uživatelská přívětivost, protože celý proces výběru je nyní intuitivnější a přehlednější.

10.3 Ekonomika

Sekce **Ekonomika** slouží k přehledné a vizuálně srozumitelné analýze nákladů spojených s provozem jednotlivých strojů. Uživatel má možnost definovat jedno nebo dvě časová období, která chce analyzovat nebo porovnat. V případě zadání jednoho intervalu se zobrazí souhrnné náklady za zvolené období, přidáním druhého období se aktivuje možnost jejich srovnání v čase.



Obrázek 21: Finální verze obrazovek v sekci „Ekonomika“

Po potvrzení výběru tlačítkem **Vygenerovat data** se načtou relevantní ekonomické údaje, které jsou následně zobrazeny v podobě sloupcového grafu. Tento graf rozděluje náklady do kategorií jako **lidé**, **abrazivo**, **součástky**, **údržba** a **opravy**, přičemž každá položka je barevně odlišena. Barevná paleta odpovídá vizuálním standardům používaným napříč celou aplikací, zejména v provozní knize, čímž je zajištěna jednotnost a snadná orientace.

Pod hlavním grafem se dále nacházejí koláčové grafy, které znázorňují procentuální rozložení nákladů v jednotlivých obdobích. Tento způsob vizualizace umožňuje rychle identifikovat nejvýznamnější nákladové položky a porovnat jejich zastoupení mezi jednotlivými časovými rámci. Uživatel tak získává ucelený pohled na nákladovou strukturu provozu a může lépe odhalovat oblasti s potenciálem pro optimalizaci či úspory.

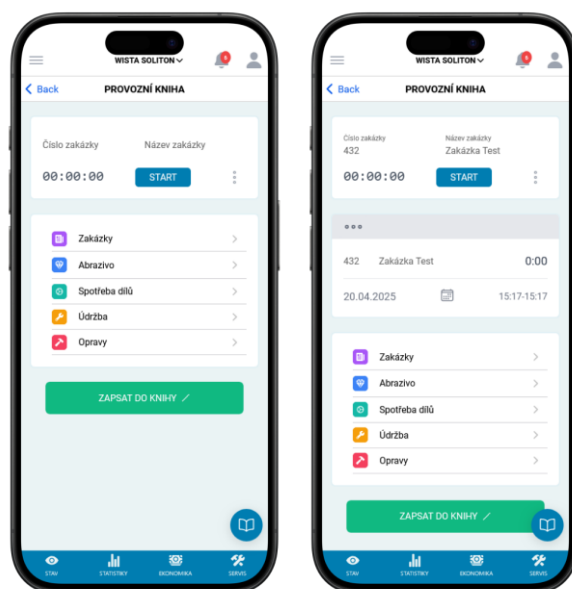
10.3.1 Zobrazení SUMACE

Druhou záložkou této sekce je **Sumace**, která přináší souhrnné informace o celkových nákladech v obou obdobích. Sloupcový graf zobrazuje celkové částky v přímém srovnání,

zatímco pod ním se nachází koláčový graf znázorňující procentuální zastoupení jednotlivých období. Tato kombinace číselných a grafických údajů poskytuje jasný přehled o vývoji nákladů v čase a napomáhá rychle vyhodnotit efektivitu úprav v provozních procesech.

10.4 Provozní kniha

Oproti původnímu návrhu provozní knihy došlo ve finální implementaci k několika podstatným úpravám s cílem zvýšit přehlednost, uživatelskou přívětivost a celkovou ergonomii rozhraní. V návrhové fázi byly jednotlivé sekce jako **Abrazivo**, **Spotřeba dílů**, **Údržba** a **Opravy** součástí jedné dlouhé obrazovky, kde se veškeré záznamy vyplňovaly přímo pod sebou. Tento přístup se však ukázal jako méně praktický při práci na menších displejích nebo při záznamech kde se zapisuje pouze část informací, jako například pouze odpis abraziva.



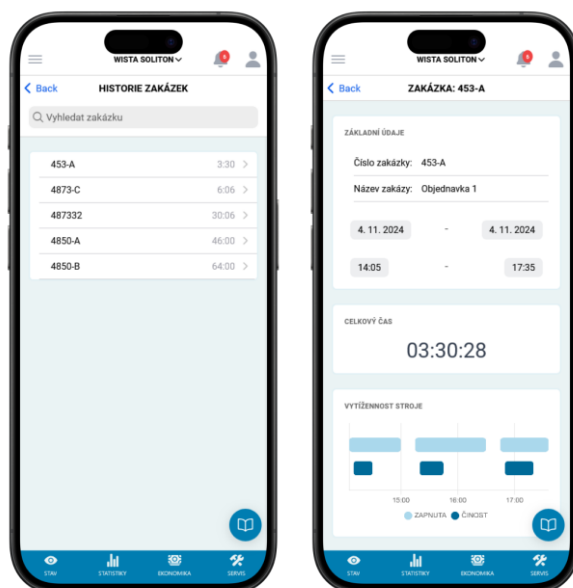
Obrázek 22: Finální verze obrazovek v sekci „Provozní kniha“

Zároveň byly do jednotlivých sekcí přidány barevné ikony, které vizuálně odpovídají stejnému označení jako v sekci Ekonomika. Díky tomu je aplikace vizuálně sjednocená a jednotlivé vstupy lze snadno propojit s jejich analytickým zobrazením.

Změny byly vedeny především snahou o vyšší efektivitu zadávání dat, zvýšení přehlednosti uživatelského rozhraní a sjednocení s ostatními částmi aplikace – jak vizuálně, tak funkčně. Výsledkem je provozní kniha, která nejen eviduje kompletní historii zakázek a souvisejících činností, ale zároveň nabízí komfortní a intuitivní prostředí i pro méně technicky zdatné uživatele.

10.4.1 Seznam zakázek

Součástí provozní knihy je funkce správy zaznamenaných zakázek, která uživatelům umožňuje pohodlně zpětně prohlížet dříve evidované záznamy. Oproti původnímu návrhu, kde byl seznam zakázek umístěn přímo na obrazovce stavu (dashboardu), byl ve finální verzi aplikace přesunut do samostatné sekce provozní knihy. Tento krok vedl ke zvýšení přehlednosti rozhraní a lepší logické struktuře celého systému.



Obrázek 23: Seznam zakázek a jejich detail

Nově je seznam zobrazen v kompaktnější podobě, která umožňuje rychlou orientaci a okamžitý přehled o základních údajích zakázky, jako je její číslo, název a celkový zaznamenaný čas. Vyhledávací pole v horní části umožňuje dynamické filtrování – seznam se automaticky aktualizuje při každém zadání znaku, čímž je možné velmi rychle dohledat konkrétní záznam i v rozsáhlejších souboru dat.

Po kliknutí na libovolnou položku se uživatel přepne do detailního zobrazení konkrétní zakázky. Zde má k dispozici všechny klíčové informace – číslo a název zakázky, datum a časy začátku a konce, celkový čas, ale také vizualizaci vytíženosti stroje v podobě Ganttova diagramu. Uživatel má možnost záznam dále upravovat nebo zcela smazat, čímž je zachována flexibilita při správě dat.

Změna oproti návrhu tak přinesla především lepší oddělení logických celků a celkově zvýšila uživatelskou srozumitelnost a komfort při práci se zakázkami.

ZÁVĚR

Tato diplomová práce se věnovala návrhu a realizaci moderní progresivní webové aplikace určené pro monitoring strojů ve výrobním prostředí. Cílem bylo vytvořit multiplatformní řešení, které propojuje snadné ovládání s robustní funkčností a přehledným zobrazením technických i ekonomických údajů. Aplikace byla navržena s ohledem na potřeby jak technických pracovníků a operátorů, tak i managementu, který může na základě zobrazených dat činit informovaná rozhodnutí.

Výsledná aplikace zahrnuje několik klíčových modulů. Mezi ně patří úvodní dashboard s provozními informacemi, statistiky pro sledování zatížení technologií v čase, přehled ekonomických nákladů rozdělený dle kategorií a období, a rozsáhlá provozní kniha pro evidenci zakázek, spotřeb, oprav a dalších vstupů. Důležitou součástí jsou rovněž technické kalkulátory umožňující rychlé výpočty klíčových parametrů bez nutnosti použití externích nástrojů.

V průběhu vývoje docházelo k postupnému zpřesňování požadavků a vylepšování uživatelského rozhraní. Některé části byly výrazně přepracovány oproti původnímu návrhu – například přesun seznamu zakázek z dashboardu do provozní knihy, nebo rozdělení nákladových kategorií do samostatných, přehledněji strukturovaných sekcí. Tyto změny byly realizovány s cílem zvýšit uživatelský komfort a přispět k lepší orientaci v systému.

Z technického hlediska aplikace staví na modulární architektuře s důsledným oddělením komponent, centrální správou stavu prostřednictvím knihovny *Pinia* a přehledným řízením datových toků. Pro komunikaci s back-endem využívá *REST API*, uživatelská autentizace je řešena pomocí *JWT* tokenů a systém umožňuje bezpečné řízení přístupu k chráněným zdrojům.

Práce dále ukazuje, že i s využitím simulovaného back-endu a *open-source* nástrojů je možné navrhnout a realizovat plnohodnotnou aplikaci, která má praktický přínos pro praxi. Do budoucna je možné aplikaci rozšířit o pokročilejší funkce, jako je *role-based access control*, live synchronizace dat přes *WebSocket*, nebo propojení s průmyslovými komunikačními protokoly jako *OPC UA* či *MQTT*.

SEZNAM POUŽITÉ LITERATURY

- [1] ROLO, Pedro. *A JavaScript Ecosystem Overview*. Online. Imaginary Cloud. Dostupné z: <https://www.imaginarycloud.com/blog/a-javascript-ecosystem-overview>. [cit. 2025-04-22].
- [2] NEAGOIE, Andrei. *Angular vs React vs Vue: The Best Framework for 2025 is....*. Online. ZTM. 2018, 12.02.2025. Dostupné z: <https://zerotomastery.io/blog/angular-vs-react-vs-vue/>. [cit. 2025-05-13].
- [3] CHARGUI, Ines. *The 2024 State of JavaScript Survey: Who's Taking the Lead?* Online. IJS. 05.02.2025. Dostupné z: <https://javascript-conference.com/blog/state-of-javascript-ecosystem-2024/>. [cit. 2025-05-13].
- [4] SAID, Mostafa. *Vite vs. Webpack: A Head-to-Head Comparison Author*. Online. Kinsta. 20.03.2025. Dostupné z: <https://kinsta.com/blog/vite-vs-webpack/>. [cit. 2025-05-13].
- [5] PARODY, Lizz. *How Node.js Works: A Comprehensive Guide in 2025*. Online. Nodesource. 28.01.2025. Dostupné z: <https://nodesource.com/blog/how-nodejs-works>. [cit. 2025-05-13].
- [6] *Top 9 JavaScript Testing Frameworks*. Online. 13.03.2025. Dostupné z: <https://www.browserstack.com/guide/top-javascript-testing-frameworks>. [cit. 2025-05-13].
- [7] WATSON, David. *JavaScript and TypeScript Trends 2024: Insights From the Developer Ecosystem Survey*. Online. *JetBrains Blog*. Dostupné z: <https://blog.jetbrains.com/webstorm/2024/02/js-and-ts-trends-2024/>. [cit. 2025-04-25].
- [8] RAMATIS, Cherry. *What is JSDoc and why you may not need typescript for your next project?* Online. *Dev*. Dostupné z: <https://dev.to/cherryramatis/what-is-jsdoc-and-why-you-may-not-need-typescript-for-your-next-project-54n1>. [cit. 2025-04-25].
- [9] REILLY, John. *TypeScript vs. JSDoc JavaScript for static type checking*. Online. *LogRocket*. Dostupné z: <https://blog.logrocket.com/typescript-vs-jsdoc-javascript/>. [cit. 2025-04-25].
- [10] *Introduction*. Online. *Vue.js*. Dostupné z: <https://vuejs.org/guide/introduction>. [cit. 2025-04-25].

- [11] *What is Angular?* Online. Angular. Dostupné z: <https://angular.dev/overview>. [cit. 2025-04-25].
- [12] *React.js overview.* Online. Sanito. 23.09.2024. Dostupné z: <https://www.sanity.io/glossary/react-js>. [cit. 2025-04-25].
- [13] *About Web Components.* Online. Medium. 13.03.2023. Dostupné z: <https://eisenbergeffect.medium.com/about-web-components-7b2a3ed67a78>. [cit. 2025-04-25].
- [14] CAMEROON, Luis. *Micro-frontends with Module Federation: Building scalable and modular web applications.* Online. Luxoft. 24.01.2024. Dostupné z: <https://www.luxoft.com/blog/micro-frontends-with-module-federation>. [cit. 2025-04-25].
- [15] *What are Micro Frontends? Definition, Uses, Architecture.* Online. GeeksforGeeks. 8.05.2024. Dostupné z: <https://www.geeksforgeeks.org/what-are-micro-frontends-definition-uses-architecture/>. [cit. 2025-04-25].
- [16] GOYAL, Charu. *Implementing Micro-frontends with Vite Federation : A Practical Guide.* Online. Medium. 8.05.2024. Dostupné z: <https://medium.com/%40charu.sharma517/implementing-micro-frontends-with-vite-federation-a-practical-guide-333351575861>. [cit. 2025-04-25].
- [17] BRANDES, Eric. *HTTP/3 is Fast!.* Online. Request Metrics. 19.02.2025. Dostupné z: <https://requestmetrics.com/web-performance/http3-is-fast/>. [cit. 2025-04-25].
- [18] *Cranking Brotli up to 11 with Cloudflare Pro and 11ty.* Online. Nooshu. 05.01.2025. Dostupné z: <https://nooshu.com/blog/2025/01/05/cranking-brotli-up-to-11-with-cloudflare-pro-and-11ty/>. [cit. 2025-04-25].
- [19] *CDN Innovations: From HTTP/3 Support to Advanced Compression.* Online. Blazing CDN. 14.08.2024. Dostupné z: <https://blog.blazingcdn.com/en-us/cdn-innovations-from-http3-support-to-advanced-compression/>. [cit. 2025-04-25].
- [20] *What is a progressive web app?* Online. MDN. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/What_is_a_progressive_web_app. [cit. 2025-04-29].

- [21] *Embracing the Evolution: The Rise of Progressive Web Apps (PWAs)*. Online. GeeksforGeeks. 02.04.2024. Dostupné z: <https://www.geeksforgeeks.org/embracing-the-evolution-the-rise-of-progressive-web-apps-pwas/>. [cit. 2025-04-29].
- [22] ZAHIN FUAD, Khondaker. *Progressive Web App (PWA) Development Ultimate Guide*. Online. Riseup Labs. 14.04.2025. Dostupné z: <https://riseuplabs.com/pwa-development-ultimate-guide/>. [cit. 2025-04-29].
- [23] *Responsive Design vs. Adaptive Design: What's the Best Choice for Designers?* Online. Uxpin. Dostupné z: <https://www.uxpin.com/studio/blog/responsive-vs-adaptive-design-whats-best-choice-designers/>. [cit. 2025-04-29].
- [24] *App Shell Architecture & Service Workers: A Beginner's Guide*. Online. Medium. 09.02.2025. Dostupné z: <https://medium.com/%40pvprasanth474/app-shell-architecture-service-workers-a-beginners-guide-6e444964c936>. [cit. 2025-04-29].
- [25] *Service workers and the application shell model*. Online. Chrome for developers. Dostupné z: <https://developer.chrome.com/docs/workbox/app-shell-model>. [cit. 2025-04-29].
- [26] *Making PWAs installable*. Online. Mdn. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Making_PWAs_installable. [cit. 2025-04-29].
- [27] PATHAK, Abhinav. *How to Add Web Push Notifications to your PWA on iOS and Android (The Easy Way)*. Online. PushAlert. 27.12.2023. Dostupné z: <https://pushalert.co/blog/pwa-push-notifications-ios-android/>. [cit. 2025-04-29].
- [28] *Website security*. Online. Mdn. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Website_security. [cit. 2025-04-29].
- [29] WILSON, Sean. *HTTPS - Web Security Best Practices*. Online. Checkbot. Dostupné z: <https://www.checkbot.io/guide/security/https/>. [cit. 2025-04-29].
- [30] *Content Security Policy (CSP) implementation*. Online. Mdn. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/CSP. [cit. 2025-04-30].
- [31] *What is CORS?* Online. Aws. Dostupné z: <https://aws.amazon.com/what-is/cross-origin-resource-sharing/>. [cit. 2025-04-30].

- [32] *Cross-Origin Resource Sharing (CORS)*. Online. Mdn. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>. [cit. 2025-04-30].
- [33] RAMOS, Marcia. *Kotlin vs Java for Mobile and Web Development*. Online. Kinsta. 12.03.2024. Dostupné z: <https://kinsta.com/blog/kotlin-vs-java/>. [cit. 2025-05-02].
- [34] POPKO, Aleksander. *Objective-C vs Swift: iOS Comparison [2024 Update]*. Online. Netguru. 27.11.2024. Dostupné z: <https://www.netguru.com/blog/objective-c-vs-swift>. [cit. 2025-05-02].
- [35] SMITH, Mark. *Why Swift is Better Than Objective-C for iOS App Development*. Online. Medium. 12.09.2023 Dostupné z: <https://zrix.medium.com/swift-vs-objective-c-why-swift-reigns-supreme-in-ios-app-development-56682317de1a>. [cit. 2025-05-02].
- [36] ASODARIYA, Dhaval a ASODARIYA, Parth. *Kotlin Multiplatform vs Swift: A Simple Side-by-Side Comparison*. Online. DhiWise. 15.11.2024. Dostupné z: <https://www.dhiwise.com/post/kotlin-multiplatform-vs-swift-a-simple-side-by-side-comparison>. [cit. 2025-05-02].
- [37] *What is Flutter?* Online. Aws. Dostupné z: <https://aws.amazon.com/what-is/flutter/>. [cit. 2025-05-02].
- [38] BUDZIŃSKI, Maciej. *What Is React Native? Complex Guide for 2024*. Online. Netguru. Dostupné z: <https://www.netguru.com/glossary/react-native>. [cit. 2025-05-02].
- [39] *Introducing Lynx: How TikTok's New Framework Compares to React Native*. Online. Medium. 17.03.2025. Dostupné z: <https://medium.com/%40mostsignificant/introducing-lynx-how-tiktoks-new-framework-compares-to-react-native-580a52d3462c>. [cit. 2025-05-02].
- [40] *Developing Hybrid Mobile Applications with Cordova: A Step-by-Step Guide*. Online. Medium. 01.07.2023. Dostupné z: <https://medium.com/@phamtuanchip/developing-hybrid-mobile-applications-with-cordova-a-step-by-step-guide-4ee0d92520ea>. [cit. 2025-05-08].
- [41] *What Is Ionic And How Does It Work?* Online. Appstronauts. 20.10.2022. Dostupné z: <https://appstronauts.co/blog/what-is-ionic-and-how-does-it-work/>. [cit. 2025-05-08].

- [42] *Hybrid mobile development with Ionic Framework*. Online. Cleverti. 17.07.2020. Dostupné z: <https://www.cleverti.com/blog/mobile-development/hybrid-mobile-development-with-ionic-framework/>. [cit. 2025-05-08].
- [43] AGGARWAL, Shanal. *Native vs. Hybrid App Development Cost Comparison*. Online. Tectahead. 05.07.2024. Dostupné z: <https://www.techaheadcorp.com/blog/native-vs-hybrid-app-development-cost-comparison/>. [cit. 2025-05-08].
- [44] TUAN, Joe. *How Much Does it Cost to Build an App in 2025: The Ultimate Guide*. Online. Topflight. 10.02.2025. Dostupné z: <https://topflightapps.com/ideas/app-development-costs/>. [cit. 2025-05-08].
- [45] *Hybrid vs Cross Platform vs Native Mobile App Development: How to Choose?* Online. InApp. Dostupné z: <https://inapp.com/blog/native-vs-hybrid-vs-cross-platform-app-development-how-to-choose/>. [cit. 2025-05-08].
- [46] SADOWSKI, Matt a NABORSHCHIKOV, Stanislav. *Cross-platform vs native app development: Ultimate Comparison*. Online. InApp. 2023, 12.03.2024. Dostupné z: <https://themobilereality.com/blog/cross-platform-vs-native-app-development>. [cit. 2025-05-08].
- [47] NIELSEN, Jakob. *10 Usability Heuristics for User Interface Design*. Online. Nngroup. 2024, 12.03.2024. Dostupné z: <https://www.nngroup.com/articles/ten-usability-heuristics/>. [cit. 2025-05-08].
- [48] WONG, Euphemia. *User Interface Design Guidelines: 10 Rules of Thumb*. Online. Interaction. Dostupné z: <https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb?srsId=AfmBOOrGHWxbSFqKLiLe49KXXIUhIB4UzVaL0yUZzgoamvtXqkgB4VWv&>. [cit. 2025-05-08].
- [49] *The 6 key UX design principles to follow*. Online. Falmouth University. Dostupné z: <https://www.falmouth.ac.uk/news/6-key-ux-design-principles-follow>. [cit. 2025-05-09].
- [50] HOLST, Christian. *The Best Practices and Key Principles of UX Design*. Online. Baymard Institute. Dostupné z: <https://baymard.com/learn/ux-design-principles>. [cit. 2025-05-09].

- [51] TOMBOC, Kai. *UX design principles*. Online. Lyssna. 04.05.2025. Dostupné z: <https://www.lyssna.com/blog/ux-design-principles/>. [cit. 2025-05-09].
- [52] *Functional vs. Non Functional Requirements*. Online. Geeks For Geeks. 2024. Dostupné z: <https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/>. [cit. 2024-12-23].
- [53] TAYLOR, Eliza. *What is Client-Server Architecture? Explained in Detail*. Online. The knowledge academy. 2024. Dostupné z: <https://www.theknowledgeacademy.com/blog/client-server-architecture/>. [cit. 2024-12-27].
- [54] *Rozdíl mezi klientským serverem a peer to peer*. Online. Differkinome. Dostupné z: <https://cs.differkinome.com/articles/technology/difference-between-client-server-and-peer-to-peer-2.html>. [cit. 2024-12-27].
- [55] *Introduction to web APIs*. Online. MOZILLA CORPORATION. MDN Web Docs. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction. [cit. 2024-12-27].
- [56] BEN, Jeremy. *Json-server-auth*. Online. Npm. Dostupné z: <https://www.npmjs.com/package/json-server-auth>. [cit. 2025-05-17].

SEZNAM OBRÁZKŮ

Obrázek 1: Micro-frontends CSR vs. SSR [16].....	28
Obrázek 2: Porovnání HTTP/2 a HTTP/3 v USA [17].....	30
Obrázek 3: Primární Barva (od nejsvětlejší po nejtmavší).....	65
Obrázek 4: Návrh přihlašovací obrazovky	66
Obrázek 5: Návrh navigace (Spodní lišta).....	67
Obrázek 6: Návrh navigace (horní lišta).....	67
Obrázek 7: Návrh navigace (rozbalovací lišta).....	68
Obrázek 8: Návrh oznámení	68
Obrázek 9: Návrh listu oznámení	69
Obrázek 10: Návrh statusu oznámení	69
Obrázek 11: Návrh detailu oznámení	70
Obrázek 12: Návrh úvodní obrazovky (Status stroje)	71
Obrázek 13: Návrh editace času	72
Obrázek 14: Návrh obrazovky pro statistiky stroje	73
Obrázek 15: Návrh obrazovky pro provozní knihu	74
Obrázek 16: Návrh obrazovky pro export dat	75
Obrázek 17: Návrh obrazovky pro manuály.....	76
Obrázek 18: Obrazovka kalkulačtorů	94
Obrázek 19: Finální obrazovka „Stavu“	97
Obrázek 20: Finální verze obrazovek v sekci „Statistiky“	98
Obrázek 21: Finální verze obrazovek v sekci „Ekonomika“	99
Obrázek 22: Finální verze obrazovek v sekci „Provozní kniha“	100
Obrázek 23: Seznam zakázek a jejich detail.....	101

SEZNAM TABULEK

Tabulka 1: Funcionální požadavky	61
Tabulka 2: Nefunkcionální požadavky	62

SEZNAM KÓDŮ

Kód 1: Ukázka typování v jazyce TypeScript	23
Kód 2: Ukázka typování pomocí JSDoc	23
Kód 3: Ukázka JSON Web App Manifest	37
Kód 4: Ukázka JSON formátu	64
Kód 5: Souborová struktura	77
Kód 6: Ukázka API volání	78
Kód 7: Ukázka definovaného objektu	78
Kód 8: Příklad Pinia store	79
Kód 9: Statický routing na stránku „EconomyPage“	80
Kód 10: Dynamický routing detailu zakázky	81
Kód 11: Navigace pomocí komponenty <ion-tab-button>	81
Kód 12: Navigace pomocí JavaScriptu	81
Kód 13: Navigace pomocí události	81
Kód 14: Import komponenty ProgressBarComponent	82
Kód 15: Ukázka použití komponent v šabloně	82
Kód 16: Ukázka předání dat do komponenty	83
Kód 17: Definování props v komponentě	83
Kód 18: Rozšířená podoba props	83
Kód 19: Ukázka použití props v šabloně	83
Kód 20: Ukázka použití props v šabloně 2	84
Kód 21: Ukázka použití props v JavaScript části	84
Kód 22: Definování emitovatelných událostí	84
Kód 23: Emitování události	84
Kód 24: Zachytávání emitu v rodičovské komponentě	84
Kód 25: Ukázka prop-drilling	85
Kód 26: Ukázka Provide / Inject	85
Kód 27: Ukázka Pinia Store	86
Kód 28: Inicializace store v kódu	86
Kód 29: Změna hodnoty v Pinia store pomocí v-model	86
Kód 30: Zobrazení hodnoty z Pinia store	86
Kód 31: Ukázka JSON struktury	87
Kód 32: Ukázka GET requestu	87
Kód 33: Ukázka GET requestu s dynamickým parametrem	88
Kód 34: Skript pro spuštění JSON serveru s auth	88

Kód 35: JSON objekt v DummyData.json.....	88
Kód 36: AccessToken	89
Kód 37: Vložení accessTokenu do localStorage.....	89
Kód 38: Login request.....	89
Kód 39: Metoda pro odhlášení uživatele	90
Kód 40: Metoda hydrate pro vložení tokenu a informací z aktuální session	90
Kód 41: Funkce pro obnovení aktuální session z localStorage	90
Kód 42: Funkce pro validace tokenu	91
Kód 43: Pole requiresAuth u routingu	91
Kód 44: Kontrola platného tokenu při každé změně aktuální URL.....	91
Kód 45: Chráněný fetch, který kontroluje, zda token je validní	92
Kód 46: Formulář pro přihlášení uživatele	93
Kód 47: Ukázka handler funkce.....	94
Kód 48: Kód pro výpočet Bernoulliho rovnice.....	95
Kód 49: Výpočet tlaku na trysce.....	95
Kód 50: Výpočet pŕotoku trysky.....	95
Kód 51: Input pro kalkulátor.....	95
Kód 52: Komponenta pro zobrazení výsledků kalkulací	96

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

PWA	<i>Progressive Web Application</i> (Progresivní webová aplikace)
UX	<i>User Experience</i> (Uživatelská zkušenost)
API	<i>Application Programming Interface</i> (Aplikační programové rozhraní)
JWT	<i>JSON Web Token</i> (Token pro autentizaci na bázi JSON)
UI	<i>User Interface</i> (Uživatelské rozhraní)
HTTP	<i>HyperText Transfer Protocol</i> (Protokol pro přenos hypertextu)
HTTPS	<i>HyperText Transfer Protocol Secure</i> (Šifrovaný protokol HTTP pomocí TLS/SSL)
P2P	<i>Peer-to-Peer</i> (Síťový model rovný s rovným)
GDPR	<i>General Data Protection Regulation</i> (Obecné nařízení o ochraně osobních údajů)
FTP	<i>File Transfer Protocol</i> (Protokol pro přenos souborů)
PDF	<i>Portable Document Format</i> (Přenositelný dokumentový formát)
JSON	<i>JavaScript Object Notation</i> (Formát pro výměnu dat)
CSV	<i>Comma-Separated Values</i> (Textový formát tabulkových dat oddělených čárkami)
WCAG	<i>Web Content Accessibility Guidelines</i>
iOS	<i>iPhone Operating System</i> (Operační systém mobilních zařízení Apple)
JS	<i>JavaScript</i>
CSS	<i>Cascading Style Sheets</i> (Kaskádové styly pro vzhled webových stránek)
ARC	<i>Automatic Reference Counting</i>
CSP	<i>Content Security Policy</i>
CORS	<i>Cross-Origin Resource Sharing</i> (Sdílení prostředků mezi různými doménami)
JSONP	<i>JSON with Padding</i>
XSS	<i>Cross-Site Scripting</i> (Typ útoku, kdy je do webu vložen škodlivý skript)
DoS	<i>Denial of Service</i> (Útok vedoucí k nedostupnosti služby)
SEO	<i>Search Engine Optimization</i> (Optimalizace pro vyhledávače)
JSDoc	<i>JavaScript Documentation</i>

