



**FAKULTA  
INFORMAČNÍCH  
TECHNOLOGIÍ  
ČVUT V PRAZE**

## ZADÁNÍ DIPLOMOVÉ PRÁCE

**Název:** Inteligentní zabezpečovací systém garáže: Nadřazený systém  
**Student:** Bc. Ondřej Červenka  
**Vedoucí:** Ing. Martin Daňhel  
**Studijní program:** Informatika  
**Studijní obor:** Návrh a programování vestavných systémů  
**Katedra:** Katedra číslicového návrhu  
**Platnost zadání:** Do konce letního semestru 2018/19

### Pokyny pro vypracování

Dle analýzy vyberte vhodnou platformu (Raspberry Pi, ZYNQ apod.) a navrhnete a implementujete systém (ve formě aplikace/bitstreamu) pro správu jednotlivých garáží.

Navržený systém na základě příchozích událostí vyhodnotí situaci a zareaguje vhodnou akcí (např. jedna z garáží se nebude hlásit, v jedné z garáží čidla zahlásí, že hoří, případně neautorizovaný vstup apod.).

Systém bude umožňovat správu podřízených systémů jednotlivých garáží (ty nejsou součástí práce). Pro komunikaci (eth/wifi) mezi jednotlivými garážemi a navrhovaným systémem navrhnete rozhraní na základě rešeršní analýzy.

Navrhnete webové rozhraní pro správu garáží, které bude umožňovat zobrazit současný stav jednotlivých garáží (otevřeno, zamčeno, obsazeno, volno), pokusů o útok včetně historie událostí. Do webového rozhraní je umožněn pouze autorizovaný přístup.

Systém navrhnete s ohledem na bezpečnost, spolehlivost a rozšiřitelnost.

Celé zařízení otestujete.

### Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.  
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.  
děkan

V Praze dne 31. ledna 2018



ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE  
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
KATEDRA ČÍSLICOVÉHO NÁVRHU



Diplomová práce

## Nadřazený systém pro správu garáže

*Bc. Ondřej Červenka*

Vedoucí práce: Ing. Martin Daňhel

2. května 2018



---

## Poděkování

Děkuji panu Ing. Martinu Daňhelovi za čas, který mi věnoval a zejména za cenné rady a odborné vedení mé diplomové práce.



---

## Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. května 2018

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2018 Ondřej Červenka. Všechna práva vyhrazena.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### **Odkaz na tuto práci**

Červenka, Ondřej. *Nadřazený systém pro správu garáže*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.



---

## Abstrakt

Tato diplomová práce se zabývá tvorbou zabezpečovacího systému pro správu garáží. Konkrétně jde o návrh a implementaci nadřazeného systému, který sbírá a zpracovává data zaslaná podřízenými systémy v jednotlivých garážích.

Výsledná aplikace komunikuje s podřízenými systémy pomocí protokolu HTTPS a umožňuje správu garáží přes webové rozhraní. Celý nadřazený systém je možné provozovat na Raspberry Pi.

**Klíčová slova** bezpečnost, webová aplikace, Raspberry Pi

---

## Abstract

This thesis deals with design and implementation of garage security and monitoring system, which collects data from subsystems placed in individual garages.

Result is an application that communicates with subsystems via HTTPS and allows garage monitoring and management through web interface. Application can be deployed on Raspberry Pi.

**Keywords** security, web application, Raspberry Pi



---

# Obsah

Úvod	1
<b>1 Teoretický úvod</b>	<b>3</b>
1.1 Názvosloví použité v práci . . . . .	3
1.2 Návrhový vzor MVC . . . . .	3
1.3 Protokol HTTP . . . . .	4
1.4 Certifikát . . . . .	5
1.5 CSRF – <i>Cross-site request forgery</i> . . . . .	5
1.6 <i>Attack surface</i> . . . . .	6
1.7 <i>Man-in-the-middle</i> útok . . . . .	6
1.8 <i>Hashování</i> . . . . .	6
<b>2 Analýza</b>	<b>9</b>
2.1 Struktura systému . . . . .	10
2.2 Výběr komunikačního protokolu . . . . .	12
2.3 Ukládání dat . . . . .	18
2.4 Zasílání upozornění . . . . .	19
2.5 Programovací jazyk pro tvorbu systému . . . . .	21
2.6 Výběr platformy . . . . .	22
<b>3 Návrh</b>	<b>29</b>
3.1 Návrhový vzor MVC . . . . .	29
3.2 Komponenta <i>model</i> . . . . .	30
3.3 Komponenta <i>view</i> . . . . .	35
3.4 Komponenta <i>controller</i> . . . . .	39
3.5 Autentizace uživatele . . . . .	42
<b>4 Implementace</b>	<b>43</b>
4.1 Struktura aplikace . . . . .	43
4.2 Implementace <i>modelu</i> . . . . .	43

4.3	Implementace <i>view</i> . . . . .	47
4.4	Implementace <i>controlleru</i> . . . . .	48
4.5	Autentizace a přihlášení uživatele . . . . .	50
4.6	Konfigurace aplikace . . . . .	50
<b>5</b>	<b>Testování</b>	<b>53</b>
5.1	Automatizované testy . . . . .	53
5.2	Testovací nasazení aplikace . . . . .	57
5.3	Simulátor podřízeného systému . . . . .	59
5.4	Test odezvy nadřazeného systému . . . . .	59
	<b>Závěr</b>	<b>61</b>
	<b>Literatura</b>	<b>65</b>
<b>A</b>	<b>Nasazení</b>	<b>73</b>
A.1	Konfigurace aplikace nadřazeného systému . . . . .	73
A.2	Konfigurace Apache2 . . . . .	74
A.3	Konfigurace Gammu SMSD . . . . .	76
<b>B</b>	<b>Uživatelská příručka</b>	<b>77</b>
B.1	První přihlášení . . . . .	77
B.2	Hlavní stránka . . . . .	78
B.3	Stránka garáže . . . . .	80
B.4	Uživatelské nastavení . . . . .	81
<b>C</b>	<b>Seznam použitých zkratk</b>	<b>83</b>
<b>D</b>	<b>Obsah přiloženého CD</b>	<b>85</b>

---

## Seznam obrázků

1.1	<i>Man-in-the-middle</i> útok . . . . .	7
2.1	Základní struktura systému . . . . .	11
2.2	Příklad struktury protokolu MQTT . . . . .	15
2.3	GSM modul Huawei E3131 . . . . .	21
2.4	Raspberry Pi 3 . . . . .	23
2.5	Přípravek Zybo Zynq-7000 . . . . .	24
2.6	Blokové schéma využití IP jádra Xilibus . . . . .	25
2.7	Struktura systému provozovaného na virtuálním serveru . . . . .	28
3.1	Struktura MVC aplikace . . . . .	30
3.2	Diagram vyhodnocování stavu garáže . . . . .	34
3.3	Vrchní úroveň případů užití webového rozhraní . . . . .	36
3.4	Případy užití správy garáží . . . . .	37
3.5	Případy užití zobrazení událostí . . . . .	38
3.6	Případy užití uživatelského nastavení . . . . .	38
5.1	Výsledky testu HTTPS konfigurace . . . . .	58
A.1	Softwarové blokové schéma nadřazeného systému . . . . .	74
B.1	Nadřazený systém (zařízení) . . . . .	77
B.2	Přihlašovací formulář webového rozhraní . . . . .	78
B.3	Výzva ke změně hesla . . . . .	78
B.4	Hlavní stránka aplikace . . . . .	79
B.5	Stránka garáže . . . . .	80



---

# Seznam tabulek

2.1	Srovnání platforem Raspberry Pi 3 a Zybo Zynq-7000 . . . . .	26
-----	--	----





---

# Úvod

Tato práce se zabývá zabezpečením garážového komplexu. V něm je potřeba jednotlivé garáže zajistit jednak proti nedovolenému vniknutí, a také proti nebezpečí požáru. Je tedy nutné sledovat události indikující vznik těchto hrozeb (například detekce kouře či pohybu). K tomu slouží zařízení umístěná v garážích, která sledují stav svého okolí pomocí čidel a jsou schopna detekovat nebezpečí.

Tato zařízení nepracují samostatně, ale odesílají data o zaznamenaných událostech centrální jednotce – nadřazenému systému<sup>1</sup>. Tento nadřazený systém zpracovává příchozí informace a varuje uživatele o potenciálních hrozbách.

Kromě toho také vytváří historii zachycených událostí. Ta může mít informativní hodnotu pro provozovatele komplexu (například údaje o vytíženosti jednotlivých garáží), nebo také posloužit při policejním vyšetřování (časy otevření dveří či zacyhčení pohybu).

Cílem práce je vytvořit a otestovat tento nadřazený systém. Výsledná aplikace bude komunikovat pomocí WiFi či Ethernetu s podřízenými systémy (monitorovacími zařízeními v garážích). Na základě získaných dat pak bude udržován stav jednotlivých garáží a vytvářena historie událostí.

Systém bude poskytovat webové rozhraní pro administraci. V něm bude možné přidávat a odebírat podřízené systémy, zobrazovat jejich stav a zaznamenané události.

Vzhledem k povaze zadání je nutné systém navrhnout s ohledem na zabezpečení přenášených informací před odposloucháváním či manipulací. Též je nutné autorizovat uživatele přistupující do webového rozhraní.

Dalším důležitým požadavkem je snadná rozšiřitelnost o nové funkce. Systém by mělo být možné v budoucnu doplnit o možnost správy rozdílných

---

<sup>1</sup>V textu práce budu dále používat označení „nadřazený systém“ pro tuto centrální jednotku a „podřízený systém“ pro monitorovací zařízení, která jsou umístěna v jednotlivých garážích.

podřízených systémů (například subsystémy pro sledování skladových zásob) či integraci s mobilní aplikací. Bude tedy potřeba navrhnout vhodné komunikační rozhraní pro předávání informací mezi systémem a jeho klienty.

V práci se chci zaměřit na tvorbu aplikace na jedné konkrétní hardwarové platformě, jako je například jednodeskový počítač Raspberry Pi. Aplikace spolu s touto platformou by pak měla tvořit kompletní zařízení, které bude možné po základní konfiguraci (připojení do WiFi sítě, nastavení hesla) hned nasadit.

Výsledné řešení by však mělo být dostatečně nezávislé na zvolené platformě. Tudíž by neměl být problém spustit systém například na osobním počítači či virtuálním serveru.

V analytické části (2) práce tedy přiblížím proces výběru vhodné platformy, komunikačního protokolu a dalších prvků systému. Také stručně popíšu podřízený systém (který spravuje garážová čidla), se kterým budu dále pracovat. Další části mapují návrh (3) systému na základě této analýzy, jeho implementaci (4) a testování (5). Nasazení na zvoleném hardwaru je popsáno v příloze A. Příloha B obsahuje stručnou uživatelskou příručku.

# Teoretický úvod

## 1.1 Názvosloví použité v práci

- **Nadřazený systém** – zařízení, jehož tvorba je cílem práce, je v textu označováno jako nadřazený systém. Podrobná specifikace nadřazeného systému je popsána v úvodu kapitoly 2.
- **Podřízený systém** – podřízené systémy jsou monitorovací zařízení, která jsou umístěná v jednotlivých garážích. Zařízení neoperují samostatně, ale zasílají data nadřazenému systému. Podřízený systém je blíže popsán v sekci 2.1.1.
- **Uživatel** – uživatelem se v textu práce myslí osoba spravující nadřazený systém a přistupující do jeho webového rozhraní. Jde tedy například o majitele garáží nebo zaměstnance pověřeného jejich správou.
- **Klíč** – náhodně generovaná data (například ve formě textového řetězce), která slouží k ověření totožnosti. Znalost klíče umožní přístup k šifrovaným datům nebo jinak zakázaným operacím. V této práci se hovoří většinou o API klíčích, kterými se prokazují podřízené systémy při komunikaci s nadřazeným systémem.
- **Webové rozhraní** – webovým rozhraním jsou v textu práce myšleny webové stránky, pomocí kterých je možné spravovat nadřazený systém.

## 1.2 Návrhový vzor MVC

MVC (*model-view-controller*) je návrhový vzor, který strukturuje aplikaci do tří částí [1]:

- *Model* reprezentuje data v aplikaci a operace, které s nimi lze provádět. Důležitá vlastnost *modelu* je nezávislost na *view* a *controlleru*. Díky tomu je možné stejný *model* použít například pro webovou a nativní aplikaci.

- *View* je zobrazení *modelu* aplikace (například pomocí webových stránek).
- *Controller* zpracovává uživatelský vstup a na jeho základě přenáší data mezi *modelem* a *view*.

Návrhový vzor MVC je použit při implementaci webové aplikace nadřazeného systému (bližší informace o použití tohoto vzoru jsou v sekci 3.1).

### 1.3 Protokol HTTP

Při implementaci nadřazeného systému byl rozsáhle využit protokol HTTP (na tomto protokolu je postaveno uživatelské rozhraní a komunikace s podřízenými systémy). Tato sekce popisuje s ním spojené principy, které jsou zmiňovány v dalším textu práce.

#### 1.3.1 Model *client/server*

Protokol používá model *client/server*. Klienti zasílají HTTP serveru požadavky, které server zpracovává a vytváří příslušné odpovědi [2]. Spojení je tedy iniciováno na straně klienta a server pouze odpovídá.

#### 1.3.2 Požadavek

Požadavek (*request*) zasílaný klientem tvoří následující části [2]:

- Metoda a URL zasláního požadavku. V této práci jsou použity především metody **get** a **post**. Metoda **get** slouží ke získávání dat (například HTML stránek) ze serveru. Metoda **post** je určena k zasílání nových informací serveru.
- Záhlaví obsahující jednotlivé hlavičky požadavku. Tato část je v práci využita pro zaslání API klíče podřízeného systému, viz sekci 3.4.1.
- Volitelná přenášená data (například obsah HTML formuláře).

Na požadavek server odpovídá zasláním požadovaných dat (pokud jsou dostupná) a návratovým kódem, který indikuje úspěšnost dotazu [2].

#### 1.3.3 Rozšíření HTTPS

Samotný protokol HTTP neimplementuje žádné šifrování, proto vzniklo jeho rozšíření zabezpečené protokolem SSL či TLS<sup>2</sup>, nazvané HTTPS [4].

Protokol TLS umožňuje autentizaci serveru pomocí certifikátu (viz sekci 1.4) a domluvu serveru a klienta na společném klíči [4]. Tímto klíčem je pak šifrována veškerá komunikace mezi klientem a serverem [4].

---

<sup>2</sup>Dále je pro jednoduchost zmiňován jen protokol TLS podle normy RFC-5246 [3].

## 1.4 Certifikát

Certifikát je obecně struktura sloužící k ověření totožnosti účastníka komunikace a distribuci jeho veřejného klíče [4]. V této práci se certifikátem myslí certifikát podle normy X.509 [5], který je používán protokolem TLS [3].

### 1.4.1 Certifikační autorita

Certifikační autorita je subjekt, který se zabývá vydáváním certifikátů. Teoreticky může certifikát vydat kdokoli, k ověření totožnosti serveru lze však použít pouze certifikát vydaný důvěryhodnou certifikační autoritou.

U takového certifikátu lze ověřit jeho pravost a tím zaručit, že nebyl podvržen (útok založený na podvrženém certifikátu popisuje sekce 1.7) [4].

Využití certifikátů v rámci této práce je blíže popsáno v sekci 2.2.2.1.

## 1.5 CSRF – *Cross-site request forgery*

CSRF je forma útoku na webovou aplikaci, při kterém může útočník odeslat falešné požadavky pomocí prohlížeče přihlášeného uživatele [6]. Útok může mít následující průběh:

1. Uživatel je v jedné záložce prohlížeče přihlášen do svého internetového bankovníctví, které není zabezpečeno proti CSRF.
2. Útočník pošle uživateli odkaz na stránku, na který uživatel klikne.
3. Stránka po načtení zašle požadavek internetovému bankovníctví, potvrzující převod peněz na útočníkův účet. Tento požadavek je platný, neboť pochází z prohlížeče přihlášeného uživatele.

Nezabezpečená aplikace internetového bankovníctví nemá jak ověřit, že požadavek nepřišel z jejích stránek.

Útok se dá použít k zaslání nežádoucího požadavku (v kontextu této práce například na smazání záznamu o garáži), ne však ke získání dat ze stránky [6]. Tomu zabraňuje koncept nazvaný *same origin policy*, implementovaný v současných prohlížečích [7]. Tento koncept omezuje to, jak spolu mohou interagovat webové stránky na různých doménách.

Ochranu proti CSRF je možné implementovat vkládáním náhodně generovaných *tokenů* do formulářů na webové stránce. Při odeslání formuláře se pak původ požadavku ověří pomocí tohoto *tokenu* – zaslaný *token* se musí shodovat s *tokenem* vloženým ve webové stránce.

Přihlášený uživatel má přístup k webovým stránkám aplikace, a je tedy možné do odeslaného požadavku vložit ověřovací *token* získaný z formuláře.

Naopak útočník může pouze zasílat požadavky, ale ne získávat data ze stránek, a tudíž nemá přístup k *tokenu*, kterým by mohl svůj požadavek zfalšovat.

Zde je opět důležitá *same origin policy*, která získání *tokenu* zabraňuje. Pokud by tento mechanismus nefungoval, mohl by útočník kromě zasílání požadavků také získávat data ze stránky, a tedy i platné *tokeny*.

Webová aplikace implementovaná v této práci využívá právě tento způsob ochrany proti CSRF (více v sekci 4.4).

### 1.6 *Attack surface*

*Attack surface* popisuje všechny možné body, kde by mohl potenciální útočník napadnout daný systém [8]. V práci je zmíněn v sekci 2.6.4, ve spojitosti se zvýšeným ohrožením při provozu serveru dostupného z internetu.

### 1.7 *Man-in-the-middle* útok

*Man-in-the-middle* útok na protokolu HTTPS využívá situace, kdy certifikát pro ověření totožnosti serveru není důvěryhodný a může být zfalšován [9]. Schéma útoku popisuje obrázek 1.1.

V tom případě může útočník (**Eva**) zachytit komunikaci mezi klientem (**Bob**) a serverem (**Alice**), a místo certifikátu serveru zaslat klientovi svůj certifikát. Poté je komunikace rozdělena na dvě části [9]:

1. **Bob** komunikuje s **Evou** jako by to byla **Alice**. K této komunikaci má **Eva** úplný přístup, neboť je šifrována pomocí jejího certifikátu.
2. Po zachycení komunikace od **Boba** **Eva** tyto data přepošle **Alici**. **Alice** zašle **Evě** odpověď šifrovanou pomocí původního certifikátu. **Eva** tuto odpověď přečte, opět zašifruje pomocí svého certifikátu a pošle **Bobovi**.

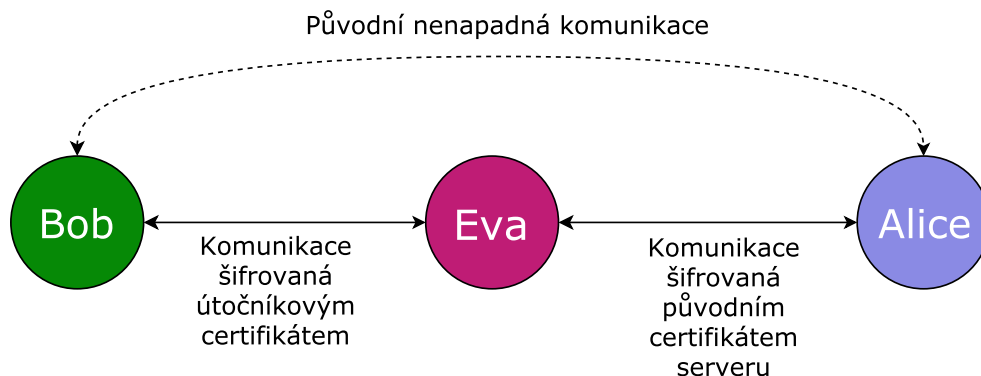
V tomto scénáři **Bob** netuší, že nekomunikuje s **Alicí**. Jelikož **Alice** nedodala důvěryhodný certifikát, nemůže **Bob** zjistit, že používá zfalšovaný.

V textu práce je možnost *man-in-the-middle* útoku diskutována v části zabývající se volbou certifikátu pro provoz HTTPS (sekce 2.2.2.1).

### 1.8 *Hashování*

*Hashováním* se obecně rozumí vytvoření otisku pevné délky – *hashe* – z libovolných vstupních dat. Důležitou vlastností je jednosměrnost tohoto procesu, tedy že z vytvořeného otisku již nelze (v rozumném čase) zrekonstruovat původní vstup. [10]

*Hashování* se používá k zabezpečení uložených hesel. Heslo uložené v čitelné podobě není při úniku souboru s heslem nijak chráněno. Proto je vhodné uložit



Obrázek 1.1: *Man-in-the-middle* útok. Eva funguje jako nežádoucí prostředník mezi Bobem (klientem) a Alicí (původním serverem). Díky tomu má přístup ke zprávám obou účastníků komunikace. [9]

místo hesla samotného jeho *hash*. Pokud dojde k úniku *hashovaných* hesel, je pro útočníka velmi problematické získat z otisků čitelná hesla [10].

*Hashování* lze provést pomocí mnoha algoritmů, ne všechny se však hodí k zabezpečení ukládaných hesel. Některé algoritmy, jako například MD5 či SHA-1, nejsou dostatečně výpočetně náročné, takže umožňují efektivní útoky hrubou silou (lze dostatečně rychle spočítat všechny možné kombinace vstupů do dané délky, a tím zjistit jaký vstup odpovídá danému *hashi*) [10].

Pro bezpečné ukládání hesel je nutné použít dostatečně náročný algoritmus. V této práci je použit algoritmus Argon2, který je považován za vhodný k *hashování* uložených hesel [10].

### 1.8.1 Solení *hashů*

*Hashování* hesla je vhodné doplnit o proces takzvaného „solení“. Při něm se nevytváří otisk samotného hesla, ale hesla doplněného o náhodně vygenerovaný řetězec – sůl. To ztěžuje použití předpočítaných tabulek (například tabulky s otisky často používaných hesel) pro útoky hrubou silou [10].

Pro bližší informace o bezpečném ukládání hesel doporučuji článek *Salted Password Hashing - Doing it Right* [10].





## Analýza

Před samotným návrhem a implementací nadřazeného systému je potřeba přesněji definovat požadavky na jeho vlastnosti a chování. Z hlediska chování by systém měl poskytovat následující funkce:

- Systém má být schopen komunikovat s podřízenými systémy pomocí WiFi či Ethernetu (v rámci místní sítě). Komunikace je založena na zasílání událostí zaznamenaných podřízenými systémy.
- Systém má uchovávat zaznamenané události, včetně data a času přijetí a podřízeného systému, který událost vytvořil.
- Systém má umožňovat správu pomocí webového rozhraní s následujícími funkcemi:
  - Přidat (registrovat) či odebrat podřízený systém.
  - Zobrazit stav registrovaných podřízených systémů.
  - Zobrazit zaznamenané události.
  - Měnit nastavení jednotlivých podřízených systémů.
  - Měnit uživatelské nastavení.
- Systém má být schopen informovat provozovatele při poplachu či poruše.

Dále jsou od nadřazeného systému požadovány tyto vlastnosti:

- **Bezpečnost** – jak komunikace s podřízenými systémy, tak přístup do uživatelského rozhraní by měly probíhat přes zabezpečený komunikační kanál.
- **Spolehlivost** – způsob komunikace i systém jako celek by měl být dostatečně spolehlivý.

- **Presistence** – data v systému by měla být uchovávána persistentním způsobem.
- **Nezávislost** – systém by neměl být příliš závislý na externích službách a alespoň v omezené míře fungovat i bez přístupu k internetu.

Na základě těchto požadavků je tedy potřeba zvolit vhodné nástroje pro implementaci systému. To je v první řadě komunikační protokol použitý pro komunikaci s podřízenými systémy. Tento protokol by měl být dostatečně robustní, rozšířený a podporovat šifrování. Také by s jeho pomocí mělo být možné bezpečně ověřit identitu podřízených systémů. Výběrem protokolu se zabývá sekce 2.2.

Další důležitá volba je způsob uchování zachycených událostí a dalších údajů. Data je nutné uchovávat persistentně, nejlépe v obecně používaném formátu, který umožní další zpracování i mimo nadřazený systém. Výběr úložiště je popsán v sekci 2.3.

Nadřazený systém má být také schopen upozornit uživatele v případě hrozícího nebezpečí nebo výpadku podřízeného systému. To je možné provést mnoha způsoby, jejichž možnosti jsou shrnuty v sekci 2.4.

Nakonec zbývá zvolit programovací jazyk a případně frameworky pro tvorbu aplikace. Vhodných jazyků je dnes velké množství, jde tedy spíše o otázku osobní preference. Volbou programovacího jazyka se zabývá sekce 2.5.

Z těchto požadavků vyplývá software potřebný k implementaci nadřazeného systému, jehož dostupnost je hlavní požadavek při volbě hardwarové platformy v sekci 2.6.

### 2.1 Struktura systému

Struktura celého systému je naznačena na obrázku 2.1. [Podřízené systémy](#) komunikují s [nadřazeným](#) na základě událostí. Nadřazený systém tyto události zpracovává a upravuje podle nich stav garáží v evidenci.

Zaznamenané události jsou také uchovávány v historii událostí spolu s dalšími metadaty, jako například čas přijetí.

Komunikace mezi podřízeným a nadřazeným systémem je postavena na modelu *client/server*. Nadřazený systém provozuje server zvoleného protokolu (viz sekci 2.2), ke kterému se podřízené systémy připojují. Komunikaci tedy vždy iniciuje podřízený systém. S možností zasílání nevyžádaných zpráv podřízeným systémům v této práci nepočítám, mohl by to však být námět pro další rozšíření.

Další, kdo přistupuje do systému, je [uživatel](#). Přes webové rozhraní může sledovat stav garáží a historii událostí. Také zde může spravovat klíče, které slouží pro přístup ke komunikačnímu API systému. Přístup do webového rozhraní je zabezpečen heslem.



Obrázek 2.1: Základní struktura systému. Podřízené systémy využívají komunikační API k zasílání událostí. Nadřazený systém tyto události zaznamenává a upravuje podle nich stav garáží. Uživatel nadřazený systém ovládá pomocí webového rozhraní.

### 2.1.1 Podřízený systém

Podřízený systém je zařízení umístěné v každé garáži, které sleduje stav okolí pomocí těchto senzorových vstupů:

- detekce kouře,
- detekce pohybu,
- stav dveří.

Základní požadavek na podřízený systém je schopnost komunikace přes Ethernet či WiFi pomocí protokolu zvoleného v sekci 2.2. Jinak může být hardware prakticky libovolný.

Návrh a implementace podřízeného systému není součástí této práce. Jeho vývojem se zabývá Miroslav Váňa ve své bakalářské práci na Katedře číslicového návrhu Fakulty informačních technologií ČVUT.

### 2.1.2 Událost

Událost slouží jako základní komunikační jednotka mezi nadřazeným a podřízeným systémem. V případě překročení mezních hodnot senzorů se jednotlivé podřízené systémy okamžitě hlásí nadřazenému systému. Kromě toho také v pravidelných intervalech odesílají kontrolní hlášení.

## 2.2 Výběr komunikačního protokolu

Nejdřív je nutné určit způsob komunikace, který bude systém používat. Díky tomu se budu při vybírání platformy moci ujistit, že jsou dostupné vhodné knihovny a další software.

Nadřazený systém bude se svými klienty (monitorovací zařízení v jednotlivých garážích) komunikovat přes WiFi nebo Ethernet. Základem komunikace bude TCP/IP protokol [2], je však potřeba zvolit vhodný protokol z aplikační vrstvy ISO OSI modelu [2], který na něm bude stavět.

### 2.2.1 Vlastní protokol

Jedna z možností je implementovat vlastní protokol pomocí TCP/IP *socketů*. Toto řešení se mi však nezdá příliš vhodné, neboť nepřináší žádné významné výhody, naopak se s ním pojí řada komplikací.

Pro vlastní protokol by bylo nutné vytvořit robustní server, který zvládá obsluhu více klientů najednou. Dále by vzhledem k citlivosti přenášených dat bylo nutné implementovat nějakou formu šifrování. Tyto velmi obsáhlé problémy přitom řeší většina dnešních protokolů.

Další nevýhodou je nutnost implementace klientské části protokolu při vytváření nových zařízení spravovaných nadřazeným systémem. To do jisté míry omezuje jeho rozšiřitelnost.

### 2.2.2 Protokol HTTPS

Další možnost je využít ke komunikaci protokol HTTPS. V tomto případě by klienti komunikovali se systémem pomocí HTTP metod jako například `get` nebo `post`.

Jelikož součástí požadavků na systém je i webové uživatelské rozhraní, bude v každém případě nutné použít webový server pro jeho provoz. Ten by pak bylo možné využít i k poskytnutí API pro komunikaci systému s podřízenými systémy.

Vhodný webový server (jako například Apache [11]) zajistí vícevláknovou obsluhu všech klientů. Protokol se také postará o šifrování přenášených dat, je však nutné získat certifikát k ověření pravosti serveru (viz sekci 2.2.2.1).

Certifikát bude potřeba zajistit i v případě, že komunikace s klienty nebude postavena na tomto protokolu. Je totiž nutné také zabezpečit webové rozhraní,

například kvůli ověření totožnosti uživatele při přihlašování. Nutnost pořízení certifikátu tedy nepředstavuje nevýhodu oproti jiným protokolům.

API realizované pomocí tohoto protokolu je poměrně snadno rozšiřitelné. Pro nově implementovanou operaci stačí definovat URL a případně formát přenášených dat.

Výhodou je také snadná implementace na straně klienta, tedy podřízeného systému. Knihovny realizující klientskou část protokolu jsou dostupné na většině populárních platform jako například Arduino (s Ethernet *shieldem*, oficiální knihovna EthernetClient [12]) nebo ESP8266 (knihovna esp8266wifi [13]).

### 2.2.2.1 Certifikáty pro provoz HTTPS

Pro provoz HTTPS serveru lze použít například certifikáty certifikační autority Let's Encrypt, které jsou poskytovány zdarma [14]. Kromě toho dodává Let's Encrypt také automatizačního klienta Certbot [15] pro snadné nasazení a aktualizaci jejich certifikátů. Bohužel certifikáty jsou vydávány pouze na doménu [14], což komplikuje použití v místní síti.

Jiná možnost je použití *self-signed* certifikátu, který není podepsaný žádnou certifikační autoritou, ale pouze vlastníkem certifikátu [16]. Může tedy sloužit k šifrování komunikace (poskytuje veřejný klíč), ale je zranitelný vůči *man-in-the-middle* útoku [16].

*Self-signed* certifikát však lze použít k šifrování komunikace na uzavřené lokální síti za předpokladu, že je server s certifikátem (přesněji s jeho soukromým klíčem) dostatečně zabezpečen [16].

Nevýhodou tohoto řešení je nedůvěra webových klientů (certifikát není podepsán certifikační autoritou a nelze tedy ověřit jeho pravost), což by ovlivnilo přístup k uživatelskému rozhraní a API systému. V případě webového rozhraní by prohlížeč zobrazil varování o neznámém certifikátu. To by však mohl uživatel ignorovat.

Podřízené systémy by při zasílání požadavků musely přeskočit krok ověření totožnosti serveru. Jak toho dosáhnout například v knihovně Requests (umožňující vytváření HTTP požadavků [17]) pro jazyk Python [18] je naznačeno v ukázce 1.

Tento jazyk a knihovna byly zvoleny z důvodů kompaktnosti ukázkového kódu, v jiných jazycích lze tuto verifikaci obejít obdobným způsobem.

```
>>> import requests
>>> r = requests.get('https://test.local/hello', verify=False)
>>> r.status_code
200
```

Ukázka 1: Vytvoření HTTPS požadavku v knihovně Requests, bez ověření totožnosti serveru.

### 2.2.2.2 Autentizace klientů na HTTPS

Přístup k API nadřazeného systému by měl být povolen pouze ověřeným klientům. Díky tomu bude možné zabránit například zasílání nepravdivých informací z neznámých zdrojů.

Jednoduchou autentizaci přes HTTPS lze realizovat například pomocí generování API klíčů. Pro každý podřízený systém bude vygenerován klíč, kterým se při zasílání požadavku systém prokáže. Seznam platných klíčů by byl udržován v databázi nadřazeného systému. Klíče by uživatel mohl přidávat nebo odebírat (například v případě odcizení podřízeného systému) pomocí webového rozhraní.

Tyto klíče by také bylo nutné nahrát a uchovávat na podřízených systémech. Detaily tohoto procesu by závisely na platformě těchto systémů. Například u Arduina by šlo klíč nahrát z uživatelského počítače pomocí sériové linky (s USB převodníkem) a udržovat ho v paměti EEPROM, která je uchována i po odpojení napájení [19].

Také by bylo možné implementovat v nadřazeném systému „registrační mód“, který by bylo možné dočasně povolit ve webovém rozhraní. V tomto módu by systém po přijetí speciálního API požadavku vygeneroval nový klíč, který by si uložil do své databáze platných klíčů, a také ho v odpovědi zaslal žádajícímu zařízení. Pokud by mód povolen nebyl, odpověď by systém chybovým kódem, například *403 – Forbidden*. Zaslání požadavku z podřízeného systému by mohlo být provedeno stisknutím tlačítka.

Tento přístup by byl pravděpodobně uživatelsky příjemnější, přináší však potencionální bezpečnostní rizika. Například pokud by uživatel zapomněl mód vypnout, systém by byl otevřený k registraci nežádoucích zařízení. Takový problém by se však dal řešit například automatickou deaktivací módu po uplynutí časového limitu.

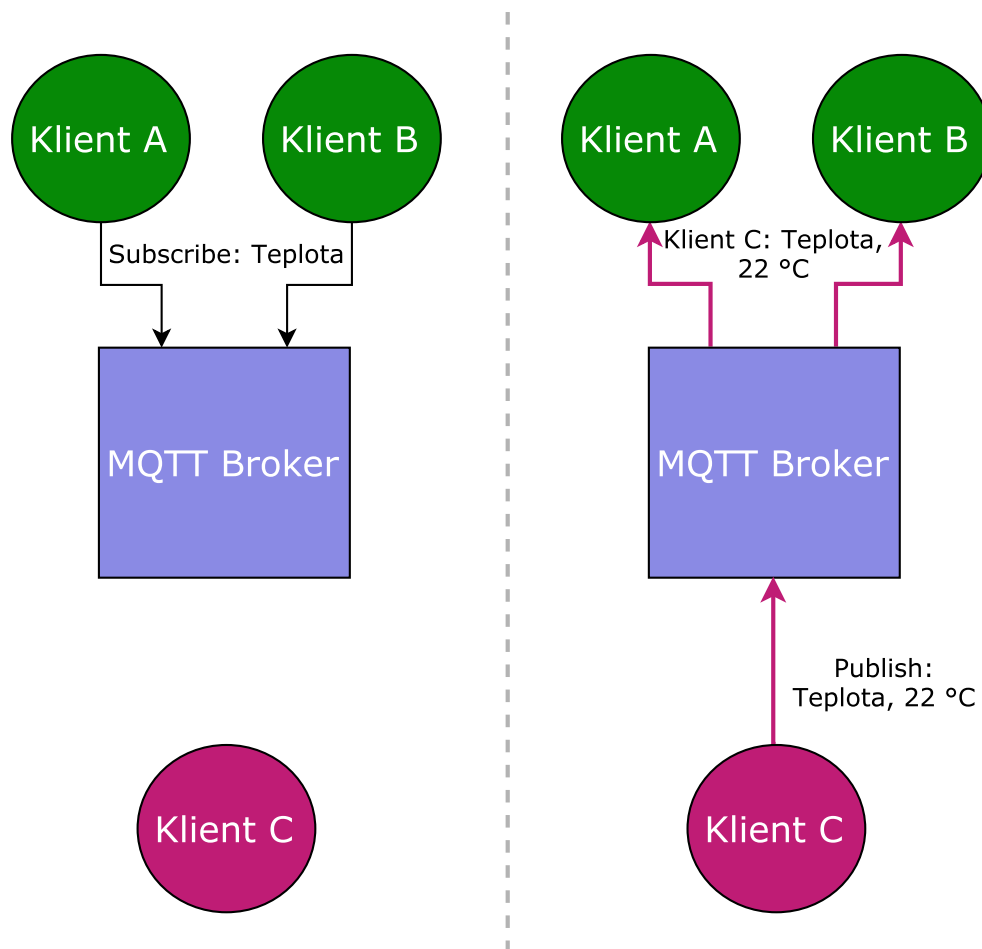
Útočník snažící se získat klíč k API by také mohl periodicky zkoušet registrační požadavek a čekat na aktivaci módu. Obrana proti tomuto útoku by byla složitější, šlo by například filtrovat IP adresy s příliš častými požadavky.

Obecně vycházím z toho, že i v případě registrace nežádoucího zařízení nemůže toto zařízení krátkodobě způsobit výraznější škody – do databáze nadřazeného systému může pouze zasílat nová data, která jsou navíc vázána k jeho identitě (API klíči). Nemůže tedy získávat data od jiných podřízených systémů či měnit jejich záznamy. Neautorizované zařízení se také objeví v seznamu registrovaných API klíčů, kde může být snadno odhaleno.

### 2.2.3 Protokol MQTT

MQTT je komunikační protokol založený na modelu *publisher/subscriber*, určený pro použití v prostředí s omezenými zdroji (malý výkon procesoru, omezená paměť atd.) [20].

Komunikace mezi jednotlivými klienty v systému je zprostředkována pomocí centrály nazývané *broker*. Ta spravuje adresy – *topics*. Na nich mohou klienti publikovat či odebírat zprávy.



Obrázek 2.2: Příklad struktury protokolu MQTT [21]. Klienti A a B jsou *subscriber* *topicu* „Teplota“. Když klient C na tuto adresu publikuje novou zprávu, *broker* se postará o její doručení všem *subscriberům*.

Na obrázku 2.2 klienti A a B začnou odebírat *topic* „Teplota“. Když pak klient C publikuje zprávu na tuto adresu, *broker* se postará o doručení všem odebírajícím klientům.

Adresy je možné hierarchicky strukturovat následujícím způsobem [21]:

- Lze tvořit skupiny, například:
  - /senzory/obyvak/teplota,

- `/senzory/kuchyne/vlhkost`.
- Zpráva je nutné publikovat na jednoznačnou adresu.
- Při odebírání je možné použít modifikátory pro specifikování skupiny adres:
  - Modifikátor `+` – odpovídá libovolnému jednomu stupni hierarchie. Například pro odebírání všech senzorů vlhkosti lze použít adresu `/senzory/+/vlhkost`.
  - Modifikátor `#` – odpovídá libovolnému počtu libovolných stupňů. Adresa `/senzory/#` tedy slouží k odebírání všech dat.

V případě této práce by nadřazený systém i podřízené systémy byly klienty *brokeru*. Podřízené systémy by publikovaly naměřená data, která by nadřazený systém odebíral. Tím by vznikla obdoba modelu *client/server*, zmíněného v sekci 2.1.

Samotný *broker* by pak mohl běžet souběžně s nadřazeným systémem na zvolené platformě (například *broker* Mosquitto je dostupný na řadě platform, včetně Raspberry Pi [22]).

Protokol podporuje tři možnosti QoS [20]:

- Nejvýše jedno doručení – tento mód pouze odešle zprávu, není zahrnut žádný opakovací mechanismus pro případ nedoručení.
- Alespoň jedno doručení – v tomto módu je zaručeno doručení zprávy, ta však může být doručena vícekrát.
- Přesně jedno doručení – zde je ošetřeno i duplicitní doručování zpráv.

Použití sofistikovanějších metod doručení má vliv na výkon, a proto se v některých případech vyplatí zvolit nižší úroveň QoS (například při posílání idempotentních zpráv). Pro tuto práci bych však pravděpodobně zvolil záruku přesně jednoho doručení.

### 2.2.3.1 Šifrování a autentizace na MQTT

V této části se budu zabývat prostředky pro šifrování komunikace, které jsou dostupné v *brokeru* Mosquitto.

První možnost je pro zabezpečení komunikace využít certifikáty, podobně jako u HTTPS. Zde by se pravděpodobně také využil *self-signed* certifikát (blíže popsáný v sekci 2.2.2.1). Mosquitto navíc také vyžaduje kořenový certifikát certifikační autority [23]. Při použití *self-signed* certifikátů by bylo nutné tuto autoritu vytvořit a používané certifikáty u ní podepsat (pro bližší informace viz [24]). Kořenový certifikát by také bylo nutné distribuovat klientům.



Kromě certifikátů lze pro šifrování použít i PSK. V tom případě *broker* a jeho klienti používají pro zašifrování komunikace společný klíč (známý jak klientovi, tak *brokerovi*). Různí klienti přitom mohou mít různé klíče. [25]

Bohužel podpora PSK v MQTT klientech není příliš rozšířená. PSK je možné použít v knihovně *libmosquitto*, určené pro C/C++ (s vazbami pro Python). U této knihovny se mi však podařilo najít pouze manuálovou stránku (viz [26]), bez informací o jejím dalším vývoji či udržování. Modul poskytující vazby do Pythonu byl nicméně předán projektu *Paho* [27].

*Paho* poskytuje implementace MQTT klientů pro mnoho platform (včetně například *Arduina* [28]). Dokumentace klientů pro C++ a Python však možnost šifrování pomocí PSK vůbec nezmiňuje [29, 30].

Tyto možnosti lze použít i k autentizaci klientů *brokeru*. Při použití certifikátů lze v konfiguračním souboru *Mosquitto* zvolit `require_certificate` [25]. Poté bude od klienta vyžadován certifikát prokazující jeho totožnost. Při použití PSK lze k autentizaci využít sdílený klíč (*broker* odmítne klienty s neplatnými klíči) [25]. Kromě toho je možno použít také autentizaci pomocí uživatelského jména a hesla, která je součástí MQTT protokolu, případně totožnost klientů neověřovat vůbec (a pouze šifrovat komunikaci) [25].

#### 2.2.4 Závěr výběru protokolu

V sekcích 2.2.2 a 2.2.3 jsem blíže prozkoumal dva poměrně rozšířené protokoly aplikační vrstvy, které by bylo možné použít pro tvorbu nadřazeného systému.

Pokud by mezi požadavky na systém bylo zahrnuto zasílání nevyžádaných zpráv podřízeným systémem (jak je zmíněno v sekci 2.1), zvolil bych pravděpodobně protokol MQTT. V něm je tato funkcionality velmi snadno implementovatelná – stačí, aby podřízené systémy odebíraly *topic*, na kterém by nadřazený systém publikoval zprávy.

Jelikož se však v této práci zabývám systémem, který zprávy pouze přijímá a zaznamenává, rozhodl jsem se pro HTTPS. Nasazení tohoto protokolu je o něco snazší (není potřeba na zařízení instalovat *broker* a zařizovat certifikační autoritu – stačí *self-signed* certifikát) a s jeho použitím mám více zkušeností. Také se částečně uvolní požadavky na volbu platformy (webový server bude potřeba v každém případě, při volbě HTTPS jako komunikačního protokolu mezi systémy tedy nebude nutný žádný další software).

Každopádně bude mým cílem navrhnout výslednou aplikaci tak, aby rozhraní pro podřízené systémy realizované pomocí HTTPS bylo možné snadno nahradit MQTT rozhraním.

K zabezpečení komunikace (včetně webového rozhraní) použiju *self-signed* certifikát. Hlavní důvod je požadavek na použití v místní síti, bez zaručeného přístupu k internetu. Toto rozhodnutí nemá vliv na návrh a implementaci systému, pouze na jeho nasazení – konkrétně konfiguraci webového serveru.

Pokud by provozovatel plánoval mít systém přístupný z internetu (přes registrovanou doménu), musí v každém případě k zabezpečení použít certifikát

podepsaný důvěryhodnou certifikační autoritou. Pak stačí pouze v konfiguračním souboru webového serveru nahradit *self-signed* certifikát podepsaným certifikátem. Není tedy nutné provádět změny v kódu aplikace.

### 2.3 Ukládání dat

Zaznamenané události bude potřeba persistentně uchovávat. Zde by šly využít jednoduché textové logy, vhodnější však bude zvolit nějaký databázový systém – například kvůli širším možnostem zpracování naměřených údajů.

Z dostupných možností mě zaujal SQLite, což není klasický databázový stroj s modelem *client/server*, ale místo toho tvoří součást programu, který databázi používá. Přístup k datům je realizován pomocí přímého čtení/zápisu do databázového souboru na disku. Díky tomu má malé nároky na diskový prostor a operační paměť. [31]

Jelikož bude nadřazený systém pravděpodobně provozován na hardwaru s omezenými zdroji, představují tyto vlastnosti nezanedbatelnou výhodu. Použití SQLite také zjednoduší nasazení aplikace, neboť nebude nutné vytvářet a konfigurovat databázový server.

#### 2.3.1 Velikost databáze

Vzhledem k požadavku na spolehlivost systému je vhodné odhadnout předpokládaný růst velikosti databázového souboru. Příliš velká databáze by mohla vyčerpat paměť zařízení, a způsobit tak jeho pád. Také by mohla negativně ovlivnit dobu odezvy nadřazeného systému.

Pokud by se ukázalo, že při dlouhodobém provozu začíná být databáze neúnosně velká, bylo by nutné omezit množství uchovávaných dat, například pomocí front zaznamenaných událostí.

Při odhadu jsem vycházel z předpokladu, že největší objem dat v databázi budou představovat zaznamenaná kontrolní hlášení podřízených systému. Pokud se budou podřízené systémy hlásit každou hodinu, představuje to 24 záznamů denně na každou garáž, což by mělo být řádově více než u jiných událostí. Například u pravděpodobně druhé nejčastější události, tedy otevření/zavření dveří, předpokládám nejvýše jednotky denně.

Událost kontrolního hlášení by měla obsahovat tyto údaje:

- Identifikátor události v databázi.
- Identifikátor garáže, ke které událost patří.
- Datum a čas zaznamenání události.
- Typ události (kontrolní hlášení).

Identifikátory a typ události pravděpodobně budou 32bitová celá čísla. Datum je možné v SQLite ukládat buď jako textový řetězec s formátem `YYYY-MM-DD HH:MM:SS.SSS`, 64bitové reálné číslo, či 32bitové celé číslo – *Unix Time* [32]. Z toho vyplývá, že jedno kontrolní hlášení bude mít velikost alespoň  $4 \cdot 32 = 128$  bitů.

Pokud nadřazený systém monitoruje 30 garáží, přičemž každá odesílá kontrolní hlášení každou hodinu, jsou každý den vytvořeny záznamy o velikosti  $30 \cdot 24 \cdot 96 = 98304$  bitů, tedy asi 98 Kb. Dá se proto předpokládat, že velikost databázového souboru neporoste nijak dramaticky.

Tento výpočet představuje pouze odhad předpokládané velikosti. Skutečná velikost bude záležet na zvolené implementaci, neměla by se však výrazně lišit od tohoto odhadu.

## 2.4 Zasílání upozornění

Nadřazený systém má být schopen zasílat uživatelům notifikace o stavu jednotlivých garáží. Notifikace mohou informovat o důležité události (například detekce kouře) nebo o výpadku podřízeného systému (v případě promeškání plánovaného kontrolního hlášení).

Tyto notifikace by měl obdržet jednak správce garáží (provozovatel nadřazeného systému), jednak jednotliví nájemci, kterým budou zasílány notifikace týkající se jejich garáží.

V následujících sekcích jsem vyzkoušel několik přístupů k odesílání těchto notifikací, a to jak pomocí e-mailu, tak SMS.

### 2.4.1 E-mailové notifikace

První možnost je zasílat notifikace pomocí e-mailu. K tomu je zapotřebí vytvořit účet, který by mohl nadřazený systém použít k odesílání.

Zde se nabízí možnost provozovat e-mailový server s účtem přímo na zařízení, na kterém poběží nadřazený systém. Provoz takového serveru je však velmi náročný, a to nejen kvůli složité počáteční konfiguraci, ale také z hlediska dlouhodobé údržby [33]. Tento přístup by tak kladl neúměrné nároky na uživatele systému.

Alternativa je použít nějakého poskytovatele e-mailových služeb, jako například Gmail od společnosti Google. Zde by si uživatel vytvořil účet určený k odesílání notifikací, a poté autorizoval nadřazený systém k odesílání zpráv z tohoto účtu.

Tímto je částečně porušen požadavek na nezávislost (viz sekci 2). Při zasílání notifikací je však nutný přístup k internetu i v případě použití vlastního e-mailového serveru a nezávislost na externí službě nevyváží komplikace spojené s jeho provozem.

Gmail umožňuje přístup k účtu pomocí protokolu SMTP, je však nutné v nastavení účtu Google povolit přístup méně zabezpečeným aplikacím, pro-

tože přihlášení je prováděno pomocí uživatelského jména a hesla [34]. Jelikož by přihlašovací údaje musely být uloženy v nadřazeném systému v čitelné podobě, není tento způsob vhodný.

Bezpečnější možnost je použít k přístupu protokol OAuth2 [35]. S jeho pomocí je možné udělovat aplikacím (v tomto případě nadřazenému systému) přístup k částem účtu Google, tedy i Gmailu. Autentizace probíhá na základě náhodně generovaných klíčů, takže není nutné uchovávat v nadřazeném systému heslo, které mohl uživatel systému použít i v jiných službách, v čitelné podobě [36].

### 2.4.2 SMS notifikace

Alternativa k e-mailu je posílání upozornění pomocí SMS. To může být pro uživatele vhodnější, protože lze upozornění přijímat na prakticky jakémkoliv telefonu, bez nutnosti e-mailového klienta či připojení k internetu.

Posílání SMS z nadřazeného systému lze provést dvěma způsoby. První možnost je připojit k počítači, na kterém nadřazený systém poběží, GSM modul se SIM kartou. Také je možné k posílání SMS využít internetovou službu, jako je například Twilio.

#### 2.4.2.1 GSM modul

Poměrně snadný způsob, jak odeslat SMS pomocí počítače, je využít GSM modul, který lze připojit do USB (například modul E3131 od společnosti Huawei, zobrazený na obrázku 2.3). Počítač může s modulem komunikovat pomocí sériové linky a příkazů, které definuje GSM standard [37]. Díky tomu nejsou nutné specializované ovladače a modul lze použít na mnoha různých zařízeních.

Modul je možné ovládat přímo pomocí sériové linky, vhodnější je však použít program či knihovnu, která zařídí zasílání příslušných příkazů. To je například projekt Gammu, který umožňuje komplexní ovládání modulu buď pomocí příkazové řádky, nebo knihovny v jazyce C [38].

Výhoda tohoto řešení je nezávislost na internetu. Odesílání upozornění by tedy fungovalo i pokud by zařízení mělo přístup pouze k místní síti. Řešení je také poměrně snadno implementované (po připojení modulu a instalaci knihovny stačí k odeslání SMS jediný příkaz) a díky jednoduchému komunikačnímu rozhraní modulu výrazně neomezuje požadavky na hardware či ovladače.

Za hlavní nevýhodu považují cenu řešení. Jednak je potřeba pořídit GSM modul, jednak je nutné platit odeslané SMS (zde by záleželo na zvoleném tarifu). Méně významné nevýhody představují nutnost udržovat v modulu aktivní SIM kartu a případně dostatečný kredit.



Obrázek 2.3: GSM modul Huawei E3131 s odkrytým slotem pro SIM kartu.

### 2.4.2.2 Twilio

K odesílání SMS lze také využít službu Twilio, která umožňuje registraci telefonního čísla a posílání SMS pomocí webového API [39]. K tomu lze použít připravenou knihovnu pro Python [40]. Přístup k API je ověřen pomocí klíčů, které uživatel obdrží po registraci [41].

Toto řešení je levnější než řešení využívající GSM modul, neboť není nutné kupovat žádný další hardware. Samotné odesílání SMS je také poměrně levné – jedna zpráva odeslaná v rámci České republiky přijde na 0.049 USD [42]. Kromě toho Twilio poskytuje další služby, které by bylo možné použít k budoucímu rozšíření nadřazeného systému [43].

Nevýhoda může být (stejně jako u řešení založeného na e-mailech) nutnost připojení k internetu a registrace účtu. Kromě toho je nutné na účtu udržovat kredit k odesílání SMS, což vyžaduje zadání platební karty.

### 2.4.3 Závěr výběru způsobu zasílání upozornění

Po otestování výše zmíněných metod zasílání upozornění jsem se rozhodl použít GSM modul. Hlavní důvod pro toto rozhodnutí byla vyšší nezávislost nadřazeného systému (není potřeba internetové připojení) a poměrně snadná implementace.

Zasílání upozornění pomocí SMS se mi zdá vhodnější než pomocí e-mailu. E-mail může být pro uživatele hůře dostupný – k jeho kontrole potřebuje vhodný telefon či počítač s připojením k internetu. Příchozí SMS je také snazší vzít na vědomí.

## 2.5 Programovací jazyk pro tvorbu systému

Pro tvorbu systému jsem se rozhodl použít programovací jazyk Python [18]. S tímto jazykem mám nejvíce zkušeností co se týče implementace webových

aplikací. Je také dostatečně rozšířený, takže výsledný systém bude možné nasa-  
dit na poměrně širokém spektru platform bez nutnosti složitějšího portování.

K vytvoření webového rozhraní i API pro podřízené systémy jsem zvolil framework Flask [44]. Hlavním důvodem jsou opět předchozí zkušenosti s tímto frameworkem. Flask také dává více volnosti při návrhu aplikace než například také velmi rozšířený framework Django [45].

## 2.6 Výběr platformy

Pro realizaci systému je nutné zvolit vhodnou platformu. Jelikož je cílem práce vytvořit fyzické zařízení, rozhodl jsem se jako základ použít některý z jedno-  
deskových počítačů, které jsou v dnešní době na trhu. Tyto počítače bývají  
cenově velmi dostupné a zároveň poskytují dostatečný výkon a podporu pro  
provoz systému.

Při výběru počítače byla nejdůležitějším kritériem podpora softwaru po-  
třebného k implementaci monitorovacího systému. Na základě předchozí ana-  
lýzy je tedy vyžadován následující software:

- Webový server Apache2.
- Databázový systém SQLite3.
- Programovací jazyk Python3.
  - Webový framework Flask.

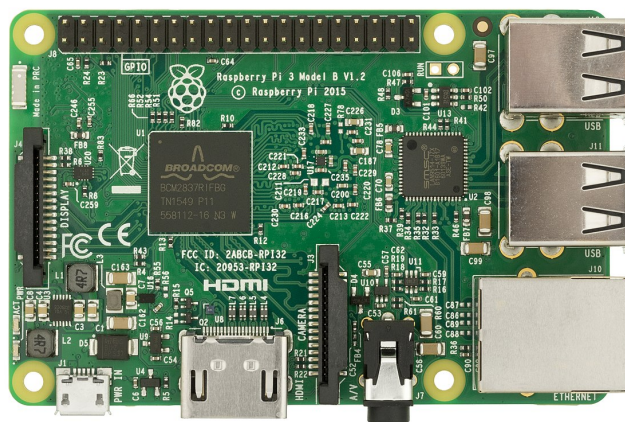
Pro provoz tohoto softwaru bude potřeba plnohodnotný operační systém,  
což vylučuje platformy využívající jednoduché mikrokontroléry, jako například  
Arduino Uno. Kromě toho je nutné připojení k síti pomocí Ethernetu nebo  
WiFi.

V dalších sekcích jsem blíže zkoumal jednodeskové počítače Raspberry Pi  
(sekce 2.6.1) a Zybo Zynq-7000 (sekce 2.6.2) a zvážil jejich výhody a nevýhody  
pro implementaci systému.

### 2.6.1 Raspberry Pi

Raspberry Pi je velmi rozšířený jednodeskový počítač. Jeho poslední verze,  
Raspberry Pi 3, je postavena na SoC Broadcom BCM2837 s čtyřjádrovým  
procesorem ARM Cortex A53, který je až o 50 % rychlejší než procesor před-  
chozí verze [46].

Dále nová verze přináší vlastní WiFi modul [46], není tedy nutné se spo-  
léhat na externí moduly. Kromě toho je možné počítač připojit k síti pomocí  
Ethernetového portu. Ten je omezený na 100 Mb/s [46], to by však vzhledem  
k objemu dat přenášných mezi nadřazeným a podřízenými systémy nemělo  
představovat problém.



Obrázek 2.4: Raspberry Pi 3 (obrázek převzat z [https://en.wikipedia.org/wiki/Raspberry\\_Pi](https://en.wikipedia.org/wiki/Raspberry_Pi)).

Deska také obsahuje čtyři USB a jeden HDMI port [46]. Ty nejsou pro implementovaný systém zásadní, nicméně při počáteční konfiguraci zařízení (například nastavení WiFi hesla) může být připojení monitoru a klávesnice pro některé uživatele pohodlnější než použití SSH či sériové linky. Připojený monitor se také hodí při řešení problémů se startem operačního systému.

Raspberry Pi 3 bohužel nemá vlastní bateriově zálohovaný RTC obvod a k udržování času využívá protokol NTP [47]. K tomu je však zapotřebí internetové připojení. Jelikož by zařízení mělo být možné používat i v síti bez přístupu k internetu, je nutné připojit externí RTC obvod, například pomocí I2C sběrnice [47]. Poté je možné systémové hodiny synchronizovat bez internetového připojení pomocí tohoto obvodu.

S počítačem je možné použít množství operačních systémů, z nichž nejrozšířenější je pravděpodobně Raspbian, linuxový systém postavený na Debianu [48]. Pro ten jsou dostupné všechny potřebné softwarové balíčky popsané v sekci 2.6. Jelikož počítač nemá žádné vlastní úložiště, je nutné operační systém provozovat na vložené SD kartě [46].

Jednou z výhod tohoto počítače je obrovské množství podporovaných hardwarových periférií a knihoven pro ně. V této práci bych využil pouze zmíněný RTC obvod, případné další rozšiřování systému (například o vestavěný LCD displej) bude na Raspberry Pi pravděpodobně snadnější než na jiných platformách.

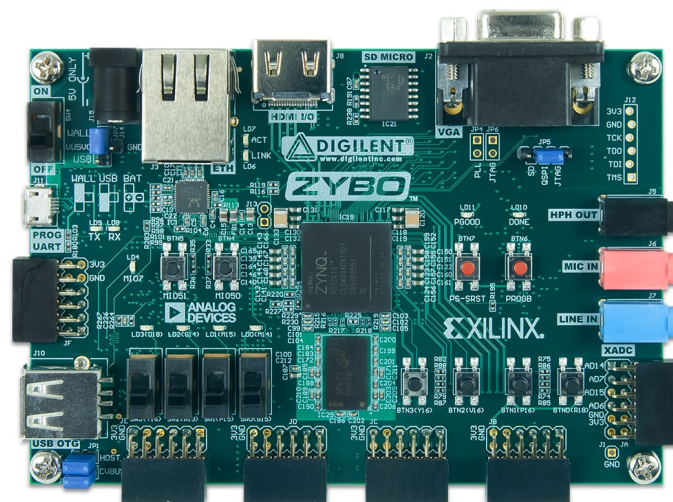
Kromě široké podpory je hlavní výhodou Raspberry Pi jeho cena. Poslední verze se pohybuje kolem 1200 Kč. K celkovým nákladům na systém je ještě třeba připočítat cenu RTC obvodu a SD karty. Zde počítám s použitím již připraveného modulu s obvodem PCF8523 (pro bližší informace o modulu viz [47]), jehož cena je přibližně 200 Kč. Jako úložiště by měla plně dostačovat

## 2. ANALÝZA

---

16GB SD karta, která se dá pořídit za 200 Kč. Náklady na hardware systému (bez GSM modulu) by se tedy měly pohybovat kolem 1600 Kč<sup>3</sup>.

### 2.6.2 Zybo Zynq-7000



Obrázek 2.5: Přípravek Zybo Zynq-7000 (obrázek převzat z <https://www.xilinx.com/products/boards-and-kits/1-4azfte.html>).

Tento přípravek od společnosti Digilent je postavený na SoC Xilinx Zynq Z-7010. Hlavní předností tohoto čipu je kombinace dvoujádrového procesoru ARM Cortex A9 s FPGA odpovídající sérii Artix-7 [49].

Díky tomu je možná těsná integrace mezi aplikací běžící na procesoru a výkonnými, úzce specializovanými moduly, které jsou syntetizované na FPGA. Tento přístup, kdy je hardware a software systému vyvíjen souběžně, se označuje jako *hardware/software codesign* [50].

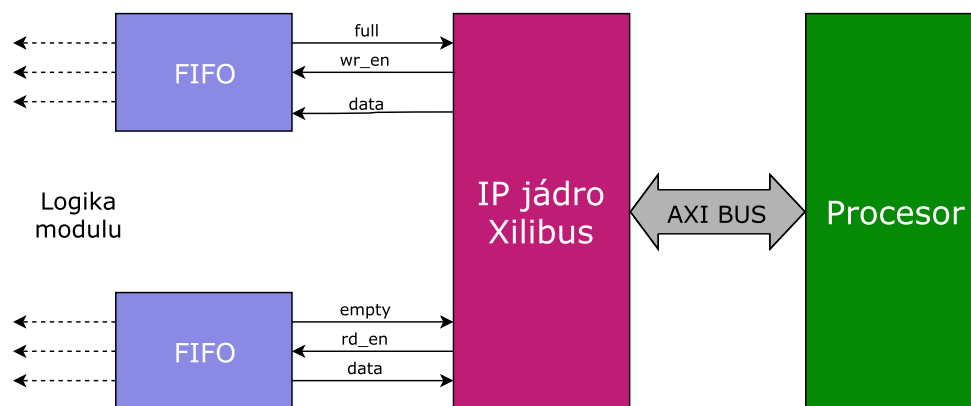
Pro SoC ze série Zynq vznikla linuxová distribuce Xilinx, vycházející z Ubuntu. Kromě plnohodnotného operačního systému (včetně například grafického rozhraní) poskytuje Xilinx také ovladače pro komunikaci s FPGA pomocí AXI sběrnice [51]. K tomu využívá **IP jádro Xilibus**, které funguje jako adaptér mezi **procesorem** a FPGA modulem (viz obrázek 2.6). Ten pak ke komunikaci může využívat standardní **FIFO** fronty a nemusí se zabývat AXI sběrnici [51].

Jelikož Xilinx staví na Ubuntu (konkrétně na verzi 12.04 LTS [51]), neměl by být problém nainstalovat software potřebný pro provoz nadřazeného systému (viz sekci 2.6).

---

<sup>3</sup>Ceny převzaté z obchodů Alza (<https://alza.cz>) a SnailShop ([www.snailshop.cz](http://www.snailshop.cz)), v únoru 2018.





Obrázek 2.6: Blokové schéma využití IP jádra Xilibus [51]. Procesor přípravku komunikuje pomocí AXI sběrnice s jádrem Xilibus. To požadovaná data zapíše či přečte z FIFO fronty příslušného modulu.

Přípravek je možné připojit k síti pomocí ethernetového portu, který podporuje rychlost až 1 Gb/s [49]. WiFi připojení by bylo možné realizovat pomocí USB modulu. Dále přípravek obsahuje HDMI a VGA port, audio konektory, čtyři tlačítka, čtyři přepínače a slot pro SD kartu [49].

Na přípravku je také k dispozici 128 MB flash paměti [49], k provozu tedy teoreticky není potřeba SD karta. Tato paměť by však pravděpodobně nestačila k instalaci vhodného operačního systému a potřebného softwaru. I zde by tedy bylo nutné použít SD kartu.

Stejně jako Raspberry Pi tato deska postrádá RTC obvod. Firma Digilent však dodává externí obvod, který lze připojit pomocí rozhraní Pmod [52].

Nevýhodou této desky (především v porovnání Raspberry Pi) je její cena. Ta se pohybuje okolo 4000 Kč<sup>4</sup>. K tomu je nutné přičíst náklady na SD kartu a RTC obvod, případně i WiFi modul. Cena zařízení by se tedy pohybovala v rozmezí 4500 až 5000 Kč.

### 2.6.3 Závěr výběru platformy

Jak vyplývá z tabulky 2.1, Raspberry Pi 3 poskytuje znatelně výkonnější procesor a více paměti za méně než třetinu ceny desky Zybo. Hlavní přidaná hodnota Zybo tedy spočívá v integraci s FPGA, ta má však v případě této práce pouze velmi omezené využití.

FPGA by bylo možné využít například k šifrování úložiště, vzhledem k předpokládaným objemům dat by však zrychlení oproti softwarovému šifrování neospravedlnilo vysokou cenu přípravku. Na druhou stranu vyšší výkon pro-

<sup>4</sup>Cena podle produktové stránky <https://www.xilinx.com/products/boards-and-kits/1-4azfte.html>, v únoru 2018.

	<b>Raspberry Pi 3</b>	<b>Zybo Zynq-7000</b>
SoC	Broadcom BCM2837	Xilinx Zynq Z-7010
Procesor	ARM Cortex A53 4 jádra, 1.2 GHz	ARM Cortex A9 2 jádra, 650 MHz
RAM	1024 MB LPDDR2	512 MB DDR3
FPGA	–	ekvivalent řady Artix-7
Úložiště	SD karta	128 MB Flash, SD karta
Síť	100 Mb/s Ethernet, WiFi	až 1 Gb/s Ethernet
Cena	1200 Kč	4000 Kč

Tabulka 2.1: Srovnání platform Raspberry Pi 3 a Zybo Zynq-7000 [46, 49].

cesoru u Raspberry Pi může mít pro nadřazený systém význam, například z hlediska odezvy uživatelského rozhraní.

Jako platformu pro implementaci nadřazeného systému jsem zvolil počítač Raspberry Pi 3, především kvůli příznivé ceně, výrazně lepšímu poměru cena/výkon (pro tuto práci), a také kvůli podpoře a rozšiřitelnosti.

Kromě nákladů popsaných v sekcích 2.6.1 a 2.6.2 je ještě potřeba počítat s pořízením GSM modulu. V závislosti na typu modulu se jeho cena může pohybovat kolem 800 až 1200 Kč<sup>5</sup>.

#### 2.6.4 Provoz aplikace na *cloudové* platformě

V této části bych chtěl popsat alternativu k provozu systému na dedikovaném zařízení, a to možnost využít virtuální server na některé *cloudové* platformě. Primární cíl práce je sice vytvořit nadřazený systém jako jednoúčelové zařízení (postavené na Raspberry Pi), nicméně provoz výsledné aplikace v *cloudu* může být v určitých situacích vhodnější řešení.

Jedním z možných uplatnění této varianty je monitorování více garážových komplexů. Místo lokálního nadřazeného systému by mohly všechny *podřízené systémy* z každého komplexu komunikovat s jedním *globálním systémem*, provozovaným na virtuálním serveru a dostupným z internetu, jak je naznačeno na obrázku 2.6.4. Při tomto provozu je však nutnost mít registrovanou doménu a zabezpečit spojení podepsaným certifikátem.

Virtuální servery nabízí například firma DigitalOcean. Cena serveru závisí na počtu výpočetních jader, dostupné RAM a velikosti úložiště. Nejlevnější konfigurace stojí 5 USD měsíčně a nabízí jednojádrový procesor, 1 GB RAM a 25 GB SSD [53]. Předpokládám, že tento výkon by stačil pro základní provoz systému, v případě vyššího počtu podřízených systémů je však možnost zvyšovat dostupnou paměť a přidávat procesorová jádra.

<sup>5</sup>Cena LTE modulu Huawei E3372h na e-shopu <https://www.t-mobile.cz> v březnu 2018 je 1151 Kč.

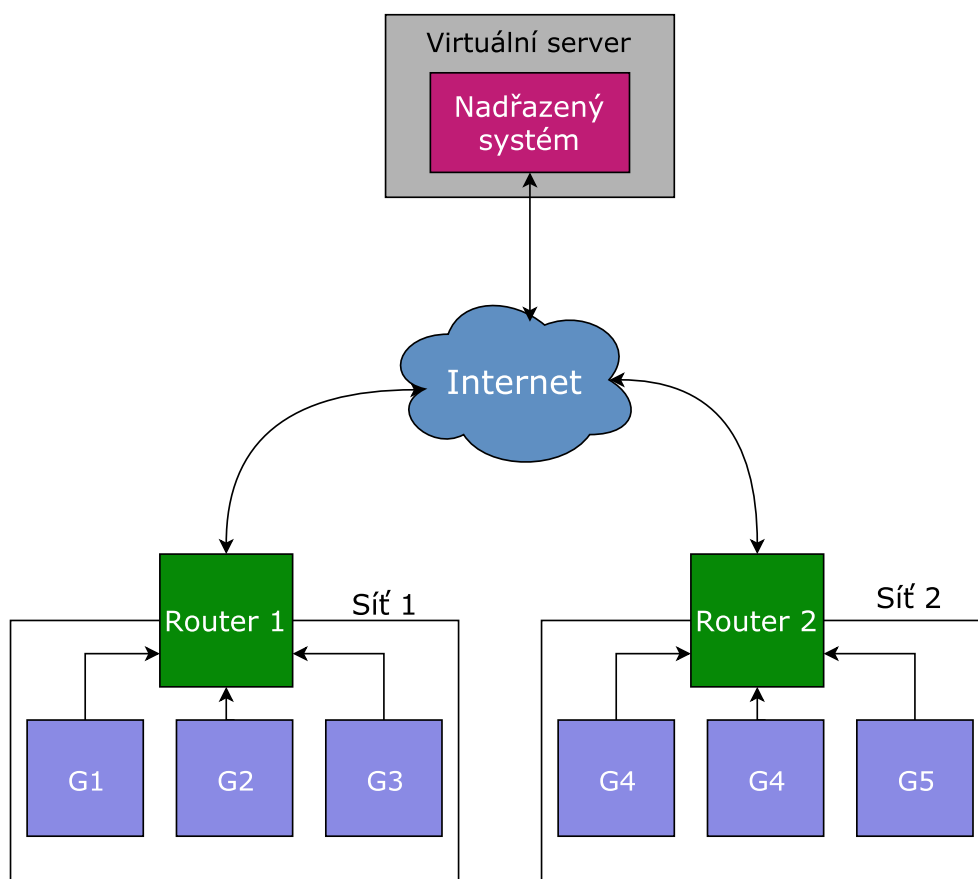
S těmito virtuálními servery lze použít řadu běžných linuxových operačních systémů jako Ubuntu, Debian či Fedora [54], dostupnost softwaru potřebného pro spuštění aplikace tedy není problém.

Hlavní nevýhodou tohoto přístupu je poněkud složitější nasazení a spuštění systému. Registrace domény, získání certifikátu a základní konfigurace webového serveru se dá sice částečně automatizovat (viz zmiňovaný Certbot [15]), pro většinu uživatelů bude však pravděpodobně snazší použít již připravené Raspberry Pi.

Také je složitější distribuovat adresu serveru s nadřazeným systémem podřízeným systémům. V místní síti může podřízený systém sám nalézt nadřazený systém testováním odpovědi zařízení v síti na registrační požadavek. Pokud by však byl nadřazený systém dostupný pouze na internetové doméně, bylo by nutné ji ručně zadat každému podřízenému systému.

Další problém může představovat hlavní výhoda tohoto řešení, a to přístupnost serveru z internetu. Ta významně zvyšuje *attack surface* celého nadřazeného systému, zvláště oproti alternativě využívající k propojení systémů pouze Ethernet, u kterého má (na rozdíl od WiFi) provozovatel fyzický přehled o připojených zařízeních.

Při nasazení na virtuálním serveru také nelze použít GSM modul. Pro posílání upozornění by tak bylo nutné zvolit jiný způsob, například službu Twilio.



Obrázek 2.7: Struktura systému provozovaného na virtuálním serveru. Podřízené systémy nezasílají požadavky lokálnímu nadřazenému systému v místní síti, ale globálnímu nadřazenému systému, který je veřejně přístupný na internetu.

## Návrh

Tato kapitola se zabývá návrhem nadřazeného systému na základě předchozí analýzy z kapitoly 2. Nadřazený systém je navržen podle vzoru MVC, jehož jednotlivé části jsou popsány v sekcích 3.2, 3.3 a 3.4.

Poslední sekce 3.5 se zabývá návrhem autentizace uživatele při přihlašování do webového rozhraní nadřazeného systému.

### 3.1 Návrhový vzor MVC

Struktura nadřazeného systému je vhodná k použití návrhového vzoru MVC, tedy *model-view-controller*. *Model* zde představují garáže (podřízené systémy), k nim vázané události a logika jejich vyhodnocování.

*View* je zobrazení těchto dat, tedy především generované HTML stránky webového rozhraní. Jako další *view* je možné považovat získávání dat (například ve formátu JSON) pomocí API nadřazeného systému, třeba při zasílání registračních klíčů podřízeným systémům.

*Controller* je pak část aplikace, která se stará o zpracování HTTP požadavků. Ty mohou přicházet jednak z uživatelova prohlížeče, jednak od podřízených systémů. Na základě těchto požadavků pak *controller* posílá příslušné příkazy *modelu*. Struktura aplikace při použití vzoru MVC je naznačena na obrázku 3.1.

Hlavní motivací pro použití tohoto vzoru je snadná rozšiřitelnost. Pokud by například bylo potřeba aplikaci doplnit o komunikaci s podřízenými systémy pomocí MQTT, stačí pouze vytvořit vhodný *controller*. Ten pak může využívat *model* aplikace stejným způsobem jako HTTP *controller*.

Tento návrhový vzor popisuje pouze nadřazený systém spravující garáže. Kromě toho je součástí aplikace ještě autentizace uživatele při přístupu do webového rozhraní. Tuto funkci jsem se rozhodl pojmout jako samostatnou komponentu, popsanou v sekci 3.5.



Obrázek 3.1: Struktura MVC aplikace. Uživatel i podřízený systém používají aplikaci pomocí *controlleru*. Ten definuje příslušná URL a zpracovává požadavky na ně – aktualizuje *model*, případně doručí odpovídající *view*.

### 3.2 Komponenta *model*

*Model* představuje jádro nadřazeného systému. Je zde implementována vnitřní reprezentace uchovávaných dat, což jsou především zaznamenané události. Každá událost má také svého původce, tedy garáž (přesněji podřízený systém v této garáži).

Kromě toho *model* implementuje *business logiku* systému, jako je například vytváření nových garáží (registraci podřízených systémů) či reakce na příchozí události.

Ostatní části aplikace (*view* a *controller*) používají *model*, bez znalosti jeho vnitřní struktury, pomocí následujících operací:

- **Vytvoření garáže** – registrace nového podřízeného systému. V případě vytváření pomocí API (tj. přímo podřízeným systémem) vyžaduje ope-

race zapnutý registrační mód. Pokud je nová garáž vytvářena ve webovém rozhraní, zapnutý registrační mód není vyžadován.

- **Editace garáže** – například změna označení.
- **Smazání garáže** – smazáním garáže dojde k odstranění zaznamenaných událostí a zneplatnění příslušného API klíče.
- **Zneplatnění API klíče** – odepření přístupu podřízenému systému bez nutnosti smazání garáže a s ní spojených událostí.
- **Zapínání registračního módu** – v tomto módu je možné vytvářet nové garáže na základě požadavku od podřízeného systému. Bližší informace jsou v sekci 3.2.1.1. Registrační mód se po uplynutí časového limitu sám vypne.
- **Přístup k uloženým datům** – obecně operace typu získání všech garáží nebo všech událostí vázané ke konkrétní garáži.
- **Vytvoření události** – základní operace využívaná podřízenými systémy.

Vnitřně pak model na základě těchto operací autentizuje pomocí API klíčů požadavky podřízených systému, validuje zasláná data, spravuje databázi nadřazeného systému nebo obstarává zasílání notifikací.

### 3.2.1 Garáž – Garage

Třída **Garage** reprezentuje konkrétní podřízený systém a uchovává s ním spojená data:

- **id** – identifikace entity v databázi.
- **tag** – uživatelem zvolené označení garáže. To slouží pro snadnější orientaci ve webovém rozhraní (uživatel nemusí garáže rozlišovat podle nic neříkajícího **id**).
- **note** – poznámka pro další popis garáže.
- **phone** – telefonní číslo nájemce garáže. V případě vyplnění budou upozornění týkající se garáže zasílána i nájemci.
- **api\_key** – klíč umožňující přístup k API systému. Ten je také zaslán zařízení při jeho registraci. Generování a správa klíčů je blíže popsána v sekci 3.2.1.2.
- **last\_report** – datum a čas posledního kontrolního hlášení.
- **next\_report** – datum a čas dalšího očekávaného hlášení.

- **period** – perioda kontrolních hlášení.
- **doors** – stav dveří garáže (otevřeno/zavřeno).
- **state** – celkový stav garáže (v pořádku, nehlásí se atd.).
- Seznam událostí spojených s touto garáží.

#### 3.2.1.1 Vytváření nových garáží

Nové garáže mohou v systému vznikat dvěma způsoby. První možnost je vytvoření nové garáže přímo v uživatelském rozhraní. Po vytvoření se zde zobrazí vygenerovaný API klíč, který je potřeba nahrát na příslušný podřízený systém. Způsob nahrávání by závisel na příslušném hardwaru (například přes sériovou linku). Tato možnost je určena především pro podřízené systémy, které by nepodporovaly zaslání registračního požadavku, a nevyžaduje zapnutý registrační mód.

Druhá možnost je použít registrační mód. V tom případě je nová garáž vytvořena na základě registračního požadavku podřízeného systému (pro bližší informace o tomto požadavku viz sekci 3.4.1). Vygenerovaný API klíč je přitom zaslán jako odpověď na požadavek, a není nutné ho ručně nahrávat. Registrační mód je možné aktivovat v uživatelském rozhraní. Pokud mód není aktivovaný, nadřazený systém odmítne všechny požadavky na registraci zasláné tímto způsobem.

#### 3.2.1.2 API klíče

Podřízené systémy se při zasílání událostí přes API prokazují klíčem. Ten slouží jednak k zamezení příjmu událostí od neautorizovaných systémů, jednak k identifikaci původu události (zdrojové garáže) v rámci nadřazeného systému. Podřízený systém tedy nemusí znát `id` garáže, ale jen `api_key`.

Vzhledem k těmto požadavkům nelze použít jeden univerzální, ale je nutné pro každý registrovaný podřízený systém vygenerovat unikátní klíč. Ten je pak spolu s dalšími záznamy o garáži uložen v databázi systému. K vytváření klíčů jsem se rozhodl použít systém UUID, umožňující generování náhodných klíčů délky 128 bitů, které jsou (pro praktické účely) unikátní [55].

Klíče je možné v uživatelském rozhraní zneplatnit, a tím odeprít přístup zvolenému podřízenému systému. Tuto operaci lze provést dvěma způsoby. V prvním případě lze smazat z databáze celý záznam příslušné garáže. Tím dojde k zneplatnění jejího klíče, ale také k odstranění zaznamenaných událostí.

Pokud si uživatel přeje data o událostech uchovat, může pouze vygenerovat nový API klíč. Přepsáním klíče se opět zneplatní přístup podřízeného systému (který má stále starý klíč), ale uchovají se zaznamenaná data.

Nově vygenerovaný klíč pak uživatel může nahrát na jiný podřízený systém, a tím například nahradit odcizené monitorovací zařízení. V tomto případě



nelze pro nahrání klíče použít registrační mód nadřazeného systému. Ten totiž vždy počítá s vytvořením nové garáže.

Vygenerované klíče jsou v databázi uloženy v čitelné podobě. Klíče by bylo možné před uložením *hashovat*, čímž by při úniku databáze nedošlo k jejich prozrazení. V tom případě by byl klíč v čitelné podobě v uživatelském rozhraní zobrazen pouze jednou, při vytvoření nové garáže. Dále už by byl uchováván jeho *hash*.

Pro ukládání klíčů v čitelné podobě jsem se rozhodl především z důvodů snadného ladění při implementaci nadřazeného a podřízených systémů. Funkci *hashování* klíčů by v případě potřeby neměl být problém doplnit.

### 3.2.2 Událost – Event

Třída **Event** představuje událost zaznamenanou podřízeným systémem. Jak bylo zmíněno v sekci 3.1, jsou tyto události vázány ke konkrétním garážím, kdy má každá událost jednoznačně určeného původce. Každá garáž může mít libovolné množství událostí.

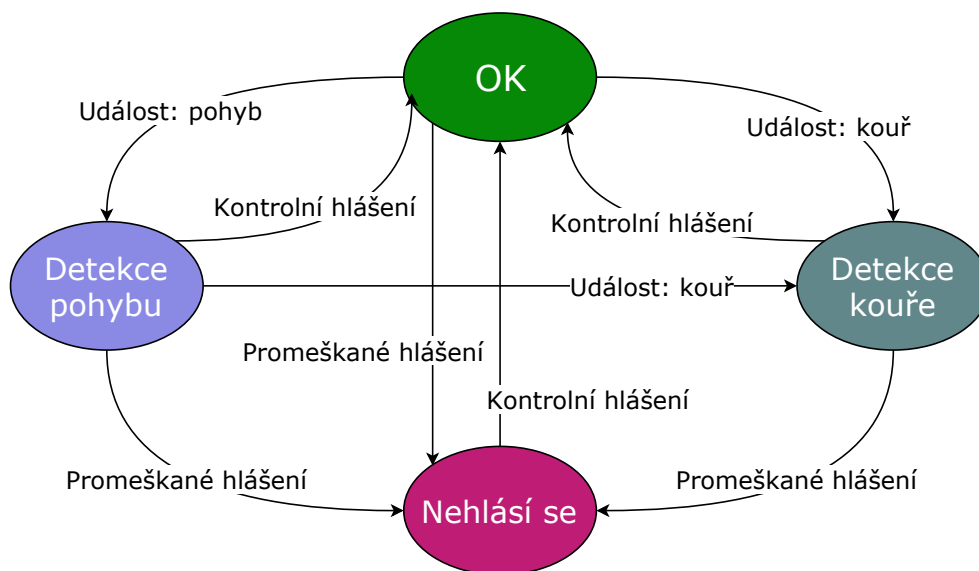
Nadřazený systém rozlišuje dva základní druhy událostí. Kontrolní (plánované) události slouží ke kontrole funkčnosti podřízených systémů. Tyto události jsou odesílány v pravidelném intervalu, určeném nadřazeným systémem. Ten v odpovědi na požadavek s kontrolní událostí zašle očekávaný čas (počet minut) do dalšího hlášení.

Kromě kontrolních hlášení mohou podřízené systémy vytvářet mimořádné události. Mimořádná událost nastane při překročení mezních hodnot některého z čidel (tedy detekce kouře, pohybu či otevření/zavření dveří).

Třída **Event** obsahuje tyto údaje:

- **id** – identifikace entity v databázi.
- **timestamp** – časové razítko.
- **type** – typ zaznamenané události. Nadřazený systém rozeznává následující typy:
  - **report** – kontrolní hlášení.
  - **door\_open** – otevření dveří.
  - **door\_close** – zavření dveří.
  - **movement** – detekce pohybu.
  - **smoke** – detekce kouře.
  - **device\_error** – chyba podřízeného systému. Tato událost není zasílána podřízeným systémem, ale vzniká na základě vyhodnocení zaslaných událostí (více v sekci 3.2.2.1). Událost nesignalizuje výpadek systému, ale jeho neočekávané chování.
- Garáž, která je původcem události.

## 3.2.2.1 Vyhodnocení události



Obrázek 3.2: Diagram vyhodnocování stavu garáže. Nadřazený systém reaguje na příchozí události odpovídající změnou stavu garáže. Změna stavu může být také vyvolána při pravidelné kontrole promeškaných hlášení (přechod do stavu „Nehlásí se“).

Příchozí události a stav garáže jsou nadřazeným systémem vyhodnocovány na základě diagramu na obrázku 3.2. Změna stavu nastává buď v reakci na příchozí událost (například detekci kouře) nebo při pravidelné kontrole zaslaných hlášení.

Pokud při této kontrole nadřazený systém zjistí, že neobdržel očekávané kontrolní hlášení, změní stav garáže na „Nehlásí se“. Tato změna nastane vždy, bez ohledu na předchozí stav. Ze stavu „Nehlásí se“ se lze dostat pouze zasláním kontrolního hlášení.

Kontrolní hlášení funguje jako potvrzení, že jakákoliv nenadálá situace v garáži byla vyřešena. Pokud například podřízený systém detekuje kouř, odešle příslušnou událost a pozdrží další kontrolní hlášení až do doby, kdy bude zdroj kouře odstraněn. Poté může zasláním kontrolního hlášení změnit stav garáže zpět na „OK“.

Kromě událostí měnících stav garáže nadřazený systém ještě rozeznává události otevření a zavření dveří garáže. Tyto události nemění stav garáže, pouze aktualizují stav dveří.

Událost otevření dveří představuje jejich odemčení pomocí klíče. Nesignalizuje tedy neoprávněné vniknutí. Pokud byly dveře odemčeny klíčem, nemá smysl detekovat pohyb.

V případě, že jsou dveře garáže otevřené, by podřízený systém neměl odesílat žádné události informující o detekci pohybu. Pokud tak učiní, je událost nadřazeným systémem ignorována. Místo ní je zaznamenána událost informující o chybě podřízeného systému (`device_error`).

Události o pohybu začne nadřazený systém opět registrovat poté, co je informován o zavření dveří. Neoprávněné vniknutí je tak detekováno zjištěním pohybu v uzavřené garáži.

#### 3.2.2.2 Zasílání upozornění

Na základě příchozích událostí odesílá nadřazený systém SMS s notifikacemi o změnách stavu garáží. SMS jsou zasílány uživateli nadřazeného systému, tedy například správci či majiteli garáží. Tyto zprávy obsahují informace o události, včetně označení garáže, ve které k ní došlo.

Další upozornění jsou zasílána osobám pronajímajícím si jednotlivé garáže (pokud mají v nadřazeném systému vyplněné správné telefonní číslo). V nich je kromě informace o proběhlé události také zobrazeno číslo, na kterém mohou kontaktovat správce garáží.

Oba typy zpráv jsou zasílány vždy při změně stavu garáže (mimo změny do stavu „OK“). Tím je zamezeno zbytečnému zasílání duplicitních SMS.

### 3.3 Komponenta *view*

Hlavním *view* nadřazeného systému je webové uživatelské rozhraní. Zde se zobrazuje celkový stav systému, jednotlivých garáží, a zachycené události. Také jsou zde přístupné ovládací prvky pro správu systému. Návrhem tohoto rozhraní se zabývá následující sekce 3.3.1.

Další *view* aplikace představují odpovědi (například vygenerované API klíče) nadřazeného systému podřízeným systémem při zpracování jejich požadavků. Tento *view* je popsán společně s API systému v sekci 3.4.1.

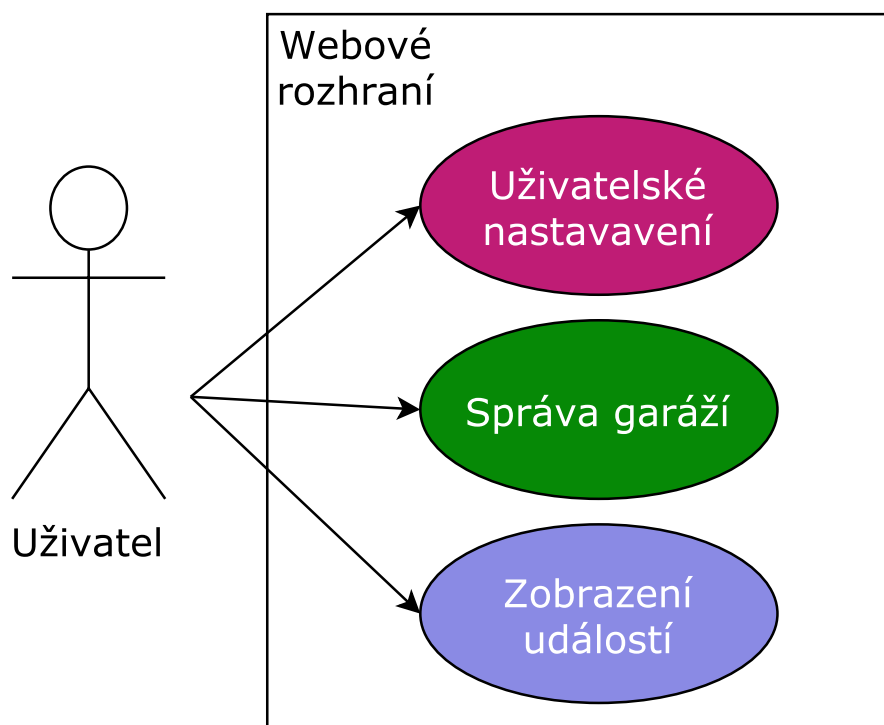
#### 3.3.1 Webové rozhraní

Webové rozhraní představuje prostředí umožňující uživateli správu celého nadřazeného systému. Rozhraní tvoří hlavní stránka, stránky jednotlivých garáží a stránky uživatelského nastavení (změna hesla a nastavení notifikací).

Případy užití rozhraní se dají shrnout do tří kategorií, popsanych na obrázku 3.3.

##### 3.3.1.1 Správa garáží

Správu garáží tvoří případy užití popsané na obrázku 3.4. Operace vytvoření garáže a zapnutí registračního módu nejsou vázané k žádné konkrétní garáži, a jsou tedy k dispozici na hlavní stránce webového rozhraní ihned po přihlášení.



Obrázek 3.3: Vrchní úroveň případů užití webového rozhraní. Uživatel do aplikace přistupuje buď za účelem změny uživatelského nastavení, správy jednotlivých podřízených systémů, případně kvůli kontrole zaznamenaných událostí.

Na hlavní stránce je také zobrazen seznam všech sledovaných garáží včetně jejich stavu, posledního hlášení, dalšího plánovaného hlášení a odkazu na stránku garáže, která obsahuje zbylé možnosti, vázané ke konkrétní garáži.

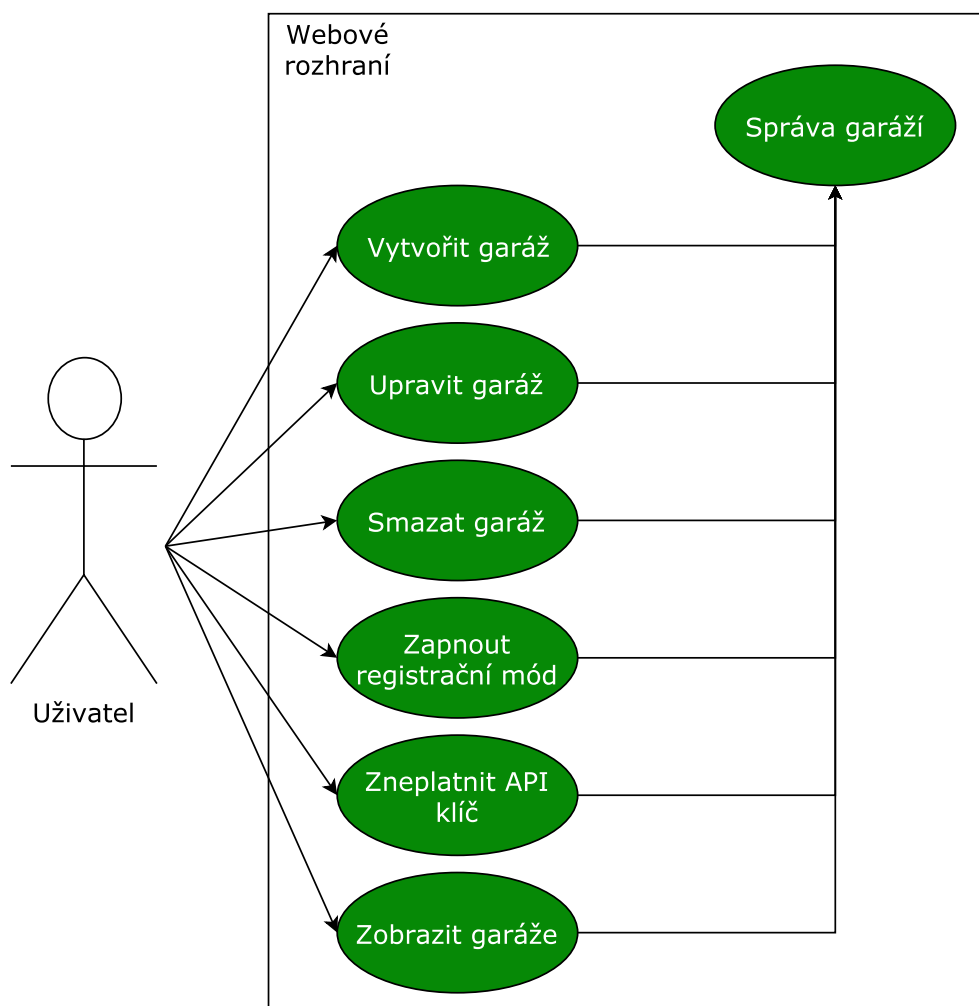
#### 3.3.1.2 Zobrazení událostí

Případy užití z kategorie zobrazení událostí popisuje obrázek 3.5. Na stránce garáže může uživatel zobrazit s ní spojené události. Seznam událostí pak může filtrovat podle typu (viz sekci 3.2.2).

#### 3.3.1.3 Uživatelské nastavení

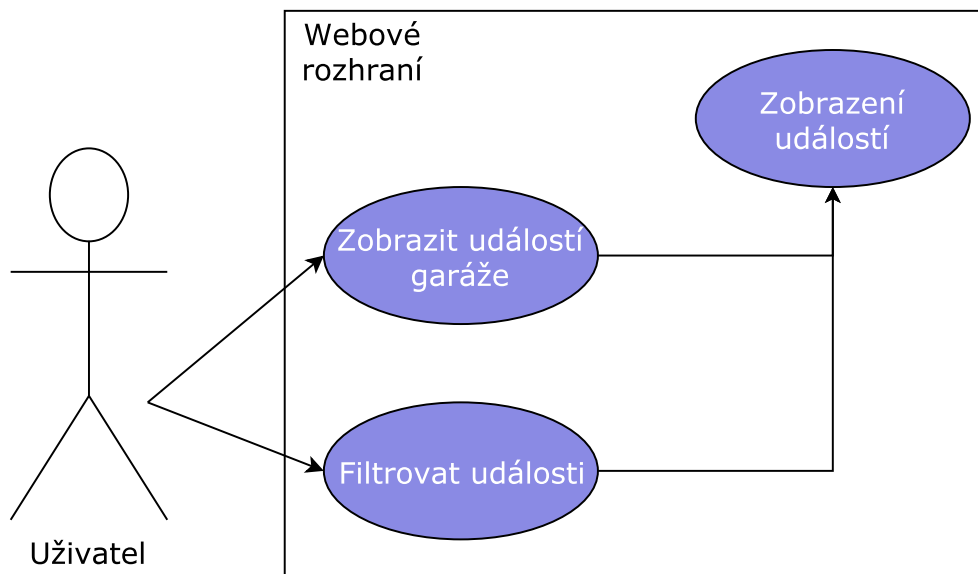
Případy užití uživatelského nastavení popisuje obrázek 3.6. Webové rozhraní umožňuje uživateli změnu přístupového hesla a nastavení telefonního čísla pro zasílání SMS notifikací.

Uživatelské nastavení není uloženo přímo v databázi, ale zvláště v konfiguračním souboru `user_config.ini`. Díky tomu je snazší toto nastavení v případě potřeby změnit i bez nutnosti otvírat webové rozhraní.

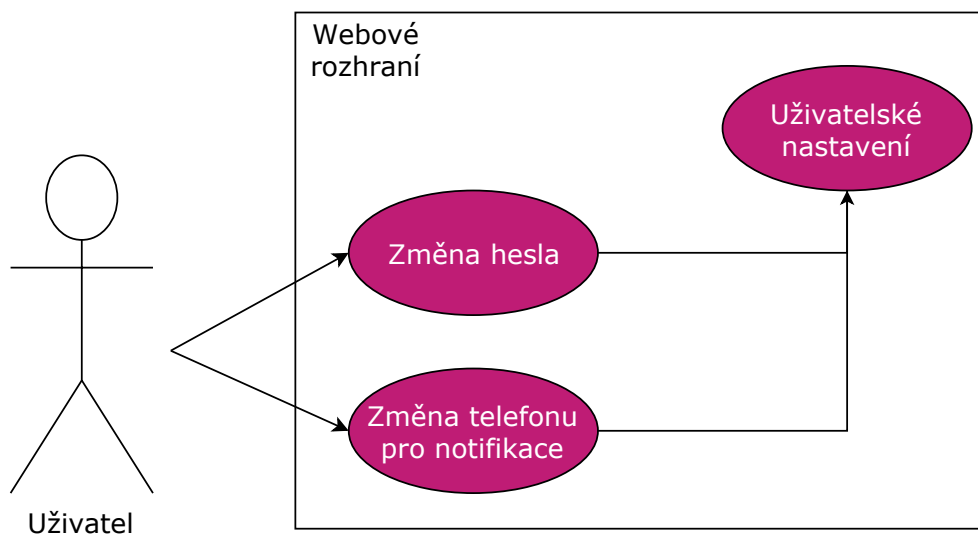


Obrázek 3.4: Případy užití správy garáží. Zde uživatel zobrazuje stav garáží a jednotlivé garáže spravuje pomocí uvedených operací.

V souboru je uložen otisk přístupového hesla (více v sekci 3.5) a telefonní číslo, na které jsou zasílána SMS upozornění o stavu garáží.



Obrázek 3.5: Případy užití zobrazení událostí. Uživatel zobrazuje události spojené s konkrétní garáží. Zobrazené události filtruje podle jejich typu.



Obrázek 3.6: Případy užití uživatelského nastavení. Uživatel ve webovém rozhraní provádí změnu hesla a změnu telefonního čísla určeného k zasílání SMS notifikací.

## 3.4 Komponenta *controller*

*Controller* slouží ke zpracování požadavků uživatele a podřízených systémů. Jelikož jde o HTTP požadavky, definuje *controller* URL, přiřazuje k nim akce, které se mají provést (kontrola přihlášení, přesměrování atd.) a vrací příslušné odpovědi (například vygenerované stránky).

*Controller* aplikace se dá rozdělit na dvě části. První část zpracovává požadavky zaslané uživatelem z webového rozhraní. Propojuje tak případy užití popsané v sekci 3.3.1 s operacemi na modelu ze sekce 3.2. URL definovaná v této části jsou využívána pouze uvnitř aplikace, a nejsou součástí API nadřazeného systému.

Druhá část zpracovává požadavky podřízených systémů. Zde definovaná URL tvoří API nadřazeného systému, pomocí kterého se mohou podřízené systémy registrovat a nahrávat nové události. Popisem API, včetně formátu požadavků a návratových hodnot, se zabývá sekce 3.4.1.

### 3.4.1 API

API nadřazeného systému slouží ke komunikaci s podřízenými systémy. S jeho pomocí se podřízený systém může registrovat a začít nadřazenému systému posílat zaznamenané události.

Odpovědi nadřazeného systému na příchozí požadavky jsou ve formátu JSON. Každá odpověď obsahuje návratový kód a případně další data, která je nutné zaslat podřízenému systému.

#### 3.4.1.1 Registrační požadavek

Registrační požadavek slouží k zaregistrování nového podřízeného systému a vytvoření záznamu s ním spojené garáže. Hlavním účelem tohoto požadavku je snadná distribuce API klíče na podřízené systémy – klíč není nutné nahrávat ručně, ale je odeslán v odpovědi na tento požadavek (pro bližší informace o klících viz sekci 3.2.1.2).

Kvůli bezpečnosti je tento požadavek povolen pouze pokud je v uživatelském rozhraní zapnutý registrační mód. Pokud je mód vypnutý, odpovídá nadřazený systém na všechny registrační požadavky chybovým kódem *403* – *Forbidden*. Tím je částečně zabráněno registraci nežádoucích podřízených systémů.

Registrační požadavek má následující formát (zaslání registračního požadavku v jazyce Python je popsáno v ukázce 2):

- URL – `/api/garages`.
- HTTP metoda – POST.
- Návratová hodnota – JSON:

### 3. NÁVRH

---

- **status** – návratový kód (201 při vytvoření garáže, 403 při vypnutém registračním módu).
- **api\_key** – unikátní klíč k API nadřazeného systému (pouze v případě úspěchu, tedy vytvoření garáže).

```
import requests
# server nadřazeného systému je přístupný
# na adrese raspberrypi.local
url = 'https://raspberrypi.local/api/garages'

# požadavek 'post', bez ověření
# self-signed certifikátu
r = requests.post(url, verify=False)

if r.status == 201:
    api_key = r.json()['api_key']
```

Ukázka 2: Zaslání registračního požadavku nadřazenému systému provozovaném na adrese `raspberrypi.local`.

#### 3.4.1.2 Kontrolní hlášení

Kontrolní hlášení slouží k ověření činnosti podřízeného systému. Jeho zasláním podřízený systém také potvrzuje, že stav sledované garáže je v pořádku.

Požadavek zasílající kontrolní hlášení má tento formát (průběh zaslání požadavku popisuje ukázka 3):

- URL – `/api/report_event`.
- HTTP metoda – POST.
- Hlavička požadavku:
  - **apikey** – unikátní klíč k API nadřazeného systému.
- Návratová hodnota – JSON:
  - **status** – návratový kód (201 při vytvoření hlášení, 403 při neplatném klíči).
  - **period** – počet minut do dalšího očekávaného hlášení.



```

import requests
# server nadřazeného systému je přístupný
# na adrese raspberrypi.local
url = 'https://raspberrypi.local/api/report_event'

# API klíč (získaný například reg. požadavkem)
# je zasílán v hlavičce požadavku
headers = {'apikey' : api_key}

# požadavek 'post', bez ověření
# self-signed certifikátu
r = requests.post(url, headers=headers, verify=False)

if r.status == 201:
    # odpověď -- čas do dalšího hlášení
    period = r.json()['period']

```

Ukázka 3: Zaslání kontrolního hlášení nadřazenému systému provozovaném na adrese `raspberrypi.local`

#### 3.4.1.3 Ostatní události

Vytváření dalších událostí je stejné jako vytváření kontrolního hlášení. Liší se pouze použité URL. Odpovědi na tyto požadavky neobsahují žádná data od nadřazeného systému, pouze návratový kód informující o úspěchu či chybě (viz ukázku 4):

- URL – podle typu události:
  - `/api/door_open_event`
  - `/api/door_close_event`
  - `/api/movement_event`
  - `/api/smoke_event`
- HTTP metoda – POST.
- Hlavička požadavku:
  - `apikey` – unikátní klíč k API nadřazeného systému.
- Návratová hodnota – JSON:
  - `status` – návratový kód (201 při vytvoření události, 403 při neplatném klíči).

```
import requests

# stejné jako při zasílání kontrolního hlášení
# liší se pouze url
url = 'https://raspberrypi.local/api/smoke_event'
headers = {'apikey' : api_key}

r = requests.post(url, headers=headers, verify=False)

if r.status == 201:
    # odpověď neobsahuje žádná data
    ...
```

Ukázka 4: Zaslání události detekce kouře nadřazenému systému provozovaném na adrese `raspberrypi.local`.

## 3.5 Autentizace uživatele

Do webového rozhraní aplikace je povolen pouze autorizovaný přístup, a je tedy nutné nějakým způsobem ověřit identitu uživatele. Jelikož aplikace nepočítá s odlišnými stupni přístupu (každý uživatel s přístupem do rozhraní má přístup ke všem jeho částem), rozhodl jsem se, že nebudu vytvářet infrastrukturu uživatelských účtů, protože by měla pouze omezené využití.

Přístup do rozhraní nevyžaduje vytvoření účtu a zadávání uživatelského jména, ale pouze heslo. *Hash* tohoto hesla (doplněného solí), je uložen v souboru s uživatelským nastavením aplikace. K vytvoření *hashe* je použit algoritmus Argon2 [56].

*Hashování* hesla je zde použito především pro případ, že by uživatel použil stejné heslo jaké používá v jiných aplikacích či službách. V případě prozrazení uživatelských dat (například při odcizení systému) útočník nemůže z otisku zjistit původní použité heslo, tudíž přístup k těmto uživatelským účtům není kompromitován.

Stejně tak někdo s přístupem k serveru, na kterém nadřazený systém běží, nemá automaticky přístup do webového rozhraní. Nicméně má stále přístup k celé (nešifrované) databázi a také může heslo libovolně měnit (spočítáním nového otisku), takže z praktického hlediska je přístup k serveru téměř ekvivalentní s přístupem do webového rozhraní.

### 3.5.1 První přihlášení

Při prvním startu nadřazeného systému je zvoleno jednoduché implicitní heslo („password“), kterým se uživatel může přihlásit do webového rozhraní.

Pokud nadřazený systém detekuje použití tohoto hesla, po přihášení uživatele ihned přesměruje na stránku umožňující změnu hesla a zobrazí varování.

## Implementace

V této kapitole je popsána implementace nadřazeného systému podle návrhu z kapitoly 3. Jednotlivé sekce popisují implementační detaily dílčích částí návrhového vzoru MVC. Poslední sekce 4.5 se zabývá implementací autentizace uživatele.

### 4.1 Struktura aplikace

Aplikace nadřazeného systému je rozdělena do tří modulů:

- `mod_main` – hlavní modul aplikace. V tomto modulu je implementován *model* systému popsáný v sekci 3.2. *Model* je využíván i ostatními moduly (konkrétně modulem `mod_api`). Dále tento modul implementuje webové rozhraní správy nadřazeného systému.
- `mod_api` – modul implementující API systému. Zde je implementována část *controlleru* ze sekce 3.4.1, definující URL, pomocí kterých mohou podřízené systémy komunikovat s nadřazeným systémem.
- `mod_auth` – modul implementující autentizaci uživatele při přihlašování do webového rozhraní.

K rozdělení aplikace na jednotlivé moduly je použit nástroj frameworku Flask nazvaný *blueprints* [57]. Hlavní důvod k využití modulů je oddělení částí aplikace podle jejich funkce.

Při strukturování aplikace jsem vycházel z článku *How To Structure Large Flask Applications* [58].

### 4.2 Implementace *modelu*

Jak je zmíněno v sekci 3.2, *model* aplikace je tvořen třídami `Garage` a `Event`. Tyto třídy přímo využívají databázi nadřazeného systému, k jejich implemen-

taci je proto použit framework SQLAlchemy [59], který výrazně usnadní práci s databází.

Tento framework poskytuje přístup k SQL databázím přímo z jazyka Python, takže není nutné psát prakticky žádný SQL kód. Tabulku v databázi je možné definovat jako Python třídu s příslušnými atributy a SQLAlchemy vytvoří odpovídající databázové schéma.

SQLAlchemy také umožňuje snadno definovat databázové vztahy. V *modelu* nadřazeného systému se vyskytuje pouze vztah 1 : *N* mezi třídami **Garage** a **Event** (jedna garáž má mnoho událostí, každá událost má právě jednu garáž). Tento vztah je definován způsobem naznačeným v ukázce 5.

```
from sqlalchemy import Column, ForeignKey, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

# třídy Event a Garage
# dědí SQLAlchemy metody
# třídy Base
class Event(Base):
    ...
    # odkaz na příslušnou garáž
    garage_id = Column(Integer,
                        ForeignKey('Garage.id'),
                        nullable=False)

class Garage(Base):
    ...
    # definice 1:N vztahu mezi garáží a událostí
    events = relationship('Event',
                          backref='Garage',
                          lazy='dynamic')
```

Ukázka 5: Vytvoření vztahu 1 : *N* mezi třídami **Garage** a **Event**.

#### 4.2.1 Dynamické načítání databázových vztahů

Definice vztahu mezi garáží a jejími událostmi je na straně třídy **Garage** provedena pomocí funkce `relationship()` (viz ukázkou 5). Zde se při testování (sekce 5.4) ukázalo, že výchozí chování této funkce není pro účely nadřazeného systému příliš vhodné.

Pokud není uvedeno jinak, funkce `relationship()` vytvoří na základě databázových záznamů seznam událostí pomocí datového typu jazyka Python

`list` [60]. Proměnná `events` by tak v tomto případě představovala `list` obsahující všechny události.

Toto chování dovoluje pohodlnou a intuitivní práci s uloženými daty, neboť je zachována sémantika Pythonu. Nehodí se však pro velké množství záznamů. Problém představuje právě načítání záznamů do datové struktury.

Při přidávání nové události je nutné nejprve načíst všechny dosud zaznamené události garáže do seznamu `events`, poté na jeho konec vložit novou událost pomocí metody `events.append()` a výsledek opět uložit do databáze. Seznam může být udržován v paměti od předchozího vkládání, nicméně samotné vložení nového prvku do dlouhého seznamu nemusí být efektivní.

SQLAlchemy nabízí několik řešení, jak definovat vztah pomocí funkce `relationship()` efektivněji [60]. Já jsem zvolil použití dynamického načítání záznamů (`lazy='dynamic'` v ukázce 5). V tom případě přístup k proměnné `events` nevrátí `list`, ale instanci třídy `Query` [60].

S její pomocí lze přidávat nové události metodou `append()` bez nutnosti načítání těch dosud zaznamenaných, a také přistupovat k uloženým událostem [60].

Použití dynamického načítání zaznamenaných událostí má poměrně výrazný vliv na výkon aplikace při komunikaci s podřízenými systémy (protože právě ty vkládají nové události).

### 4.2.2 Kontrola zmeškaných hlášení

V databázi nadřazeného systému je potřeba pravidelně provádět kontrolu, zda podřízené systémy zaslaly v očekávaný čas kontrolní hlášení. Pokud bylo plánované hlášení promeškáno, je nutné změnit stav příslušné garáže na „Nehlásí se“.

Aplikace nadřazeného systému nemá v zásadě možnost, jak tuto kontrolu sama iniciovat, neboť pouze reaguje na příchozí požadavky (od uživatele či podřízeného systému). Provedení kontroly může být důsledkem takového požadavku, například pokud uživatel otevře hlavní stránku webového rozhraní.

Provádět kontrolu hlášení pouze v reakci na vnější podnět však není dostatečné. Pokud by nadřazený systém musel pro provedení kontroly hlášení čekat interakci s webovým rozhraním (nebo například s podřízeným systémem), nemuselo by vůbec dojít ke změně stavu garáže a odeslání upozornění. Je proto nutné zajistit pravidelné provádění kontrol na základě vnitřního podnětu.

Tento problém jsem vyřešil použitím plánovače APScheduler [61]. APScheduler funguje jako knihovna do Pythonu, a umožňuje plánovat provádění zvolených funkcí. Nejde tedy o externí program, plánovač je přímo součástí kódu nadřazeného systému [61]. Vytvoření pravidelné kontroly hlášení podřízených systémů je naznačeno v ukázce 6.

Plánovač se kromě kontroly hlášení hodí také při vypínání registračního módu. Ten z bezpečnostních důvodů po aktivaci běží po dobu tří minut. Jeho

```
from apscheduler.schedulers.background import \
    BackgroundScheduler

# BackgroundScheduler běží v samostatném vlákne,
# neblokuje tedy webovou aplikaci
scheduler = BackgroundScheduler()

# přidání pravidelného úkolu do plánovače
# Garage.check_reports je statická metoda
# třídy Garage, která provede kontrolu hlášení
# u všech garáží v databázi (a případně upraví jejich stav)
scheduler.add_job(Garage.check_reports, 'interval', minutes=5)
scheduler.start()
```

Ukázka 6: Pravidelná kontrola hlášení podřízených systémů pomocí knihovny APScheduler. Po startu plánovače je každých 5 minut volána metoda `Garage.check_reports()`.

vypnutí je naplánováno obdobně jako kontrola hlášení, jediný rozdíl je, že úkol není spouštěn v pravidelném intervalu, ale pouze jednou.

### 4.2.3 Zasílání upozornění

Nadřazený systém zasílá SMS upozornění při změně stavu garáže (kromě změny do stavu „OK“, viz sekci 3.2.2.1). Změnu stavu je možné hlídat pomocí dekorátoru `@listens_for` z frameworku SQLAlchemy [62], jak je naznačeno v ukázce 7. Dekorátor je konstrukt jazyka Python, který umožňuje rozšířit danou funkci či metodu o další funkcionalitu [63].

K odesílání SMS pomocí GSM modulu je použit program Gammu. S ním lze odesílat zprávy pomocí příkazové řádky a příkazu `gammu sendsms` [64]. Tento přístup však není příliš vhodný.

Jelikož by byl příkaz volán přímo z kódu aplikace nadřazeného systému, bylo by nutné tuto aplikaci spouštět s vhodnými přístupovými právy, protože je vyžadován přístup k sériové lince. Příkaz je navíc blokující – po jeho spuštění je nutné čekat na odeslání SMS a odpověď mobilní sítě. To by zbytečně zpomalovalo zasílání událostí, které vyvolají změnu garáže a odeslání upozornění.

Pro odesílání SMS je vhodnější využít démona Gammu SMSD, který běží na pozadí a v pravidelných intervalech odesílá všechny SMS přidáné do jeho fronty pomocí příkazu `gammu-smsd-inject` [64].

Přidání zprávy do fronty je provedeno okamžitě, není nutné čekat na reakci GSM modulu či sítě. Pro přidání také nejsou potřeba žádná speciální práva, stačí pouze přístup ke složce, která slouží jako fronta pro zprávy k odeslání [64].

```

from sqlalchemy.event import listens_for

# funkce se provede při nastavení
# atributu state třídy Garage
@listens_for(Garage.state, 'set', named=True)
def send_notification(**kwargs):
    # pokud nedošlo ke změně hodnoty state
    # není třeba provádět žádnou akci
    if kwargs['value'] == kwargs['oldvalue']:
        return
    # stejně tak pokud je nový stav OK
    if kwargs['value'] == Garage.STATE_OK:
        return

    # odeslání SMS na čísla
    # nájemce a správce garáží

```

Ukázka 7: Odeslání SMS na základě změny stavu garáže. Změna stavu je zachycena pomocí dekorátoru `@listens_for`. Jelikož `@listens_for` reaguje pouze na nastavení hodnoty `Garage.state`, je ještě nutné provést kontrolu, zda opravdu došlo ke změně stavu.

## 4.3 Implementace *view*

Tato část se věnuje implementaci stránek webového rozhraní nadřazeného systému. Jednotlivé HTML stránky rozhraní jsou generovány pomocí šablonovacího systému Jinja [65], který je součástí frameworku Flask [66].

Jinja umožňuje dynamické generování HTML stránek z předem vytvořených šablon a dalších vstupů, v tomto případě především dat z *modelu* aplikace. Tato data jsou šabloně předána *controllerem* (viz sekci 4.4). V šablonách Jinja používá syntaxi založenou na Pythonu [65]. Příklad HTML šablony pro tabulku se zaznamenanými událostmi zobrazuje ukázka 8.

Jinja také dovoluje definici filtrů sloužících ke zpracování zobrazovaných dat [65]. V ukázce 8 je použit filtr `event_filter`. Ten dostane jako vstup instanci třídy `event` a vytvoří její textovou reprezentaci s údaji o typu a času události. Tím je oddělena implementace této třídy v *modelu* a její zobrazení ve *view*.

### 4.3.1 Formuláře

Webové rozhraní obsahuje několik formulářů, které slouží například k editaci garáže či změně hesla. Tyto formuláře jsou implementovány pomocí frameworku WTForms, který je opět integrován ve Flasku [67].

Framework poskytuje třídu `FlaskForm`, pomocí které je možné vytvořit

```
<table class=basic_table>
  <tr>
    <th>Seznam událostí</th>
  </tr>
  <!-- použití cyklu v systému Jinja -->
  {% for event in garage.events %}
    <tr>
      <!-- přístup k proměnné pomocí bloku {{ }} -->
      <td class=event_type_{{event.type}}>
        {{ event | event_filter }}
      </td>
    </tr>
  {% endfor %} <!--konec bloku cyklu -->
</table>
```

Ukázka 8: HTML šablona tabulky zaznamenaných událostí, využívající šablonovací systém Jinja. Proměnná `garage` je šabloně předána *controllerem* aplikace.

podtřídy popisující jednotlivé formuláře [67]. V těchto podtřídách jsou pak definovány pole formuláře.

U některých polí je potřeba provádět kontrolu zadávaných hodnot (například dodržení minimální délky hesla). To je provedeno pomocí další součásti WTForms, nazvané validátory [67]. Pomocí těchto předdefinovaných validátorů lze na vstupní data aplikovat nejrůznější omezení (délka řetězce, rozsah číselných hodnot atd.) a určit chybovou zprávu zobrazenou při jejich porušení [67].

Využití tohoto frameworku při implementaci formuláře pro editaci údajů garáže popisuje ukázka 9. Vygenerování HTML kódu formuláře, včetně zobrazení chybových zpráv po odeslání neplatného formuláře, je provedeno pomocí systému Jinja.

## 4.4 Implementace *controlleru*

*Controller* aplikace je implementován spárováním URL a funkce, která se má provést při příchozím požadavku na tuto URL. Jako odpověď na požadavek je pak klientovi zaslána návratová hodnota funkce. Tento princip (demonstrováný v ukázce 10) představuje základní stavební kámen při tvorbě *controlleru* ve frameworku Flask.

Pro spárování funkce a URL je použit dekorátor `@route`. Pomocí dekorátoru `@route` je definováno URL a případně i povolené HTTP metody, které může klient při požadavku použít. Implicitně je povolena pouze metoda `get` [69].



```

from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField, TextAreaField
from wtforms.validators import NumberRange

...

class GarageForm(FlaskForm):
    tag = StringField('Označení')
    period = IntegerField('Perioda hlášení (minuty)',
        validators=[
            NumberRange(1, 999,
                message='Perioda musí být mezi 1 a 999')
        ])
    phone = StringField('Telefonní číslo',
        validators=[
            validate_phone_number
        ])
    note = TextAreaField('Poznámka')

```

Ukázka 9: Implementace formuláře pro editaci údajů garáže pomocí frameworku WTForms. Při kontrole vstupu je ověřen rozsah zadávané periody a telefonní číslo. Platnost telefonního čísla je ověřena pomocí knihovny *Phone-numbers* [68].

Pokud funkce zpracovávající požadavek mění obsah databáze (například vytváří novou garáž), je u požadavku vyžadována metoda `post`. Tyto požadavky jsou také chráněny proti CSRF útoku. Do každého formuláře používajícího metodu `post` je vložen náhodně generovaný *token*, podle kterého může aplikace ověřit původ požadavku. Ochrana proti CSRF je automaticky zapnuta u všech formulářů vytvořených pomocí WTForms (viz sekci 4.3.1) [67].

#### 4.4.1 Implementace API

Způsobem popsaným v sekci 4.4 je implementováno i API nadřazeného systému. Operace a URL ze sekce 3.4.1 jsou spárovány s příslušnými funkcemi upravujícími *model* aplikace.

Jelikož se podřízené systémy prokazují pomocí API klíče, není zde nutné provádět žádnou kontrolu přihlášení jako při zpracování požadavků ve webovém rozhraní. Pouze je zkontrolována přítomnost zaslání klíče v databázi. Také není nutné implementovat ochranu proti CSRF útokům.

```
from flask import Blueprint, render_template

from .models.garage import Garage

# deklarace blueprintu mod_main
mod_main = Blueprint('main', __name__)

# URL a příslušná akce jsou definovány
# vzhledem k blueprintu
@mod_main.route('/')
@login_required # kontrola přihlášení uživatele
def index():
    # získání údajů o všech garážích
    garages = Garage.query.all()

    # vygenerování stránky pomocí Jinja
    # zobrazovaná data jsou předána jako
    # parametry funkce render_template()
    return render_template('main/index.html',
                           garages=garages,
                           reg_mode=Garage.reg_mode)
```

Ukázka 10: Přiřazení funkce k URL. Funkce `index` bude zavolána při každém příchozím požadavku s metodou `get` na kořenové URL `/`. Návrátová hodnota funkce je vygenerovaná HTML stránka, která bude zaslána klientovi.

## 4.5 Autentizace a přihlášení uživatele

Autentizace uživatele je provedena zadáním hesla. Heslo je zadáváno pomocí formuláře na přihlašovací stránce a posíláno požadavkem `post`. Použití metody `get` není vhodné, neboť heslo jako součást URL zůstane uchované jednak v historii prohlížeče, jednak v záznamu požadavků na serveru.

Pokud uživatel poskytl správné heslo, je v jeho HTTP relaci nastavena proměnná `logged_in` na hodnotu `true`. Tato proměnná slouží ke kontrole přihlášení uživatele při přístupu k dalším částem webového rozhraní. Kód provádějící kontrolu je vkládán do funkcí pomocí dekorátoru `@login_required`, jehož použití lze vidět v ukázce 10 (bližší informace o dekorátorech je možno najít v sekci 4.4). Odhlášení je provedeno nastavením hodnoty `logged_in` na `false`.

## 4.6 Konfigurace aplikace

Aplikaci je nutné před spuštěním nakonfigurovat. K tomu je využit konfigurační soubor. Zde je definováno například umístění databázového souboru

(všechny možnosti nastavení aplikace vytvořené ve frameworku Flask popisuje dokumentace [70]).

Konfigurační soubor může provozovatel nadřazeného systému určit pomocí proměnné prostředí `GARAGE_SYSTEM_CONFIG`. Pokud tato proměnná není nastavena, je použita implicitní konfigurace.



# Testování

Tato kapitola popisuje kroky provedené při testování nadřazeného systému. Nejprve byla připravena sada automatizovaných testů pro ověření základní funkčnosti aplikace (sekce 5.1).

V další fázi testování byla aplikace nasazena na virtuálním serveru (sekce 5.2) a na Raspberry Pi 3 (sekce 5.4), na kterém byla testována odezva systému.

V rámci testování byl také vytvořen simulátor podřízeného systému (sekce 5.3).

## 5.1 Automatizované testy

K aplikaci jsem vytvořil sadu automatizovaných testů, které ověřují funkčnost jednotlivých komponent. Konkrétně jde o testy pro *model* aplikace (zde je testována především třída **Garage**), webové rozhraní a API systému. Tyto testy byly implementovány pomocí frameworku pytest [71].

Části aplikace, u kterých je automatizované testování problematičtější, jako například funkce využívající APScheduler (viz sekci 4.2.2) nebo zasílání notifikací, jsem otestoval ručně.

### 5.1.1 Testovací konfigurace

Před spuštěním automatizovaných testů je nutné upravit konfiguraci aplikace. Především je třeba určit umístění testovací databáze a souboru s uživatelským nastavením. Tyto soubory jsou v průběhu testů upravovány a po dokončení smazány.

Také je nutné vypnout ochranu proti CSRF v celé aplikaci, což výrazně zjednoduší testování formulářů aplikace. Jelikož požadavky s metodou **post** vytvořené v testech nepocházejí z webových stránek aplikace, nemohou dodat CSRF *token*, který je součástí formuláře.

Sdílení *tokenu* mezi aplikací a testy by sice bylo možné implementovat, bylo by však zbytečně komplikované. Za vhodnější postup považuji vypnutí

CSRF ochrany pro většinu testů a funkčnost ochrany otestovat v samostatném testu (viz sekci 5.1.6).

Změna konfigurace aplikace je prováděna nastavením proměnné prostředí `GARAGE_SYSTEM_CONFIG` na soubor s testovací konfigurací (viz sekci 4.6).

### 5.1.2 Testování *modelu*

Testování *modelu* aplikace spočívá především v testování metod třídy `Garage`. Třída `Event` slouží pouze k uchování dat o události a neimplementuje žádné metody, které by bylo možné testovat.

Testovány jsou následující scénáře:

- Přidání garáže.
- Smazání garáže.
- Kontrola zmeškaných hlášení.
- Zaslání kontrolního hlášení.
- Změna stavu po zaslání události.
- Zneplatnění API klíče.
- Získání zaznamenaných událostí podle typu.

Zde stojí za zmínku test kontroly zmeškaných hlášení. V něm je garáži zasláno kontrolní hlášení, čímž je určen čas dalšího očekávaného hlášení. Po překročení tohoto času je potřeba otestovat, zda byl stav garáže změněn na „Nehlásí se“.

Jelikož je minimální perioda hlášení 1 minuta, prosté čekání (například pomocí funkce `sleep()`) by proces testování neúnosně zpomalilo (pro srovnání spuštění všech testů aplikace na běžném počítači zabere asi 4 sekundy). Vhodnější je tedy časový posun nasimulovat.

K tomu je použita knihovna `FreezeGun` [72]. Tato knihovna umožňuje explicitně nastavit datum a čas, které vrátí funkce `now()`, používaná v implementaci třídy `Garage`. Použití knihovny při testování kontroly promeškaných hlášení je demonstrováno v ukázce 11.

### 5.1.3 Testování API

Testy API nadřazeného systému ověřují funkčnost následujících operací:

- Vytvoření garáže při vypnutém registračním módu.
- Zaslání události.
- Zaslání události s neplatným API klíčem.

```

from freezegun import freeze_time

# všechna volání datetime.datetime.now()
# pocházející z této funkce vrátí
# hodnotu 2011-01-01 00:00:00
@freeze_time("2011-01-01 00:00:00")
def test_check_report(garage):
    new_garage = garage.add_garage()
    new_garage.period = 60 # explicitní nastavení periody
    new_garage.add_report_event()
    new_garage.check_report()

    assert new_garage.state == garage.STATE_OK

    # nastavení návratové hodnoty funkce now()
    # v rámci bloku with
    with freeze_time("2011-01-01 02:00:00"):
        new_garage.check_report()
        assert new_garage.state == garage.STATE_NOT_RESPONDING

```

Ukázka 11: Test kontroly promeškaných hlášení. Pomocí knihovny FreezeGun je čas nastaven na půlnoc 1. 1. 2011. Poté je čas posunut o dvě hodiny a otestována změna stavu garáže.

U každého testu je zkontrolováno, zda byla odeslána správná odpověď na příslušný požadavek – návratový kód (například *404 – Forbidden*) a zasláná data. Je tedy testována pouze reakce API *controlleru*, vliv operací na databázi nadřazeného systému je pokryt v testu *modelu* (viz sekci 5.1.2).

#### 5.1.4 Testování webového rozhraní

V těchto testech je zkoumán *controller* webového rozhraní a zobrazované HTML stránky. Testovány jsou tyto scénáře:

- Zobrazení hlavní stránky.
- Zobrazení stránky garáže.
- Zobrazení událostí.
- Zobrazení chybové stránky při zadání neexistující garáže.
- Filtrování událostí.
- Úprava dat garáže (označení, poznámka a telefonní číslo), včetně testu ověření neplatných hodnot.

- Úprava uživatelského nastavení (telefonní číslo pro zasílání upozornění).

V každém testu je ověřováno, zda aplikace v reakci na požadavek vygeneruje odpovídající webovou stránku a vrátí odpovídající návratový kód.

Zobrazení testovaných stránek vyžaduje přihlášení do webového rozhraní nadřazeného systému. To by vyžadovalo zaslání přihlašovacího požadavku se správným heslem před spuštěním testů.

Další možnost je nastavit přihlášení explicitně, modifikací testovací HTTP relace. Proto není nutné spoléhat na funkcionality přihlašování, která je testována samostatně.

Framework Flask umožňuje k aplikaci vygenerovat testovacího klienta, jehož relaci je možné libovolně modifikovat [73]. Díky tomu lze nastavit proměnnou `logged_in`, která je používána ke kontrole přihlášení (viz sekci 4.5). Nastavení této proměnné je provedeno způsobem naznačeným v ukázce 12.

```
# app_client je testovací klient Flask aplikace
# získaný pomocí metody test_client()
def set_logged_in(app_client, value):
    with app_client as c:
        with c.session_transaction() as s:
            s['logged_in'] = value
```

Ukázka 12: Explicitní nastavení proměnné `logged_in` na požadovanou hodnotu. Pomocí takto modifikovaného testovacího klienta lze zasílat požadavky, na které bude aplikace reagovat jako při přihlášení.

### 5.1.5 Testování autentizace uživatele

Toto testování ověřuje jednak funkci *controlleru* provádějícího autentizaci, jednak přístup k souboru s uloženým otiskem hesla. Jsou zde testovány tyto scénáře:

- Zobrazení přihlašovací stránky.
- Kontrola přihlášení při přístupu k dalším částem systému (viz sekci 4.5).
- Přihlášení pomocí implicitního hesla.
- Zadání neplatného hesla.
- Odhlášení.
- Změna hesla.

Opět jsou kontrolovány návratové kódy odpovědí. Také je testováno, zda byla provedena příslušná přesměrování (například přesměrování na přihlašovací stránku při pokusu o přístup k částem rozhraní bez přihlášení) a zobrazení chybových hlášek (například při zadání neplatného hesla).



### 5.1.6 Testování ochrany proti CSRF

Jelikož byla u předchozích testů vypnuta ochrana proti CSRF, je vhodné provést ještě jednoduchý test, který ověří její fungování. Zde stačí zaslat testovanému formuláři data pomocí metody `post` bez platného CSRF *tokenu*.

Výchozí chování CSRF ochrany ve frameworku Flask je v případě chybného nebo neplatného *tokenu* zaslat návratový kód *400 – Bad request* [67]. V těle odpovědi je pak informace o problému s *tokenem*. Stačí tedy otestovat návratový kód a obsah odpovědi (viz ukázkou 13).

```
def test_csrf():
    ...
    # zapnutí ochrany proti CSRF
    app.config['WTF_CSRF_ENABLED'] = True
    app.config['WTF_CSRF_CHECK_DEFAULT'] = True

    # zaslání požadavku na přihlášení bez platného tokenu
    response = app.test_client().post('/login', data={
        # na zaslání hesle nezáleží
        # žádost bude odmítnuta na základě neplatného tokenu
        'password' : 'test password',
        'csrf_token' : 'some fake token'
    })

    assert response.status == '400 BAD REQUEST'
    # odpověď by měla zmiňovat CSRF
    assert 'CSRF' in response.data.decode('utf-8')
```

Ukázka 13: Test ochrany proti CSRF.

## 5.2 Testovací nasazení aplikace

V rámci testování jsem se rozhodl nasadit aplikaci nadřazeného systému na virtuálním serveru, způsobem popsáním v sekci 2.6.4. Tím je jednak otestován průběh nasazování aplikace, jednak je běžící aplikaci možné použít pro demonstrační účely. Aplikace je momentálně<sup>6</sup> přístupná na doméně `https://demo-garaze.tk`.

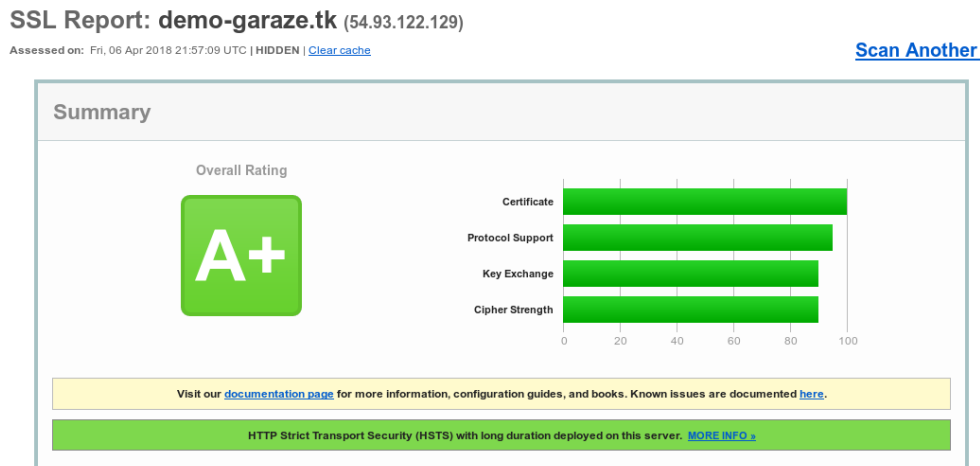
Jelikož je aplikace dostupná z internetu, není vhodné použít k provozu HTTPS *self-signed* certifikát. Místo toho je k doméně registrován platný certifikát autority Let's Encrypt.

Aplikace k provozu využívá server Apache2. Při konfiguraci HTTPS na tomto serveru jsem vycházel z článku *Strong SSL Security on Apache2* [74].

<sup>6</sup>Aplikace byla spuštěna 30. 3. 2018. Jelikož je registrace domény platná do 30. 6. 2018, bude její provoz po tomto datu ukončen.

## 5. TESTOVÁNÍ

Správnost konfigurace a platnost certifikátu byla ověřena pomocí testu poskytovaného společností Qualys. Výsledné hodnocení je možné vidět na obrázku 5.1.



Obrázek 5.1: Výsledky testu HTTPS konfigurace poskytnutného společností Qualys.

Nasazení aplikace v reálném prostředí odhalilo několik chyb, které nebyly při lokálním testování patrné. V první řadě se ukázalo, že server Apache2 od verze 2.4 ignoruje v příchozím požadavku hlavičky, které obsahují podtržítka [75]. Podtržítka byla původně použita při zasílání hlavičky s API klíčem (původně `api_key`). Tuto chybu jsem opravil přejmenováním hlavičky na `apikey`.

Druhá chyba se týkala nastavení hodnoty `secret_key` v konfiguračním souboru aplikace. Tento klíč slouží k podepisování HTTP relace či generování CSRF *tokenů* [69]. V konfiguraci aplikace použité při lokálním testování byl klíč vždy nastaven na náhodně vygenerovanou hodnotu.

Po nasazení se ukázalo, že WSGI [76] modul použitý pro spárování Python aplikace se serverem Apache2 tuto aplikaci v některých případech restartuje. To by obecně ničemu nevadilo, nicméně při každém restartu byla znovu vygenerována nová hodnota `secret_key`, což vedlo k zneplatnění všech aktivních HTTP relací a CSRF *tokenů*. Uživatel rozhraní tak byl náhle odhlášen nebo nemohl odesílat formuláře.

V konfiguračním souboru aplikace, který je použit při jejím nasazení, bylo proto generování klíče nahrazeno pevnou hodnotou (dostatečně dlouhým náhodně generovaným klíčem, který však není reinicializován při restartu aplikace).

### 5.2.1 Provozní server

Aplikace byla nasazena na virtuálním serveru ze služby Amazon EC2, s jedním procesorovým jádrem a 1 GB paměti. Použit byl operační systém Ubuntu Xenial 16.04.

## 5.3 Simulátor podřízeného systému

V rámci testování aplikace nadřazeného systému jsem vytvořil simulátor podřízeného systému. Ten umožňuje snadné zasílání požadavku na API nadřazeného systému, podobně jako by to dělal reálný podřízený systém.

Simulátor je implementován jako jednoduchá stránka, která je součástí webového rozhraní nadřazeného systému. Zasílání požadavků je realizováno pomocí Javascriptu a knihovny jQuery [77].

Stránka simulátoru je dostupná na `https://adresa-systemu/simulator`, pokud byla aplikace nadřazeného systému spuštěna v ladícím módu (bylo nastaveno `DEBUG = True` v konfiguračním souboru aplikace).

## 5.4 Test odezvy nadřazeného systému

V závěru testování jsem vyzkoušel dobu odezvy nadřazeného systému při nasazení na Raspberry Pi 3. Chtěl jsem ověřit především dobu zpracování události zaslané podřízeným systémem.

Při testu se ukázalo, že doba odezvy roste s počtem událostí, které byly pro danou garáž zaznamenány. Zpočátku byly zaslané události zpracovány zhruba za 150 až 200 milisekund. Pokud však měla daná garáž v databázi uloženo již kolem 2000 událostí, vzrostla doba zpracování až na 800 milisekund.

Příčinou byl výchozí způsob, kterým framework SQLAlchemy mapuje databázové záznamy na Python objekty. Problém byl vyřešen dynamickým načítáním záznamů, které je popsáno v sekci 4.2.1. Po této úpravě se doba odezvy ustálila na přibližně 200 milisekundách, bez ohledu na počet zaznamenaných událostí.



---

# Závěr

V rámci práce byl vytvořen nadřazený systém pro zabezpečení garáží. Systém má za úkol sbírat zaznamenaná data od podřízených systémů monitorujících jednotlivé garáže a v případě nebezpečí odeslat zprávu s upozorněním. Uživatel (majitel či správce garážového komplexu) může systém spravovat pomocí webového rozhraní.

Aplikace nadřazeného systému byla vytvořena v jazyce Python a nasazena na jednodeskovém počítači Raspberry Pi 3. Výsledkem práce je tedy jednoúčelové zařízení, které je po krátké konfiguraci možné hned použít.

Text práce lze rozdělit do pěti kapitol: teoretický úvod, analýza, návrh, implementace a testování. Teoretický úvod slouží k definici použitého názvosloví. Kromě toho obsahuje stručný popis některých pojmů a konceptů, které se v práci vyskytují.

V analýze byla na základě zadání práce vytvořena specifikace nadřazeného systému, popisující jeho základní chování a požadavky, které má splňovat. Podle specifikace byl pak zvolen software a hardware vhodný k implementaci.

Významný prostor byl věnován volbě protokolu, na kterém bude postavena komunikace mezi nadřazeným a podřízenými systémy. Analýza se blíže zaměřila na protokoly HTTPS a MQTT. Kromě základních funkcí obou protokolů byly zkoumány také možnosti zabezpečení a autentizace účastníků komunikace. Pro komunikaci s podřízenými systémy byl nakonec zvolen protokol HTTPS.

Kromě protokolu byl v analýze také zvolen způsob ukládání dat (databázový systém SQLite3) a odesílání upozornění. Právě zasílání upozornění se ukázalo jako poměrně zajímavý problém s několika možnými řešeními. V rámci analýzy byly otestovány internetové služby pro zasílání e-mailů (Gmail) a SMS (Twilio). Kromě toho bylo také otestováno zasílání SMS pomocí GSM modulu připojeného přes USB port. Tento způsob byl nakonec použit ve výsledném zařízení.

Pro implementaci aplikace nadřazeného systému byl vybrán jazyk Python a webový framework Flask. Hlavní důvod volby byla moje předchozí zkušenost

s těmito nástroji.

V závěru analytické části byla zvolena vhodná hardwarová platforma pro tvorbu výsledného zařízení. Zkoumány byly jednodeskové počítače Raspberry Pi 3 a Zybo Zynq-7000. Deska Zynq-7000 byla zajímavá především díky integraci FPGA obvodu, ukázalo se však, že ten by měl v práci pouze velmi omezené využití. Jako platforma pro tvorbu zařízení bylo tedy zvoleno Raspberry Pi 3, které poskytuje vyšší procesorový výkon za výrazně nižší cenu.

Na základě analýzy byl vytvořen návrh aplikace nadřazeného systému, kde byl jako základ použit vzor *model-view-controller*. Díky tomuto vzoru by měl být systém poměrně snadno rozšiřitelný. Například přidání možnosti komunikovat s podřízenými systémy pomocí protokolu MQTT lze realizovat pouze doplněním vhodného *controlleru*.

Dále byl v kapitole návrh popsán způsob vyhodnocování stavu monitorovaných garáží a odesílání upozornění.

Pro komunikaci s podřízenými systémy bylo navrženo API, pomocí kterého mohou nadřazenému systému zasílat události popisující stav garáže (například při detekci kouře). Podřízené systémy se při přístupu k API prokazují pomocí náhodně generovaných klíčů.

Zde bylo potřeba vyřešit problém distribuce klíčů podřízeným systémům. K tomu byl implementován takzvaný registrační mód, který lze dočasně povolit ve webovém rozhraní nadřazeného systému. Když je mód zapnutý, mohou se podřízené systémy registrovat pomocí k tomu určeného API požadavku, a přístupový klíč jim je zaslán v odpovědi.

Webové rozhraní nadřazeného systému bylo navrženo na základě předpokládaných případů užití, jako je například zobrazení monitorovaných garáží či událostí zaznamenaných v konkrétní garáži. Přístup do rozhraní je ověřen pomocí hesla, které je uchováváno v zabezpečené podobě pomocí *hashovacího* algoritmu Argon2.

Jak je zmíněno výše, aplikace nadřazeného systému byla implementována v jazyce Python s pomocí frameworku Flask. Kromě toho bylo použito několik dalších nástrojů, jako například program pro ovládání GSM modulu Gammu nebo framework SQLAlchemy, který výrazně usnadnil práci s databází systému.

K výsledné aplikaci byla vytvořena sada automatizovaných testů, které ověřují funkčnost jednotlivých částí (například přihlašování do webového rozhraní). Tyto testy slouží především k dalšímu rozšiřování aplikace, neboť s jejich pomocí lze rychle ověřit, zda přidaný kód neporušil nějakou již implementovanou funkcionalitu. Pro další testování byl také vytvořen jednoduchý simulátor podřízeného systému.

V rámci testování byla aplikace nasazena na virtuálním serveru služby Amazon EC2<sup>7</sup>. Účelem bylo vyzkoušet proces nasazování aplikace. Nasaze-

---

<sup>7</sup>Aplikace je do konce června 2018 dostupná na <https://demo-garaze.tk>.

---

nou aplikaci (včetně simulátoru) je také možné použít k demonstraci funkce nadřazeného systému.

Výsledná aplikace nadřazeného systému byla poté úspěšně nasazena i na Raspberry Pi 3, čímž vzniklo zařízení, které bylo cílem této diplomové práce.

### **Zdrojové kódy**

Při vývoji aplikace nadřazeného systému byl použit verzovací systém Git. Repozitář se zdrojovým kódem je volně k dispozici (pod licencí LGPL) na serveru Github – <https://github.com/ggljzr/mi-dip-impl>. Repozitář také obsahuje testy, předpřipravené konfigurační soubory a stručný návod k nasazení aplikace na Raspberry Pi.





---

## Literatura

- [1] Dalling, T.: Model View Controller Explained. 2009, [cit. 2018-04-07]. Dostupné z: <https://www.tomdalling.com/blog/software-design/model-view-controller-explained/>
- [2] Dostálek, L.; Kabelová, A.: *Velký průvodce protokoly TCP/IP a systémem DNS*. Computer Press, třetí vydání, 2002, ISBN 80-7226-675-6.
- [3] Dierks, T.; aj.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, srpen 2008. Dostupné z: <https://tools.ietf.org/html/rfc5246>
- [4] Dostálek, L.: *Velký průvodce protokoly TCP/IP: Bezpečnost*. Computer Press, první vydání, 2001, ISBN 80-7226-513-X.
- [5] Housley, R.; aj.: Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459, Internet Engineering Task Force, leden 1999. Dostupné z: <https://tools.ietf.org/html/rfc2459>
- [6] Krawczyk, P.; aj.: Attack Surface Analysis Cheat Sheet. 2018, [cit. 2018-04-06]. Dostupné z: [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))
- [7] Zalewski, M.: Browser Security Handbook, part 2. 2009, [cit. 2018-04-06]. Dostupné z: <https://code.google.com/archive/p/browsersec/wikis/Part2.wiki>
- [8] Bird, J.; aj.: Cross-Site Request Forgery (CSRF). 2017, [cit. 2018-04-02]. Dostupné z: [https://www.owasp.org/index.php/Attack\\_Surface\\_Analysis\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet)
- [9] Callegati, F.; aj.: Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security and Privacy*, ročník 7, únor 2009: s. 78 – 81. Dostupné z: <http://ieeexplore.ieee.org/document/4768661/>

- [10] Hornby, T.: Salted Password Hashing - Doing it Right. 2017, [cit. 2018-04-02]. Dostupné z: <https://crackstation.net/hashing-security.htm>
- [11] Apache Software Foundation: Apache – Frequently Asked Questions. 2015, [cit. 2018-03-04]. Dostupné z: <https://wiki.apache.org/httpd/FAQ>
- [12] Arduino: Web Client. 2015, [cit. 2017-10-25]. Dostupné z: <https://www.arduino.cc/en/Tutorial/WebClient>
- [13] Grokhotkov, I.: esp8266wifi – Client Example. 2017, [cit. 2017-10-25]. Dostupné z: <https://github.com/esp8266/Arduino/blob/master/doc/esp8266wifi/client-examples.rst>
- [14] Electronic Frontier Foundation: Let's Encrypt – Frequently Asked Questions. 2017, [cit. 2017-11-07]. Dostupné z: <https://letsencrypt.org/docs/faq/>
- [15] Electronic Frontier Foundation: Certbot – About. [cit. 2017-10-18]. Dostupné z: <https://certbot.eff.org/about/>
- [16] Wallen, J.: When are self-signed certificates acceptable for businesses? 2017, [cit. 2017-11-08]. Dostupné z: <https://www.techrepublic.com/article/when-are-self-signed-certificates-acceptable-for-businesses/>
- [17] Reitz, K.: Requests: HTTP for Humans. 2018, [cit. 2018-03-01]. Dostupné z: <http://docs.python-requests.org/en/master/>
- [18] The Python Software Foundation: The Python Tutorial. 2018, [cit. 2018-03-04]. Dostupné z: <https://docs.python.org/3/tutorial/index.html>
- [19] Arduino: EEPROM Library. 2018, [cit. 2017-03-01]. Dostupné z: <https://www.arduino.cc/en/Reference/EEPROM>
- [20] Lampkin, V.: What is MQTT and how does it work with WebSphere MQ? 2012, [cit. 2017-10-25]. Dostupné z: [https://www.ibm.com/developerworks/mydeveloperworks/blogs/aimsupport/entry/what\\_is\\_mqtt\\_and\\_how\\_does\\_it\\_work\\_with\\_websphere\\_mq?lang=en](https://www.ibm.com/developerworks/mydeveloperworks/blogs/aimsupport/entry/what_is_mqtt_and_how_does_it_work_with_websphere_mq?lang=en)
- [21] Jaffey, T.: MQTT and CoAP, IoT Protocols. 2014, [cit. 2017-11-12]. Dostupné z: [https://eclipse.org/community/eclipse\\_newsletter/2014/february/article2.php](https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php)
- [22] Newsom, C.: Mosquitto Message Broker. 2016, [cit. 2017-11-16]. Dostupné z: <https://github.com/mqtt/mqtt.github.io/wiki/Mosquitto-Message-Broker>

- 
- [23] Light, R.: mosquitto.tls – Mosquitto Manual. [cit. 2017-11-20]. Dostupné z: <https://mosquitto.org/man/mosquitto-tls-7.html>
  - [24] Nguyen, J.: OpenSSL Certificate Authority. 2015, [cit. 2017-11-20]. Dostupné z: <https://jamielinux.com/docs/openssl-certificate-authority/>
  - [25] Light, R.: mosquitto.conf – Mosquitto Manual. [cit. 2017-11-20]. Dostupné z: <https://mosquitto.org/man/mosquitto-conf-5.html>
  - [26] Light, R.: libmosquitto – Mosquitto Manual. [cit. 2017-11-21]. Dostupné z: <https://mosquitto.org/man/libmosquitto-3.html>
  - [27] Eclipse Foundation: Python – Mosquitto Documentation. [cit. 2017-11-21]. Dostupné z: <https://mosquitto.org/documentation/python/>
  - [28] Eclipse Foundation: Embedded MQTT C/C++ Client Libraries. [cit. 2017-11-21]. Dostupné z: <http://www.eclipse.org/paho/clients/c/embedded/>
  - [29] Eclipse Foundation: Paho C++ Documentation. [cit. 2017-11-21]. Dostupné z: <http://www.eclipse.org/paho/files/cppdoc/index.html>
  - [30] Eclipse Foundation: Paho Python Documentation. [cit. 2017-11-21]. Dostupné z: <https://pypi.python.org/pypi/paho-mqtt>
  - [31] Hipp, D. R.: About SQLite. [cit. 2017-12-12]. Dostupné z: <https://www.sqlite.org/about.html>
  - [32] Hipp, D. R.: Datatypes In SQLite Version 3. [cit. 2018-03-03]. Dostupné z: <https://www.sqlite.org/datatype3.html>
  - [33] Anicas, M.: Why You May Not Want To Run Your Own Mail Server. 2014, [cit. 2018-02-26]. Dostupné z: <https://www.digitalocean.com/community/tutorials/why-you-may-not-want-to-run-your-own-mail-server>
  - [34] Google : Send email from a printer, scanner, or app. [cit. 2018-04-02]. Dostupné z: <https://support.google.com/a/answer/176600?hl=en>
  - [35] Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749, Internet Engineering Task Force, říjen 2012. Dostupné z: <https://tools.ietf.org/html/rfc6749>
  - [36] Google : Using OAuth 2.0 to Access Google APIs. [cit. 2018-04-02]. Dostupné z: <https://developers.google.com/identity/protocols/OAuth2>

- [37] ETSI : *GSM Technical Specification – GSM 07.05*. ETSI, 1996, ISBN 2-7437-0825-5, 1-74 s. Dostupné z: [http://www.etsi.org/deliver/etsi\\_gts/07/0705/05.00.00\\_60/gsmst\\_0705v050000p.pdf](http://www.etsi.org/deliver/etsi_gts/07/0705/05.00.00_60/gsmst_0705v050000p.pdf)
- [38] Čihař, M.: Gammu. 2018, [cit. 2018-04-02]. Dostupné z: <https://wammu.eu/gammu/>
- [39] Twilio : Twilio – Programmable SMS. 2018, [cit. 2018-04-02]. Dostupné z: <https://www.twilio.com/sms>
- [40] Twilio : The Twilio Python Helper Library. 2018, [cit. 2018-04-02]. Dostupné z: <https://www.twilio.com/docs/libraries/python>
- [41] Twilio : Auth Tokens and How to Change Them. 2018, [cit. 2018-04-02]. Dostupné z: <https://support.twilio.com/hc/en-us/articles/223136027-Auth-Tokens-and-how-to-change-them>
- [42] Twilio : Twilio – SMS pricing. 2018, [cit. 2018-04-02]. Dostupné z: <https://www.twilio.com/sms/pricing/cz>
- [43] Twilio : Twilio – Our products. 2018, [cit. 2018-04-02]. Dostupné z: <https://www.twilio.com/products>
- [44] Ronacher, A.: Flask. 2018, [cit. 2018-03-04]. Dostupné z: <http://flask.pocoo.org/>
- [45] Django Software Foundation : Why Django? 2018, [cit. 2018-03-04]. Dostupné z: <https://www.djangoproject.com/start/overview/>
- [46] Benchoff, B.: Introducing the Raspberry Pi 3. 2016, [cit. 2018-01-25]. Dostupné z: <https://hackaday.com/2016/02/28/introducing-the-raspberry-pi-3/>
- [47] Adafruit: Adding a Real Time Clock to Raspberry Pi. 2016, [cit. 2018-01-25]. Dostupné z: <https://learn.adafruit.com/adding-a-real-time-clock-to-raspberry-pi/overview>
- [48] Raspberry Pi Foundation : Raspbian FAQ. [cit. 2018-03-04]. Dostupné z: <https://www.raspbian.org/RaspbianFAQ>
- [49] Digilent: ZYBO FPGA Board Reference Manual. 2016, [cit. 2018-01-29]. Dostupné z: [https://reference.digilentinc.com/\\_media/zybo:zybo\\_rm.pdf](https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf)
- [50] Teich, J.: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, ročník 100, květen 2012: s. 1411 – 1430. Dostupné z: <http://ieeexplore.ieee.org/document/6172642/>

- 
- [51] Xilinx: Getting Started with Xilinx for Zynq-7000. [cit. 2018-01-29]. Dostupné z: [http://xillybus.com/downloads/doc/xillybus\\_getting\\_started\\_zynq.pdf](http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf)
  - [52] Digilent: Pmod RTCC Reference Manual. 2016, [cit. 2018-01-30]. Dostupné z: [https://reference.digilentinc.com/\\_media/reference/pmod/pmodrtcc/pmodrtcc\\_rm.pdf](https://reference.digilentinc.com/_media/reference/pmod/pmodrtcc/pmodrtcc_rm.pdf)
  - [53] DigitalOcean: DigitalOcean – Pricing. 2018, [cit. 2018-02-05]. Dostupné z: <https://www.digitalocean.com/pricing/>
  - [54] DigitalOcean: DigitalOcean – Droplets. 2018, [cit. 2018-02-05]. Dostupné z: <https://www.digitalocean.com/products/droplets/>
  - [55] Leach, P.; aj.: A Universally Unique Identifier (UUID) URN Namespace. RFC 4122, Internet Engineering Task Force, červenec 2005. Dostupné z: <https://tools.ietf.org/html/rfc4122>
  - [56] Dinu, D.; Khovratovich, D.; aj.: Argon2. 2015, [cit. 2018-04-6]. Dostupné z: <https://github.com/p-h-c/phc-winner-argon2>
  - [57] Ronacher, A.: Modular Applications with Blueprints. 2017, [cit. 2018-03-15]. Dostupné z: <http://flask.pocoo.org/docs/0.12/blueprints/>
  - [58] Tezer, O. S.: How To Structure Large Flask Applications. 2014, [cit. 2018-03-16]. Dostupné z: <https://www.digitalocean.com/community/tutorials/how-to-structure-large-flask-applications>
  - [59] Bayer, M.: SQLAlchemy. [cit. 2018-03-16]. Dostupné z: <https://www.sqlalchemy.org/>
  - [60] Bayer, M.: Collection Configuration and Techniques. [cit. 2018-04-26]. Dostupné z: <http://docs.sqlalchemy.org/en/latest/orm/collections.html>
  - [61] Grönholm, A.: Advanced Python Scheduler. 2014, [cit. 2018-03-17]. Dostupné z: <https://apscheduler.readthedocs.io/en/latest/index.html>
  - [62] Bayer, M.: SQLAlchemy 1.2 Documentation. [cit. 2018-04-12]. Dostupné z: <http://docs.sqlalchemy.org/en/latest/index.html>
  - [63] Smith, K. D.; aj.: Decorators for Functions and Methods. 2004, [cit. 2018-03-20]. Dostupné z: <https://www.python.org/dev/peps/pep-0318/>
  - [64] Čihař, M.: The Gammu Manual. 2018, [cit. 2018-04-12]. Dostupné z: <https://wammu.eu/docs/manual/>

- [65] Ronacher, A.: Jinja2 Documentation. 2008, [cit. 2018-03-17]. Dostupné z: <http://jinja.pocoo.org/docs/2.10/>
- [66] Ronacher, A.: Flask – Templates. 2017, [cit. 2018-03-17]. Dostupné z: <http://flask.pocoo.org/docs/0.12/templating/>
- [67] Jacob, D.; Yang, H.: Flask-WTF User guide. [cit. 2018-03-19]. Dostupné z: <https://flask-wtf.readthedocs.io/en/stable/>
- [68] Drysdale, D.: phonenumbers Python Library. 2017, [cit. 2018-04-10]. Dostupné z: <https://github.com/daviddrysdale/python-phonenumbers>
- [69] Ronacher, A.: Flask – API. 2017, [cit. 2018-03-20]. Dostupné z: <http://flask.pocoo.org/docs/0.12/api/>
- [70] Ronacher, A.: Flask – Configuration Handling. 2017, [cit. 2018-03-28]. Dostupné z: <http://flask.pocoo.org/docs/0.12/config/>
- [71] Krekel, H.; aj.: Pytest Documentation. 2015, [cit. 2018-03-27]. Dostupné z: <https://docs.pytest.org/en/latest/contents.html>
- [72] Pulec, S.: FreezeGun: Let your Python tests travel through time. 2018, [cit. 2018-03-27]. Dostupné z: <https://github.com/spulec/freezegun>
- [73] Ronacher, A.: Testing Flask Applications. 2017, [cit. 2018-03-28]. Dostupné z: <http://flask.pocoo.org/docs/0.12/testing/>
- [74] van Elst, R.: Strong SSL Security on Apache2. 2015, [cit. 2018-03-30]. Dostupné z: [https://raymii.org/s/tutorials/Strong\\_SSL\\_Security\\_On\\_Apache2.html](https://raymii.org/s/tutorials/Strong_SSL_Security_On_Apache2.html)
- [75] Apache Software Foundation: Overview of new features in Apache HTTP Server 2.4. 2018, [cit. 2018-03-30]. Dostupné z: [http://httpd.apache.org/docs/trunk/new\\_features\\_2\\_4.html](http://httpd.apache.org/docs/trunk/new_features_2_4.html)
- [76] Eby, P. J.: Python Web Server Gateway Interface. 2003, [cit. 2018-03-30]. Dostupné z: <https://www.python.org/dev/peps/pep-0333/>
- [77] jQuery Foundation : About jQuery. 2015, [cit. 2018-03-20]. Dostupné z: <https://learn.jquery.com/about-jquery/>
- [78] Dumpleton, G.: mod\_wsgi. 2018, [cit. 2018-04-17]. Dostupné z: <https://modwsgi.readthedocs.io/en/develop/>
- [79] Ronacher, A.: Flask – mod\_wsgi (Apache). 2018, [cit. 2018-04-17]. Dostupné z: [http://flask.pocoo.org/docs/0.12/deploying/mod\\_wsgi/](http://flask.pocoo.org/docs/0.12/deploying/mod_wsgi/)

- [80] Ellingwood, J.: How To Create a Self-Signed SSL Certificate for Apache in Ubuntu 16.04. 2016, [cit. 2018-04-17]. Dostupné z: <https://www.digitalocean.com/community/tutorials/how-to-create-a-self-signed-ssl-certificate-for-apache-in-ubuntu-16-04>





---

## Nasazení

V této kapitole je stručně popsáno nasazení aplikace nadřazeného systému na jednodeskovém počítači Raspberry Pi 3 s operačním systémem Raspbian. Blokové schéma nasazené aplikace je na obrázku A.1.

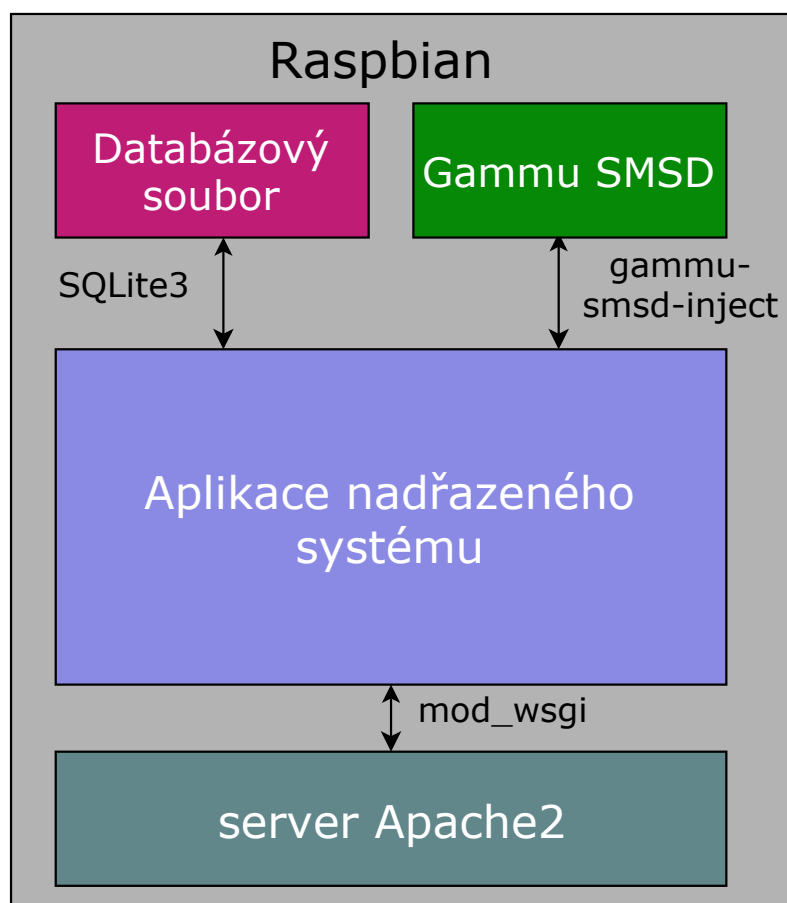
K instalaci a spuštění aplikace je zapotřebí tento software:

- Webový server Apache2.
  - Nutno nainstalovat modul `mod_wsgi` [78].
- Programovací jazyk Python3.
  - Systém pro správu balíčků `pip`. Díky němu lze snadno nainstalovat všechny potřebné Python moduly, jako například Flask nebo SQLAlchemy.
- Program pro ovládání GSM modulu Gammu. Nadřazený systém je možné spustit i bez instalace tohoto programu, pouze nebude fungovat zasílání SMS upozornění.

### A.1 Konfigurace aplikace nadřazeného systému

Před nasazením aplikace je vhodné zvolit konfigurační soubor, který má být použit pro její nastavení. To je možné provést nastavením proměnné prostředí `GARAGE_SYSTEM_CONFIG` na cestu ke zvolenému souboru.

Pokud tato proměnná není nastavena, je použit výchozí konfigurační soubor `default_config.py`.



Obrázek A.1: Softwarové blokové schéma nadřazeného systému. Aplikace běží na OS Raspbian. Jako webový server je použit Apache2, se kterým aplikace komunikuje pomocí modulu `mod_wsgi` [78]. Databáze je upravována pomocí systému SQLite3, pro odesílání SMS s upozorněními je použit program Gammu (bližší informace v sekci 4.2.3).

## A.2 Konfigurace Apache2

Po instalaci potřebného softwaru je nejprve nutné nakonfigurovat webový server Apache2 tak, aby doručoval aplikaci nadřazeného systému.

K doručení Python aplikace je použit modul `mod_wsgi`, jehož základní konfigurace je popsána v článku [79]. Spuštění aplikace pomocí `mod_wsgi` je provedeno pomocí skriptu z ukázky 14.

```
import sys
# předpokládané umístění kořenového adresáře aplikace
sys.path.insert(0, '/var/www/garage')
from garage_system import app as application
```

Ukázka 14: Skript pro spuštění aplikace na serveru Apache2 pomocí modulu `mod_wsgi`. Skript importuje objekt aplikace nadřazeného systému (`app`), který je pak tímto modulem používán [79].

### A.2.1 Přístup k databázi a souboru s uživatelským nastavením

Aplikace nadřazeného systému potřebuje měnit obsah databáze a souboru s uživatelským nastavením (`user_config.ini`). Je tedy nutné povolit uživateli, pod kterým aplikace běží, čtení a zápis těchto souborů.

Apache2 spouští aplikace jako uživatel `www-data`, přístup k souborům lze povolit způsobem popsáním v ukázce 15.

```
# nastavení vlastníka souboru na
# uživatele www-data ze skupiny www-data
$ sudo chown www-data:www-data /cesta/k/user_config.ini
$ sudo chown www-data:www-data /cesta/k/app.db
# povolení čtení a zápisu souborů vlastníkov
# vlastníkov a skupině
$ sudo chmod 660 /cesta/k/user_config.ini
$ sudo chmod 660 /cesta/k/app.db
```

Ukázka 15: Nastavení přístupových práv k databázi a souboru s uživatelským nastavením. Umístění souborů je možné nastavit v konfiguračním souboru aplikace (viz sekci A.1).

### A.2.2 Konfigurace HTTPS

Jelikož je při nasazení na Raspberry Pi použit *self-signed* certifikát, vycházel jsem při konfiguraci HTTPS z článku *How To Create a Self-Signed SSL Certificate for Apache in Ubuntu 16.04* [80].

Při konfiguraci HTTPS se ukázalo, že Apache2 nainstalovaný z repozitářů Raspbianu nemá ve výchozím nastavení povolený SSL modul<sup>8</sup> a je nutné ho ručně povolit pomocí příkazu `a2enmod ssl`. Pro srovnání, při instalaci Apache2 na testovacím serveru ze sekce 5.2 byl tento modul povolen implicitně.

<sup>8</sup>Situace v dubnu 2018.

### A.3 Konfigurace Gammu SMSD

Konfigurace démona Gammu SMSD je poměrně snadná, jak je vidět v ukázce 16. Nejprve je potřeba zvolit způsob komunikace s GSM modulem – v tomto případě jde o AT příkazy (pro bližší informace viz [37]). Také je možnost nastavit explicitně rychlost sériové linky (například 19200 baudů).

Dále je třeba zvolit umístění zařízení. To nemusí být vždy `ttyUSB0`, konkrétní umístění lze zjistit po připojení GSM modulu příkazem `dmesg | grep "GSM"`.

V konfiguračním souboru je také zvoleno umístění složek, do kterých budou ukládány zprávy určené k odeslání a již odeslané zprávy. Do složky zpráv k odeslání (`outbox`) bude aplikace nadřazeného systému vkládat odesílaná upozornění pomocí příkazu `gammu-smsd-inject`. Je proto nutné, aby aplikace (respektive uživatel, pod kterým běží, což je `www-data`) měla přístup do této složky.

Při použití výchozích složek (`/var/spool/gammu`), jako v ukázce 16, stačí pro povolení přístupu přidat uživatele `www-data` do skupiny `gammu`. Tato skupina je vytvořena při instalaci programu Gammu.

```
[gammu]
# komunikace pomocí AT příkazů
# linka nastavená na 19200 baudů
connection = at19200
port = /dev/ttyUSB0
device = /dev/ttyUSB0
# synchronizace času mezi počítačem a GSM modulem
synchronizetime = yes

[smsd]
# pro ukládání zpráv je použit souborový systém
service = files
logfile = syslog
# kontrola složek každých 5 sekund
loopsleep = 5

# složky pro ukládání SMS
inboxpath = /var/spool/gammu/inbox/
outboxpath = /var/spool/gammu/outbox/
sentsmspath = /var/spool/gammu/sent/
errorsmspath = /var/spool/gammu/error/
```

Ukázka 16: Konfigurační soubor Gammu SMSD.

## Uživatelská příručka



Obrázek B.1: Nadřazený systém (zařízení).

Tato uživatelská příručka se zabývá webovým rozhraním nadřazeného systému. Sekce příručky odpovídají jednotlivým stránkám rozhraní. V každé sekci jsou popsány možnosti, které má uživatel na dané stránce k dispozici.

Přihlášení do webového rozhraní je prováděno pomocí hesla. Přihlašovací formulář je zobrazen na obrázku B.2.

### B.1 První přihlášení

Pro první přihlášení je možné použít implicitní heslo „password“. Poté je uživatel přesměrován na formulář se změnou hesla a vyzván ke změně z implicitní hodnoty (viz obrázek B.3).

## Správa garáží



The screenshot shows a login form with a dark blue header containing the title 'Přihlášení'. Below the header, there is a label 'Heslo:' followed by a light gray rectangular input field. At the bottom of the form is a dark blue button with the text 'Přihlásit'.

Obrázek B.2: Přihlašovací formulář webového rozhraní.

## Změna hesla



The screenshot shows a password change form. At the top, there is an orange warning box with the text 'Je nutné změnit heslo z implicitně nastavené hodnoty'. Below this, there are three labels with corresponding light gray input fields: 'Staré heslo:', 'Nové heslo:', and 'Nové heslo znovu:'. At the bottom of the form is a dark blue button with the text 'Potvrdit'.

Obrázek B.3: Výzva ke změně hesla.

## B.2 Hlavní stránka

Po úspěšném přihlášení se zobrazí hlavní stránka webového rozhraní. Zde má uživatel k dispozici následující možnosti (viz čísla na obrázku B.4):

1. **Nová garáž** – vytvoření nové garáže pomocí uživatelského rozhraní. Nová garáž je přidána do seznamu garáží na této stránce.
2. **Registrační mód** – indikátor stavu registračního módu. Registrační mód umožňuje podřízeným systémům vytvářet nové garáže registračním požadavkem. Při registraci pomocí tohoto módu je podřízenému systému automaticky zaslán API klíč, a není nutné ho ručně nahrávat. Po zapnutí je registrační mód aktivní po dobu tří minut, poté se sám automaticky vypne.
3. **Nastavení** – uživatelské nastavení aplikace:

Menu		Garáže				
Obnovit						
Nová garáž	1					
Registrační mód:						
Vypnutí	2					
Prepnout reg. mód						
Nastavení	3					
Změna hesla						
Uživatel						
Odhlásit						

ID	Označení	Dveře	Poslední hlášení	Další plánované hlášení	Stav
[1]	Testovací garáž	Zavřeno	2018-04-16 13:38:43.844141	2018-04-16 14:38:43.844141	Nehlásí se
[2]	Durhá garáž	Otevřeno	2018-04-10 03:40:56.686703	2018-04-10 04:40:56.686703	Nehlásí se
[3]	Novák Jan	Otevřeno	Žádné	Žádné	OK
[4]	GARS_HUS_NOV	Zavřeno	2018-04-20 19:55:32.309514	2018-04-20 20:55:32.309514	Nehlásí se
[5]	Nová garáž	Otevřeno	2018-05-01 22:19:29.667151	2018-05-01 23:19:29.667151	Nehlásí se

Obrázek B.4: Hlavní stránka aplikace.

- **Změna hesla** – možnost změny přístupového hesla do webového rozhraní.
- **Uživatel** – viz sekci B.4.
- **Odhlásit** – odhlášení z webového rozhraní.

#### 4. Seznam sledovaných garáží (podřízených systémů):

- **ID** – identifikátor garáže v systému, zároveň slouží jako odkaz na stránku garáže (viz sekci B.3).
- **Označení** – uživatelem zvolené označení garáže.
- **Dveře** – stav dveří garáže (otevřeno/zavřeno).
- **Poslední hlášení** – datum a čas posledního zaznamenaného hlášení.
- **Další plánované hlášení** – datum a čas dalšího plánovaného hlášení.
- **Stav** – stav garáže (OK, Nehlásí se, Detekce pohybu, Detekce kouře).

#### 5. Záznam garáže.

## B.3 Stránka garáže

**Menu**

- Obnovit
- Zneplatnit API klíč
- Smazat garáž
- Zpět

**Novák Jan**

**Údaje o garáži**

- ID: 3
- Stav: OK
- Dveře: Zavřeno
- Poslední hlášení: 2018-05-02 00:34:44.094693
- Další plánované hlášení: 2018-05-02 01:34:44.094693
- API Klíč: 66d2ebc3651b446b871e37c0c04676

**Nastavení garáže**

Označení: Novák Jan

Perioda hlášení (minuty): 60

Telefonní číslo: +420123456789

Poznámka: Jan Novák, Božetěchova 756/45, Praha

**Potvrdit**

**Vše | Kontrolní hlášení | Otevření dveří | Zavření dveří | Pohyb | Kout | Chyba zařízení**

**Seznam událostí**

[2018-05-02 00:34:44.094760]	Kontrolní hlášení
[2018-05-02 00:34:42.404769]	Detekce kouřel
[2018-05-02 00:34:40.673520]	Kontrolní hlášení
[2018-05-02 00:34:38.664769]	Zavření dveří
[2018-05-02 00:34:36.088426]	Otevření dveří
[2018-05-02 00:34:32.558096]	Kontrolní hlášení

Obrázek B.5: Stránka garáže.

Na obrázku B.5 je zobrazena stránka konkrétní garáže. Zde jsou k dispozici tyto možnosti:

- Zneplatnit API klíč** – vygeneruje nový API klíč garáže. Po jeho vygenerování již nebude podřízený systém v této garáži moci zasílat nové události. Pro obnovení přístupu je nutné nahrát na systém nový klíč, případně vytvořit novou garáž pomocí registračního módu.
- Smazat garáž** – smaže zobrazenou garáž, včetně zaznamenaných událostí.
- Údaje o garáži**, včetně API klíče.
- Nastavení garáže:**
  - Označení** – uživatelem zvolené označení, které se zobrazí na hlavní stránce.
  - Perioda hlášení** – perioda kontrolních hlášení podřízeného systému (v minutách).
  - Telefonní číslo** – telefonní číslo nájemce garáže. V případě vyplnění budou upozornění týkající se garáže zasílána i nájemci. Telefonní číslo je třeba vyplnit včetně předvolby (+420 pro Českou republiku).
  - Poznámka** – volitelná poznámka ke garáži (max 256 znaků).



- 5. **Filtry událostí** – možnost filtrovat zobrazené události podle typu.
- 6. **Události** – tabulka zaznamenaných událostí, včetně data a času.

## B.4 Uživatelské nastavení

V uživatelském nastavení může uživatel měnit telefonní číslo, na které nadřazený systém zasílá upozornění o změně stavu garáže.



## Seznam použitých zkratek

**API** Application Programming Interface

**AXI** Advanced eXtensible Interface

**CPU** Central Processing Unit

**CSRF** Cross-Site Request Forgery

**DDR** Double Data Rate

**EEPROM** Electrically Erasable Programmable Read-Only Memory

**FPGA** Field-programmable gate array

**GSM** Global System for Mobile communications

**HDMI** High-Definition Multimedia Interface

**HTML** HyperText Markup Language

**HTTP** HyperText Transfer Protocol

**HTTPS** HTTP Secure

**I2C** Inter-Integrated Circuit

**IP** Intellectual Property

**JSON** JavaScript Object Notation

**LCD** Liquid-Crystal Display

**LPDDR** Low Power DDR

**LTS** Long Term Support

**MQTT** Message Queuing Telemetry Transport

**MVC** Model-View-Controller

**NTP** Network Time Protocol

**OS** Operační systém

**OSI** Open Systems Interconnection

**PSK** Pre-Shared Key

**QoS** Quality of Service

**RAM** Random Access Memory

**RTC** Real Time Clock

**SD** Secure Digital

**SIM** Subscriber Identity Module

**SMTP** Simple Mail Transfer Protocol

**SSD** Solid State Drive

**SSL** Secure Socket Layer

**SoC** System on Chip

**TCP/IP** Transmission Control Protocol / Internet Protocol

**TLS** Transport Layer Security

**URL** Uniform Resource Locator

**USB** Universal Serial Bus

**UUID** Universally Unique Identifier

**WSGI** Web Server Gateway Interface

## Obsah přiloženého CD

	readme.txt.....	stručný popis obsahu CD
	mi-dip.....	Git repozitář se zdrojovým kódem práce ve formátu $\text{\LaTeX}$
	mi-dip-impl...	Git repozitář se zdrojovým kódem implementace (včetně testů)
	DP_Cervenka_Ondrej_2018.pdf.....	text práce ve formátu PDF