Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

BAKALÁŘSKÁ PRÁCE

MONITOROVÁNÍ FIREMNÍ SÍTĚ

10. května 2020

Tomáš Květoň

Zadání práce

1. Seznamte se s principy a technologiemi pro monitorování síťového provozu.
2. Prozkoumejte, jaké funkcionality dané podnikové sítě je potřeba monitorovat.
3. Navrhněte vhodné způsoby monitorování vybraných funkcionalit a způsob prezentace získaných údajů.
4. Navrhněte programové vybavení s využitím poznatků z předchozích bodů zadání. Při návrhu aplikace bude kladen důraz na její snadnou rozšiřitelnost o další funkcionality.
5. Navržený systém realizujte, ověřte jeho funkcionalitu a navrhněte vhodná rozšíření.

Poděkování

Tímto bych chtěl poděkovat vedoucímu mé bakalářské práce Ing. Ladislavu Pešičkovi za cenné rady, nápady, připomínky a především za čas, který strávil během tvorby mé bakalářské práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

10. května 2020 Tomáš Květoň

Abstract

The goal of this bachelor thesis is to create a network monitoring tool, which will graphically represent the gathered data in a suitable way and will be easily expandable with further routines. The major task is to create an option to display data from elsewhere and monitor internal a computer network at the same time. The idea of design came from the real company environment where this new monitoring tool replace the older monitoring tool currently working there. Additionally, the new monitoring tool should provide a suitable option for other companies that are searching for a simple monitoring tool. One of the features of this software is recording device history and display statistics of them. Analysis of existing monitoring tools, shortcomings in the same market area and analysis of selected technologies are the major interest in the theoretical part of the thesis. These technologies are selected with an emphasis on the most suitable solution for the specific company environment for which the monitoring tool is designed, but also for general use in other companies. The practical part of the thesis proposes the overall system and then describes its implementation. In the end, the implemented system is tested and other planned extensions of the system are mentioned here.

Abstrakt

Cílem této práce je vytvoření systému pro kontrolu firemní sítě, který bude vhodným způsobem graficky reprezentovat získaná data a bude snadno rozšířitelný o další testy. Hlavním zadáním je vytvořit možnost pro zobrazování dat odkudkoliv a zároveň monitorovat interní síť. Systém byl navržen na základě konkrétního firemního prostředí, kde by měl nahradit již zastaralý systém a zároveň být vhodnou možností pro další firmy, které hledají jednoduché řešení monitorování sítě. Jednou z funkcí je logování dat a zajistit vhodné grafické prezentace. Teoretická část práce je primárně zaměřena na analýzu existujících aplikací, nedostatky tvořící na podobném trhu a analýzu následně vybraných technologií. Tyto technologie jsou vybírány s důrazem na nejvíce vhodné řešení pro konkrétní prostředí pro které je aplikace navrhována, ale zároveň i pro obecné použití v jiných prostředích. V praktické části je navržen celkový systém a následně popsána jeho implementace. V závěru je implementovaný systém otestován a jsou zde zmíněny i další plánovaná rozšíření systému.

Obsah

1. Úvod

Cílem této práce je vytvořit jednoduchý systém pro monitorování firemní sítě pro menší podniky. Realizovaný systém bude modulární, a bude umožňovat snadnou správu a rozšiřitelnost aplikace. Pro firmu je důležité udržet všechna svá místní zařízení pod dohledem, aby bylo vidět, jestli jsou dostupná, popřípadě mít schopnost spouštět vzdálené testy. Správce bude mít přístupnou aplikaci pro ovládání, kde bude moci spravovat svá zařízení reprezentovaná pomocí IP adres. Následně lze těmto zařízením přiřazovat jednotlivé testy z výběru a řádně je přizpůsobit svým potřebám. Velmi důležitou věcí je informování uživatele o případném výpadku zařízení či jiné chybě v jeho systému. Aplikace bude vybavena možností upozornění při výpadku některého z nich.

První část této práce je věnována teoretickému rozboru, kde jsou analyzovány některé vybrané aplikace podobného zaměření. Jsou zde zmíněny pozitiva, negativa a následně pak i jejich souhrn pro náš systém. V následující kapitole jsou poté analyzovány možnosti technologií, které by mohly být použity na vývoj. Na závěr je uveden rozbor použitých technologií pro konkrétní začlenění do implementace.

V druhé části jsou využity poznatky z teoretického rozboru a jsou zde navrženy všechny vhodné funkcionality, které by měl výsledný systém obsahovat. Následně je v této části navržena celková struktura systému, ukládání dat, komunikace a přenos informací nebo zabezpečení dat systému.

V poslední části práce je popsána implementace navrženého systému, je zde uvedena struktura výsledného projektu a funkce jednotlivých částí. Poté je zde popsán postup testování systému a v závěru je navržen budoucí rozvoj systému.

2. Analýza potřebných vlastností

V této kapitole budou analyzovány nutné vlastnosti pro náš systém, které potřebujeme, abychom splnili požadavky naší cílové firmy. Zároveň se tyto požadavky pokusíme splnit, aby byl systém použitelný i pro jiné společnosti.

Cílová firma vlastní několik typů zařízení, která jsou potřeba testovat. U některých je informace o jejich stavu kritická a u dalších zase méně. My si pokusíme rozdělit zařízení podle této úrovně, abychom věděli, v jakém pořadí máme do našeho systému postupně dodávat nové testovací metody.

2.1 Nezbytné testovací metody

Důležitou informací v celém systému je dostupnost. Zda máme zařízení v počítačové síti dostupné nebo nelze zjistit jednoduchým testem PING[68]. Takový test by měl být základem každého testování a měl by být nastavitelný dle požadavků uživatele monitorovací aplikace (možnost opakovaného testování nebo vypršení času na testování). Z toho důvodu by jedna z prvních testovacích metod měla být právě test PING, který lze snadno aplikovat na téměř každé zařízení.

V naší cílové firmě se nalézá velké množství UPS[81] zařízení. UPS je záložním zdrojem, pokud dojde k poruše připojení elektrického proudu. Pokud by došlo k takové poruše mimo pracovní dobu, tak by to mohlo znamenat možná rizika. Zároveň, pokud by odstávka elektrického proudu trvala déle než jsou UPSky schopné pokrýt, tak v ideálním případě chceme bezpečně zařízení vypnout. Z tohoto důvodu potřebujeme znát stav UPS zařízení, abychom dle těchto informací mohli správně naložit.

V našem případě test UPS obsahuje ještě další požadavek a tím je získávání dat z tohoto zařízení. V reálném prostředí se můžeme setkat s více typy zařízení UPS a každé má jiný software, který nabízí jiné možnosti pro získání takových dat. V cílové firmě se nachází celkem 3 typy zařízení UPS a všechny poskytují možnost připojení pomocí protokolu SNMP[73].

SNMP tvoří standard pro získání dat z takovýchto zařízení.

Implementací těchto dvou kritických testovacích metod získáme pevný základ pro naší aplikaci.Naší snahou bude vytvořit programové prostředí takové, že bude velmi snadno rozšiřitelné (modulární). V těchto dvou testovacích metodách nalezneme již potřebné vlastnosti pro tvorbu nových metod.

2.2 OSTATNÍ TESTOVACÍ METODY

Mezi další testovací metody, které potřebujeme implementovat (jedná se o nadstavbu systému a v aktuální verzi systému se nevyskytují), se řadí například:

- Testování teploty pomocí teploměrů umístěných v serverové místnosti nebo jiných důležitých místech firmy
- Testování kapacity diskového prostoru na cílovém zařízení
- Testování logovacích souborů (hledání shody textového řetězce)
- Spouštění externích testů (příkazové soubory (Windows))
- Rozesílání pravidelných SMS zpráv o stavu systému

Nachází se zde mnoho dalších testů, které mohou být použity v mnoha případech, a pokud bychom měli již implementaci testů z předchozí kapitoly 2.1, tak implementace těchto testů nebude složitá z pohledu programátora. Vše se odvíjí od samotné implementace, kterou chceme systém připravit pro taková možná rozšíření a následně do systému snadno přidat další testy dle potřeby.

2.3 MOŽNÁ ROZŠÍŘENÍ

Jedna z možných rozšíření, která by mohla být přínosem, je podpora I/O modulů Quido[38], které dokáží zaznamenávat vstupní informace přes hardwarové rozhraní COM[23] a získávat informace pro mnoho zařízení různých typů a umožní testování těchto zařízení. Quido jsou zařízení, která mají vstupy a výstupy, a která lze sledovat a ovládat přes počítačovou síť (Ethernet[31]), USB[82], RS232/RS485[71].

2.4 Souhrn

Hlavním cílem práce bude vypracování takového systému, který bude snadno rozšířitelný o nové testovací metody. Zároveň musíme připravit minimalistický model tohoto systému (MVP[54]). Chtěli bychom implementovat všechny kritické testy, aby bylo možné systém otestovat v reálném prostředí, a pokud úspěšně vytvoříme toto fungující prostředí, tak budeme dále vyvíjet nové testy, které zvýší použitelnost systému.

3. Analýza existujících aplikací

V této kapitole budou analyzovány tři vybrané nástroje pro monitorování počítačové sítě, které jsou často využívány. Zároveň tato řešení již byla konzultována a zvažována k nasazení pro naše konkrétní cílové prostředí. Naše požadavky na tento systém obsahují limity, které vedou k vytvoření vlastního řešení a zároveň můžeme rozšířit tento systém pro veřejnost.

Myšlenka takové aplikace či systémového nástroje je získat dohled nad funkcionalitou interní počítačové sítě ve firmě. Nástroje analyzované v této kapitole jsou: Nagios, SolarWinds Server and Application Monitor (SAM) a Zabbix. Všechny tyto aplikace jsou veřejně používané. Celkové shrnutí analýzy je uvedeno na závěr (3.4).

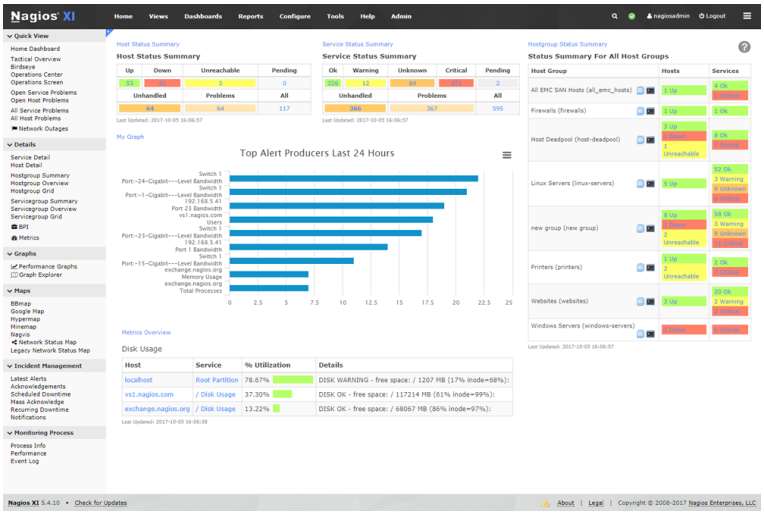
3.1 NAGIOS

Nagios je zaměřen na široké množství uživatelů. Je vhodný jak pro velké firmy, ale také pro malé a střední firmy. Nagios je placený systém s velice flexibilním předplatitelským plánem. Také nabízí verzi zdarma [61] s omezeními (viz. [56]).

Zaměřuje se na monitorování počítačových sítí. Je jednoduchý k instalaci, avšak jeho nastavení může být komplexní. Nastavení se provádí pomocí textových/konfiguračních souborů. [59]

Nagios nabízí velmi širokou škálu možného testování jako jeho konkurenti. Z našeho úhlu pohledu se zaměřujeme na jednoduché testy PING, testy UPS a testy logovacích souborů. Nagios nabízí všechny tyto vlastnosti, přesně tak, jak by jsme je chtěli i my implementovat [58] [57]. Zmíněná konkurence nabízí velmi podobné řešení, ale nesmíme zapomenou, že hledáme řešení plně zdarma (limitace protokolem SNMP, který je podporován pouze placenou verzí), tedy možný open-source a zároveň s kompatibilitou pro operační systém Windows.

Jedná se o jedno z nejoblíbenějších řešení [60] a umí velké množství funkcí. Náhled uživatelského rozhraní můžeme vidět na obrázku níže (Obr. 3.1).



Obrázek 3.1: Náhled rozhraní Nagios [62]

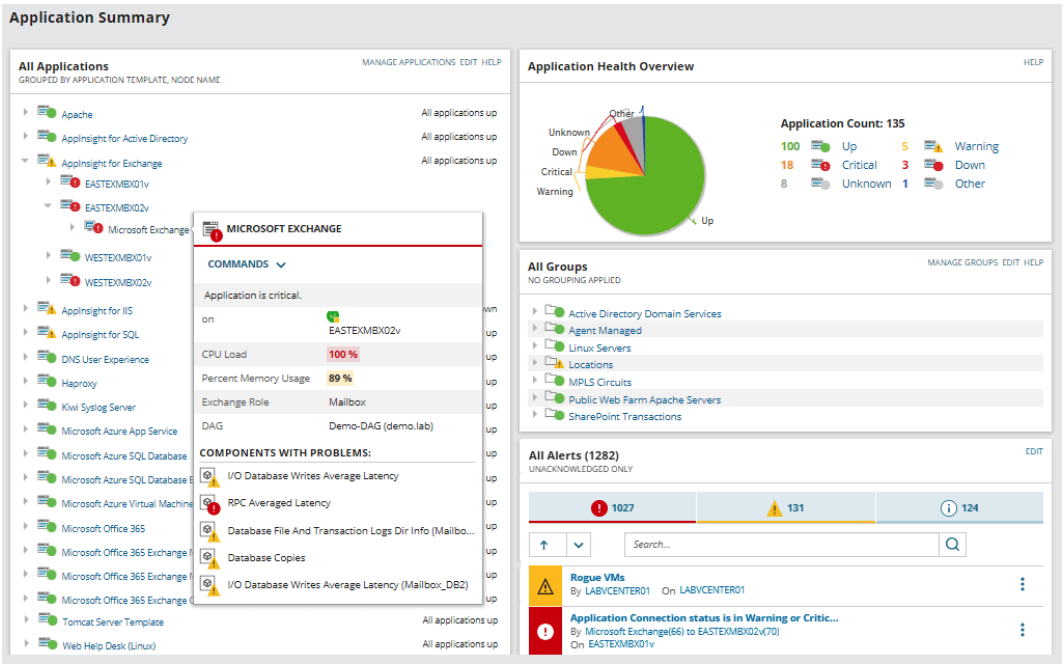
3.1.1 Souhrn

- Placená verze (cena začíná na $3.495 - 2020)
- Verze zdarma s limitací některých funkčností (viz. [56]).
- Podpora operačního systému Windows a Linux

3.2 SolarWinds Server and Application Monitor (SAM)

SAM umožňuje monitorovat tisíce zařízení. Je možné monitorovat systémy, služby, hardware a mnoho dalších věcí. Nabízí přehledné grafické rozhraní k vizualizaci stavu, ale zároveň nabízí spoustu komplexních systémů, které z aplikace mohou tvořit zbytečně komplikované prostředí pro jednoduché nasazení. [74]

SolarWinds nabízí velmi slibné řešení, které splňuje mnoho našich požadavků. Například monitorování log souborů [75], jednoduchého pingování nebo UPS testy pomocí SNMP [76]. Aplikace však nenabízí možnost verze zdarma. Nabízí pouze placenou verzi začínající na ceně $2.995 (2020) s možností 30-ti denní zkušební doby. Podpora je pouze pro Windows server [77]. Samotný náhled uživatelského rozhraní aplikace SAM lze vidět na obrázku níže (Obr. 3.2).



Obrázek 3.2: Náhled rozhraní SAM [77]

3.2.1 Souhrn

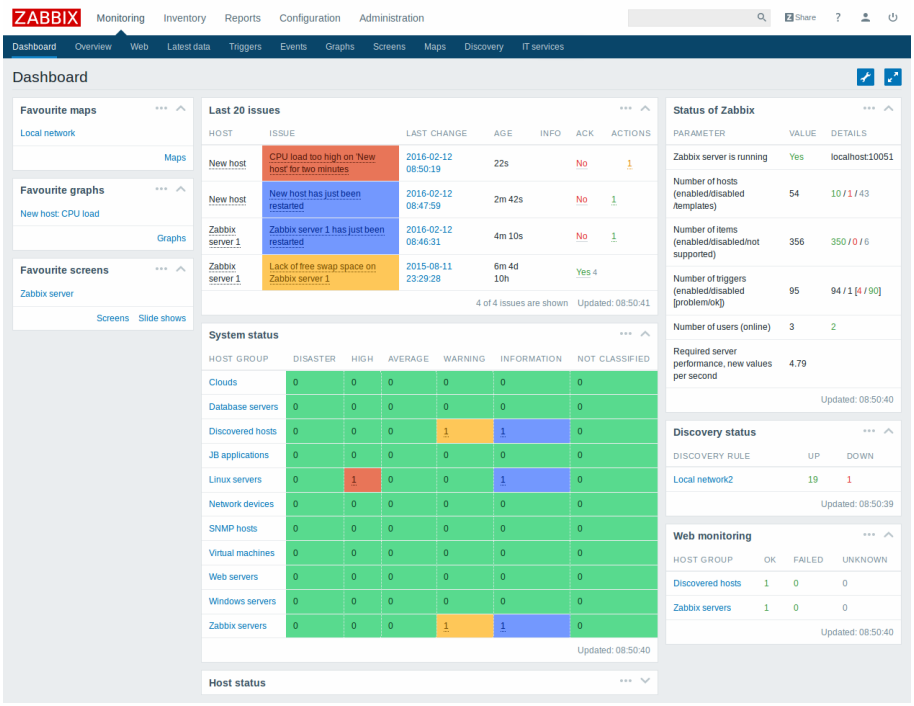
- Placená verze (cena začíná na $2.995 - 2020)

- Podpora operačního systému Windows

3.3 Zabbix

Zabbix je open-source monitorovací nástroj. Je populární pro svoji jednoduchost a příjemné webové rozhraní, které je plně konfigurovatelné. Zaměřuje se na monitorování hardwaru počítačových sítí a serverů. Jednou se zajímavých vlastností je predikce stavu podle zpětně zaznamenané historie [95].

Díky tomu, že je Zabbix open-source, tak má aktivní uživatelskou komunitu a velmi dobrou dokumentaci. Je velmi efektivní pro méně jak 1000 zařízení. Poté jeho výkon a rychlost klesá (podrobněji vysvětleno [94]). Jednou z nevýhod je chybějící nabídka real-time testů a reportů (viz. [94]). Zabbix je podporován pouze UNIX systémy.



Obrázek 3.3: Náhled rozhraní Zabbix [96]

3.3.1 Souhrn

- Verze zdarma (open-source)
- Podpora operačního systému Windows pouze pomocí agentů [93] (hlavní server musí být na jiném operačním systému).

3.4 Souhrn

Všechny zkoumané aplikace umožňují monitorování sítě. Dokáží zobrazit statistiky a informovat správce o změnách stavů. Zároveň všechny tyto aplikace ukazují svoji profesionalitu svými funkcemi, ale zároveň i možnostmi nastavení.

Nicméně, po analýze se můžeme všimnout, že na závěr nemůžeme najít takové řešení, které by podporovalo platformu Windows, mělo možnost verze zdarma bez zásadních omezení a zároveň se jednalo o jednoduše nastavitelnou aplikaci. Samozřejmě zde nejsou vyjmenována všechna řešení, ale především ta nejpoužívanější a kladně hodnocená.

Náš požadavek je najít takový nástroj, který by byl schopný podporovat platformu Windows, měl velice jednoduché ovládání s možnostmi vlastní konfigurace a možnost zasílat SMS na předem známé telefonní číslo při výpadku.

Další věcí je, že všechny tyto nástroje se zabývají velmi detailními testy. V našem případě bychom rádi měli monitorování sítě abstraktněji, například testy ping nebo vyhodnocení dat vytažených přes protokol SNMP. Zmíněné nástroje podporují tyto testy jen z části, které my chceme mít trochu více přizpůsobené testovacímu prostředí. Z tohoto důvodu je naším záměrem aplikace, která bude podporovaná systémem Windows, ale zároveň udržet si možnost vydat tuto aplikaci multiplatformně. Dalším záměrem je vytvořit jednoduše použitelné prostředí pro monitorování počítačové sítě, kde budou předdefinované testy, které mají možnost vlastní konfigurace podle potřeb uživatele. V neposlední řadě je potřeba mít také možnost odesílání zpráv správci při možném výskytu chyby a také přehledné produkční zobrazení.

4. Analýza technologií pro zpracování

V této kapitole budou analyzovány technologie, které by se mohly použít pro zpracování našeho řešení. Jedná se hlavně o technologie, se kterými jsem se již dostal to styku a mám s nimi dostatek zkušeností, Tyto technologie se zároveň hojně prakticky používají ve velkých společnostech ke zpracování jejich řešení.

Každá se zmíněných technologií je stručně obecně popsána pro obeznámení se základními vlastnostmi dané technologie. Hlavní výběr a podrobnější analýza je v souhrnu těchto technologií (4.6).

4.1 PHP

PHP[67] je velmi populární skriptovací jazyk pro univerzální účely. Běžně se kód procesuje na straně webového serveru pomocí PHP interpretora implementovaného například pomocí CGI (Common Gateway Interface). PHP je primárně zamýšleno pro UNIX systémy, nicméně dokáže běžet i na jiných operačních systémech s mírně omezenou funkcionalitou.

V prostředí PHP se nachází i několik velmi dobrých frameworků, které by mohly naší aplikaci nabídnout požadovanou kvalitní základnu. Mezi těmi nejlepšími se nachází například Laravel[48] a Symfony[80].

Laravel[48] je open-source PHP web framework se záměrem vytvářet webové aplikace na architektonickém modelu MVC (Model-View-Controller)[53]. Základem Laravelu je Symfony, ze kterého je vytaženo to nejlepší a je obestavěno přídavnými funkcemi a tím tvoří dohromady právě framework zvaný Laravel.

4.2 JavaScript

JavaScript[43] je vysoko-úrovňový, často just-in-time kompilovaný jazyk. Má syntaxi složených závorek, dynamické typování proměnných a objektově orientované možnosti řešení.

Vedle HTML[35] a CSS[25], JavaScript je jednou z nejdůležitějších technologií ve světě webu. JavaScript nabízí interaktivní webová řešení a je nedílnou součástí webových aplikací. Jednou z největších výhod JavaScriptu je funknost na straně klienta, a proto všechny hlavní internetové prohlížeče mají JavaScript engine již vestavěný pro jeho spouštění.

JavaScript engine byl používán pouze jen v internetových prohlížečích, nicméně v dnešní době se používají i v jiných serverových webových řešeních, běžně pomocí technologie Node.js[63]. Také se používá v řadách aplikací vytvořené s pomocí frameworku jako je Electron[28], React.js[70] nebo Ionic[40].

Electron[28] je řešení, o kterém je vhodné uvažovat. Jedná se o multiplatformní řešení, které má již na svém portfoliu velmi známé aplikace, které ho využívají. Jedná se o open-source framework spravovaný společností GitHub. Nabízí možnost tvorby desktopových aplikací s uživatelským rozhraním za pomoci HTML a CSS. Kombinuje Chromium[37] pro renderování a Node.js jako runtime. Electron je jedním z hlavních GUI frameworků[29].

4.3 JAVA

Java je objektově orientovaný programovací jazyk pro obecné použití. Java je především známá svojí možností běžet na velkém množství operačních systémů operačním systému bez omezení. Java je primárně používána pro desktop aplikace a zároveň zároveň může tvořit back-end kód na serveru.

Jedním z možných rozšíření pro Javu je JavaFX, která dodá snadné rozhraní pro tvorbu grafického uživatelského rozhraní. Hlavním záměrem jeho vzniku bylo nahradit Swing (knihovna pro tvorbu GUI) něčím novějším.

4.4 C++ / C

Rodinu programovacích jazyků C představuje C a jeho objektově orientovaná verze C++. C je nízkoúrovňový programovací jazyk. Obě dvě možnosti v základu nenabízí možnost grafického uživatelského rozhraní, ale jedná se o vysoce výkonné programovací jazyky. Oba programovací jazyky jsou nativně podporovány UNIX systémy, ale je možné je provozovat i na jiných operačních systémech s omezenou funkcionalitou.

4.5 C#

Další z rodiny C, který je vyvíjen společností Microsoft. Je objektově orientovaný. Byl vyvinut jako součást v .NET Frameworku.

.NET Framework běží primárně na operační systému Windows. Zahrnuje rozsáhlou knihovnu pojmenovanou jako Framework Class Library (FCL). Programy napsané pro rozhraní .NET Framework se spouštějí v softwarovém prostředí (na rozdíl od hardwarového prostředí) s názvem Common Language Runtime (CLR). CLR je aplikační virtuální stroj, který poskytuje služby, jako je zabezpečení, správa paměti a zpracování výjimek. Programový kód napsaný pomocí rozhraní .NET Framework se proto nazývá „managed code“. FCL a CLR společně tvoří .NET Framework[1].

.NET Framework se dále rozděluje na několik částí, které mají svůj vlastní význam. Primárně podporuje operační systém Windows, ale určité části .NET Frameworku dokáží běžet více platformách.

4.6 Souhrn

Prakticky je možné naše řešení vytvořit v jakékoli ze zmíněných technologií, ale některá řešení jsou vhodná více, některá méně. Naše podmínka je samozřejmě multi-platforma, pokud to bude možné. Minimálně je potřeba vytvořit řešení pro platformu Windows. S velkou pravděpodobností bude potřeba pracovat ve velkém množství s vlákny, které v jazycích C a C++ mohou být problém při multiplatformním řešení. Další věcí, kterou chceme uživatelům nabídnout je přehledné a jednoduché uživatelské prostředí. To jsme schopni ve všech případech splnit, ale některé technologie, jako například Electron, pracují s HTML a CSS, v tom případě bychom měli daleko lepší a větší možnosti než například rozhraní JavaFX[41]. Zároveň chceme jednoduchou správu aplikace a modulárnost aplikace.

Pokud se na zmíněné technologie koukneme z pohledu tvorby grafického uživatelského rozhraní, tak rozhodně nejlépe na tom budeme u webové aplikace postavené na Electron nebo PHP frameworku Laravel, který taktéž využívá HTML a CSS. Na druhou stranu nejhůře přizpůsobitelnou tvorbou grafického uživatelského prostředí z našeho výběru, podle osobní preference, je v C, C++. Tyto technologie nejsou nejmladší, ale na druhou stranu, velmi účinné po výkonnostní stránce a jejich použití bych osobně viděl v jiných aspektech použití.

Java si zase drží svá vlastní pravidla, zejména ohledně CSS. Z toho důvodu vše začíná být nepřehledné a odbočující ze standardů[42].

Osobní preference i preference cílové firmy, kde bude systém přednostně nasazen, je nepoužít Javu. Důvodem je velká konzumace paměti, která je u Javy velmi dobře známá. Druhým důvodem, který vychází z analýzy, je horší správa grafického rozhraní, které bychom chtěli udržet na co nejpřehlednější úrovni. Na základě toho bych, dle vlastních preferencí, vyřadil programovací jazyk C a C++. Dále z vlastní zkušenosti s React.js bych se velmi rád vyvaroval řešení přes JavaScript, které je v poslední době velmi moderní a velmi používané. Hlavním problémem je, že JavaScript nabízí velmi povolná pravidla jak udržet programátora dodržovat řádnou strukturu projektu. Struktura je zakládána na procedurálním programování s objektovými prvky. I přes moderní trend bych se rád tomuto řešení vyvaroval a držel se standardů, které fungují a při velkém projektu není pak potřeba nadstandardních metod pro udržení projektu v chodu.

Po zvážení všech těchto informací máme zbylé dvě technologie - PHP a C#. Výsledný verdikt je i přes to velmi jednoduchý, protože PHP je primárně podporovaný UNIX systémy a použití na jiných operačních systémech by nemuselo být vždy nejlepší, zejména při použití vícevláknového zpracování. Na druhé straně C# (.NET) může být multiplatformní a nabízí velkou škálu součástí pro různé použití. Závěrem tedy vybereme .NET jako naši vývojovou technologii. Nabízí velmi bohatou možnost tvorby grafického rozhraní na desktopu (osobně bych řekl, že tu nejlepší možnost bez použití webview - standardní HTML a CSS)[85]. Dokážeme vytvořit multiplatformní řešení, ale také zároveň plně podporuje platformu Windows a v neposlední řadě je možné vytvořit webové řešení[12].

5. Popis použitých technologií

V této kapitole budou analyzovány použité technologie pro vytvoření našeho zadání. Jsou zde uvedeny veškeré technologie, které jsou zapotřebí k běhu našeho systému. Všechny tyto technologie jsou velmi obecně popsány tak, aby byl zřejmé k čemu se a jak se dají využít. Podrobné začlenění do projektu je popsáno v kapitole návrhu (6).

5.1 .NET - Struktury (frameworks)

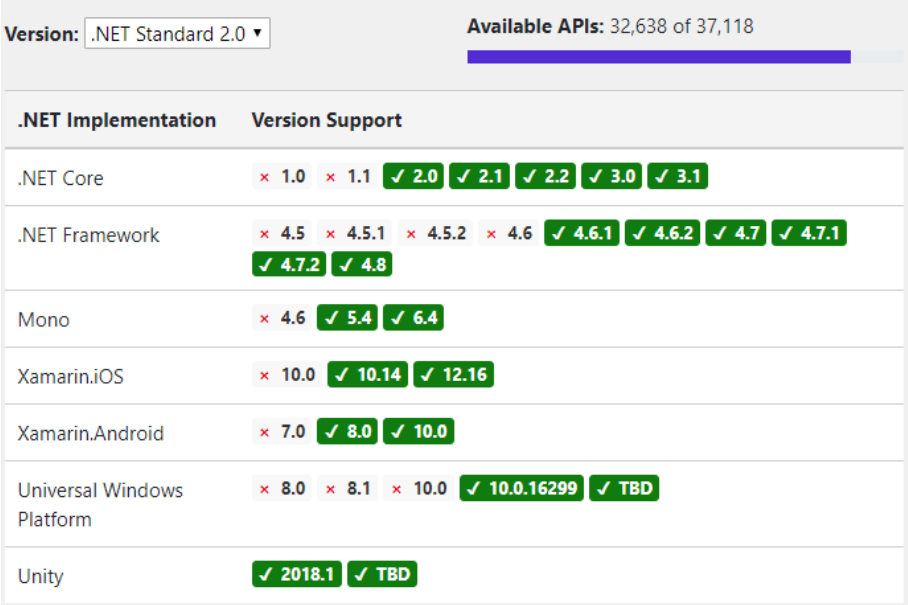
.NET se rozděluje na několik kategorií, které mezi sebou mohou navzájem spolupracovat. Avšak každá z těchto kategorií se hodí pro trochu jiný vývoj. V následující kapitole různé .NET kategorie projdeme a shrneme všechny jejich základní rozdíly a následně budeme schopni v návrhu aplikace (6) správně vybrat z těchto kategorií pro náš projekt [8].

5.1.1 .NET Standard[6]

Existuje mnoho implementací .NET. Každá implementace dovoluje .NET kódu být spuštěna na jiných platformách - Linux, maOS, Windows, iOS, Android a spoustu dalších. .NET Standard je formální specifikací .NET API, kde je záměrem být dostupný na co nejvíce platformách jako například: .NET Core, .NET Framework, Mono nebo také Unity.

.NET Standard dovoluje vytvářet knihovny proti odsouhlasené množině API s ujištěním, že mohou být použité v jakékoli implementaci .NET aplikací - mobilní, dekstop, IoT, webové, nebo kdekoli, kde píšete .NET kód.

Každá nová verze .NET Standardu přidává stále více API k použití. Používáním vyšších verzí .NET knihoven nabízí širší využití .NET Standardu. Jako příklad (Obrázek 5.1) je ukázka podpory předposlední verze .NET Standardu pro ostatní knihovny .NETu.



Obrázek 5.1: Podpora knihovny .NET Standard[7]

5.1.2 .NET Core

.NET Core je cross-platform verze .NETu pro vytváření webových stránek, služeb a konzolových aplikací. Je velmi blízký .NET Standardu, ze kterého implementace vychází. Nabízí více možných implementací než samotné operační systémy. Podrobnější specifikace nabízí konkrétní rozdělení na jednotlivých platformách jako například: UWP nebo například ASP.NET Core.

Během jeho užívání se .NET Core stal velmi populárním a standardem mnoha aplikací. Z toho důvodu se do budoucna stále rozvíjí a Microsoft také oznámil unifikaci těchto .NET platforem. Na Obrázku 5.2 můžeme vidět vývoj .NET Core se zároveň nově připravující se verzí .NET 5.0, která spojí všechny platformy pod jeden „kabát". Podle zpráv Microsoftu se bude jednat o další velkou kapitolu pro .NET, ale nehodlají opomíjet standardy, které momentálně platí. Tak jako Windows Forms, které jsou velmi staré, tak i v tomto případě můžeme očekávat velmi dlouhou údržbu. Když si uvědomíme složitost .NETu, tak i spoustu vývojových firem dlouhou dobu zůstane u starších standardů, než se nové technologie usadí.

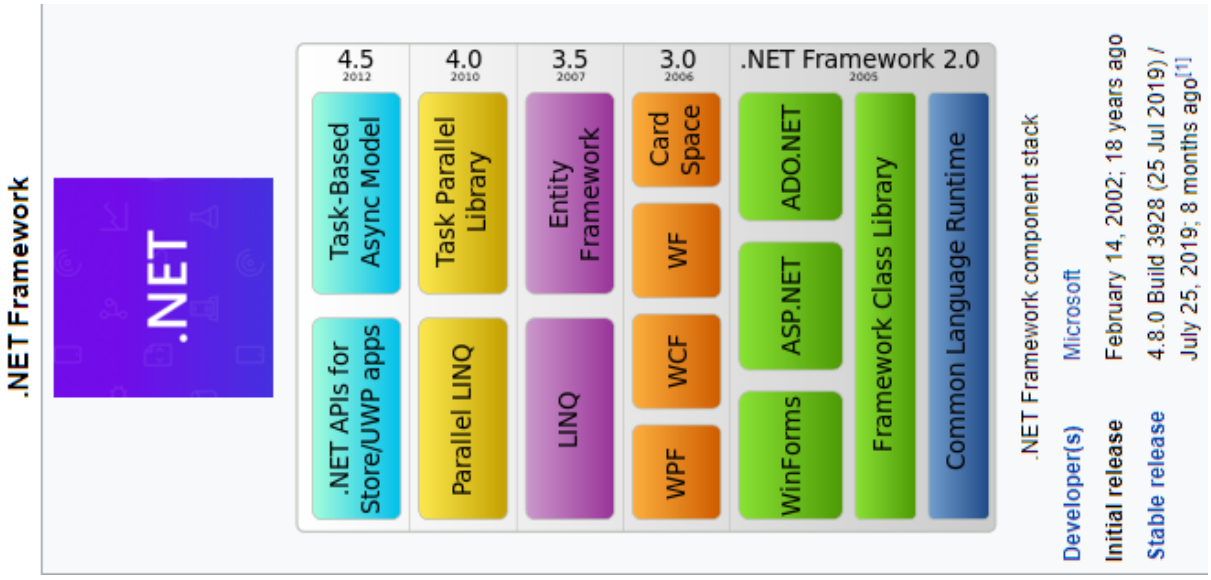
5.1.3 .NET Framework

.NET Framework[1] je originální implementací .NET, která podporuje webové stránky, služby, desktop aplikace a další na platformě Windows. .NET Framework podporuje pouze platformu

Version	Status	Latest release	Latest release date	End of support
.NET 5.0	Preview	5.0.0-preview.1	2020-03-16	
.NET Core 3.1 (recommended)	LTS	3.1.3	2020-03-24	2022-12-03
.NET Core 3.0	End of life	3.0.3	2020-02-18	2020-03-03
.NET Core 2.2	End of life	2.2.8	2019-11-19	2019-12-23
.NET Core 2.1	LTS	2.1.17	2020-03-24	2021-08-21
.NET Core 2.0	End of life	2.0.9	2018-07-10	2018-10-01
.NET Core 1.1	End of life	1.1.13	2019-05-14	2019-06-27
.NET Core 1.0	End of life	1.0.16	2019-05-14	2019-06-27

Obrázek 5.2: Verze .NET Core

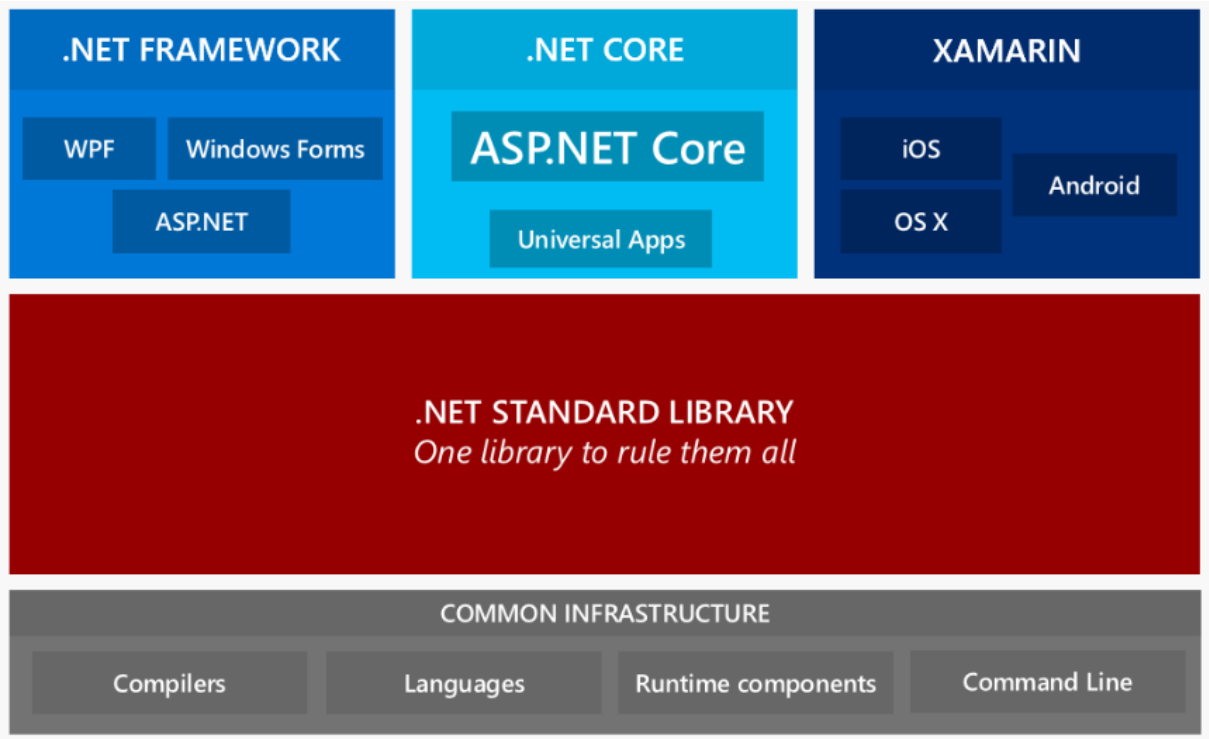
Windows a díky tomu má rozšířenou funkcionalitu v tomhle směru. Na obrázku 5.3 můžeme vidět vývoj .NET Frameworku od počátku.



Obrázek 5.3: .NET Framework - vývoj[1]

5.1.4 Abstraktní pohled[2]

Z obrázku 5.4 je vidět základní rozdělení celého .NETu. V další sekci (5.2) probereme detailněji konkrétní prostředí, která tyto struktury nabízejí: Windows Forms, WPF a UWP. Tato prostředí jsou pro tvorbu desktopových aplikací, ale jak již víme .NET podporuje i tvorbu webových stránek, které v našem řešení můžeme použít. Samozřejmě jsou zde i řešení pro mobilní zařízení, ale v poslední době je Microsoft řešení pro mobilní zařízení na ústupu, jak ze strany vývojáře, tak i z uživatelského hlediska. Nicméně je zde ASP engine, který slouží právě pro tvorbu zmíněných webových stránek. Detailnější rozdělení ASP je popsáno v sekci 5.2.4.



Obrázek 5.4: .NET - rozdělení[3]

5.2 .NET - Vývojové prostředí

.NET má několik programových prostředí, na kterých lze postavit aplikaci. V této sekci si přiblížíme tato prostředí a na závěr si nad nimi uděláme abstraktní pohled (5.2.6).

5.2.1 Windows Forms

Jedná se o open-source grafickou knihovnu tříd, která je součástí .NET Frameworku nebo Mono Frameworku. Nabízí na platformě Windows možnost vytvořit velmi bohaté klientské grafické prostředí. Jedná se již o docela starou technologii. První vydání bylo v roce 2002. Samozřejmě bylo od té doby stále rozvíjeno. Nabízí velmi jednoduché použití. Jinými slovy, můžeme si představit Windows Forms jako mřížku pixelů s pozicí vašich prvků, která je nastavena při době konstrukce (design time) podle velikosti a souřadnicím. To je velmi dobré, pokud aplikace běží vždy na stejném rozlišení. S vyvíjející se dobou se zvedá i rozlišení a rozlišení DPI se také mění. V takovém případě již nejsou Windows Forms nejvhodnější. Samozřejmě toto bylo dále vylepšeno v .NET Frameworku verze 4.7, který je jednou z posledních verzí. Takže i přes svoje stáří, Windows Forms je stále velmi dobře udržovaný[84].

Zároveň je potřeba dodat, že Windows Forms je hodně odlehčený UI design (lightweight) a přístupný .NET Core a plně použitelný pro .NET Framework. Windows Forms nepodporuje XAML. Dále akcelerace pro vykreslování uživatelského rozhraní není nejnovější.

Windows Forms tvoří, navzdory stáří, velmi dobrý způsob prezentace UI i v roce 2020. Pro jednoduché GUI je to velmi efektivní a rychlý způsob implementace s porovnáním s jinými možnostmi, nejen z .NETu. Také z hlediska paměťových nároků je tento systém velice nenáročný.

5.2.2 WPF

WPF (Windows Presentation Foundation) nabízí možnost vytvářet desktop klientské aplikace pro operační systém Windows s velmi širokými možnostmi. XAML je stavební prvek uživatelského rozhraní, nicméně VisualStudio nabízí i grafický designer. WPF bylo poprvé spuštěno v roce 2006. Jedná se o nástupce Windows Forms, které podporuje modernější prvky rozhraní, vylepšenou akceleraci vykreslování pomocí grafické knihovny DirectX a spoustu dalšího. Velmi důležitým zlepšením je oddělení business logiky od UI, které Windows Forms nedokáže poskytnout[86].

WPF je nynějším standardem. Příkladem je VisualStudio, které je právě napsáno pomocí WPF. Je daleko více flexibilní než Windows Forms. Díky XAMLu je editace a úprava rozhraní velice jednoduchá. WPF je velmi stará technologie, podobně jako Windows Forms, ale nabízí technologie, které dnes stále mají hodnotu a jsou obstojné vůči jakýmkoli jiným technologiím

pro tvorbu GUI na trhu.

5.2.3 UWP

Universal Windows Platform byl prvně představen na Windows Serveru 2012 a Windows 8. Nabízí možnost vytvořit uživatelské rozhraní, které poběží na více typech zařízení. Nyní je UWP součástí Windows 10. Hlavním záměrem vytvořit novou verzi, která by potencionálně nahradila Windows Forms a UWP bylo, škálování (DPI) a multimédia. Mezi to zapadá například podpora dotykové obrazovky[83].

Přes všechny tyto novinky vývoj UWP není úplně jistý a směr se několikrát změnil. Tak, jak obecně známe změnu z Windows 7 na Windows 8 a následně na Windows 10, tak v tomto právě figurovalo UWP, které se vývojem změnilo a momentálně se zaměřuje primárně na desktop aplikace.

5.2.4 ASP.NET[14]

ASP.NET rozšiřuje knihovny .NET o možnost vytvářet webové stránky. Prvotně byl spuštěn jako add-on IIS serveru. ASP.NET funguje na protokolu HTTP. ASP.NET je součástí .NET Frameworku a nabízí 3 styly pro vytváření webových aplikací: Web Forms, ASP.NET MVC a ASP.NET Web Pages.

5.2.4.1 Web Forms

Web Forms je rozšířeno pomocí akčních interaktivních modelů. Využívá Windows Forms a WPF. Nicméně nelze oddělit business logiku od uživatelského rozhraní.

5.2.4.2 ASP.NET MVC

ASP.NET dodává na MVC (Model-View-Controller) základní stavební prvky pro dynamické webové stránky. Tyto prvky jsou: uživatelské rozhraní (View), data (Model) a aplikační logika (Controller).

5.2.4.3 ASP.NET Web Pages

ASP.NET Web Pages kombinuje základní ASP s pomocí PHP. Nicméně nelze oddělit business logiku od uživatelského rozhraní.

5.2.5 ASP.NET Core[14]

Jedná se o novou verzi ASP.NET pro vývoj webových stránek. Může být spuštěn různými systémy jako jsou Windows, MacOS nebo Linux. Jedná se o kombinaci MVC Frameworku a WEB API v jedné programovací knihovně. Výhody oproti základní verzi ASP.NET je multiplatformnost a možnost hostování na serverech: Kestrel, IIS, HTTP.sys, Nginx, Apache a Docker.

5.2.6 Abstraktní pohled - analýza

V našem zadání bude potřeba server, na kterém bude databáze s daty. Zároveň potřebujeme uživatelské rozhraní, které bude zobrazovat data a kde bude možnost je editovat. Pro použití webového serveru může být použita jiná technologie. Vytvořit vše v v jedné technologii udrží celý projekt vhodně organizovaný. Použitím základního ASP.NET budeme limitováni, ale rozšířená nadstavba ASP.NET Core nabízí vhodný základ pro naše řešení. Další velkým přínosem je práce na modelu MVC, který ASP.NET Core nabízí. Snadno tedy můžeme webovou část naší aplikace udržovat.

Jedním z našich cílů je možnost multiplatformnosti. Jelikož .NET nabízí řešení i pro desktop, můžeme vytvořit ovládací aplikaci, která se bude připojovat právě na tento server. Detailní popis návrhu je podrobně popsán v kapitole 6.

Pokud tedy dojde na tvorbu desktop části aplikace máme 3 možnosti: Windows Forms, WPF nebo UWP. Z osobních zkušeností je Windows Forms velmi užitečná a snadno nasaditelná technologie, ale pro naše potřeby by mohla být již trochu zastaralá. My potřebujeme udržet naše uživatelské rozhraní co nejpřívětivější a pravděpodobně nebude ani tolik jednoduché. Dále je potřeba zmínit, že .NET má velice prohnutou křivku učení a díky tomu si hodně zakládá na komunitě uživatelů. Přestože, že .NET nabízí velmi silné schopnosti vývoje a jazyk C#, nabízí velice přívětivou syntaxi s porovnáním k jiným jazykům, není tolik preferovaný z důvodů složitosti. V případě, že se máme rozhodnout mezi WPF nebo UWP, tak je potřeba zmínit nejistý vývoj UWP a také to, že UWP nemá tak silnou komunitu jako jejich předchůdci - WPF a Windows Forms. Vývoj freelancera bez solidního základu velké firmy by mohl být velmi obtížný. Proto z osobního hlediska se přikláním k řešení WPF, které nabízí, nijak zvlášť rozdílné schopnosti vůči UWP a je v dnešní době stále pevným základem pro vývoj desktop aplikací.

5.3 SQL Server (MSSQL)

Microsoft SQL Server je relační databázový systém vyvíjený společností Microsoft. Je to software s primární funkcí ukládat data a poskytnout data na požádání jiným aplikacím dat na požadavek pro ostatní aplikace. SQL server je možné provozovat v linuxových kontejnerech s podporou platformy Kubernetes nebo v systému Windows. V našem řešení bude sloužit jako výchozí volba, nicméně bude jej možné snadno zaměnit za jiné řešení - například MySQL, která je zdarma [78].

5.4 SQLite databáze

Na rozdíl od běžných databází jako jsou MySQL/MariaDB nebo PostgreSQL, které běží jako služba, SQLite je pouze malá knihovna. Databáze je ukládána do jednoho obyčejného souboru na disk, který je bez jakýkoliv problémů přenositelný mezi operačními systémy. Databáze je určena pro menší aplikace nebo pro aplikace, kde není kladen důraz na uživatelská oprávnění přístupu, neboť neobsahuje některé funkce, které obsahují ostatní velké databázové systémy. Často se využívá u aplikací pro mobilní zařízení nebo jako jednoduché uložiště pro nenáročné offline aplikace[79].

5.5 Bearer Token JWT (autorizace)

Bearer autorizace je známá také jako tokenová autorizace, je HTTP autentizační schéma, které zahrnuje bezpečností tokeny nazývané Bearer tokeny. Název „Bearer autorizace"se dá rozumět jako „dej přístup nosiči (bearer) tohoto tokenu". Bearer token je zakódovaný řetězec znaků, většinou generovaný serverem v odpovědi na přihlášení. Pro autorizaci je potřeba vždy token zaslat v hlavičce HTTP požadavku[15].

JWT token má strukturu rozloženou na 3 části oddělené tečkou. První část je zakódovaná pomocí base64 a udává, jaký algoritmus je použit pro zabezpečení klíče a o jaký typ tokenu se jedná. Druhou částí je takzvaný „payload"zakódovaný v base64. Jedná se o data, která s sebou token může nést. Je zde pouze jeden parametr, který je povinný a to je JTI[46]. Poslední částí (veřejný klíč) je část nesoucí zakódovaný podpis soukromého klíče pomocí algoritmu specifikovaného v první části tokenu [47].

5.6 NPM

NPM je balíčkovací správce primárně pro javascriptové prostředí Node.js, které je představeno klientskou příkazovou řádkou. Pomocí tohoto balíčkovacího správce jsme schopni udržovat náš webový projekt bez zbytečností a snadno udržovatelný pro knihovny třetí strany. Díky tomuto správci můžeme snadno říci, které knihovny chceme do našeho projektu vložit a automaticky při kompilaci projektu nám tyto knihovny, zvolené verze, naimportuje do našeho projektu. Není nutné vše udržovat ručně[64].

5.7 NuGet

Jedná se o balíčkovací manager zabudovaný přímo do prostředí VisualStudio. Jedná se o nativní balíčkovací systém pro .NET a nabízející instalaci spousty rozšíření, která jsou přímo od Microsoftu, tak i od jiných uživatelů, přímo do vašeho projektu, bez ruční instalace [65].

5.8 SASS

Jedná se o preprocesovanou nadstavbu pro CSS (Cascading Style Sheets). Sass nabízí dvě možné syntaxe. Jedna je podobná značkovacímu jazyku HAML[34], která používá odsazení pro oddělení kódových bloků a novější syntaxi nazývanou SCSS, která vychází ze základního CSS [72].

Nabízí možnost tvorbu funkcí a proměnných, které usnadní přehlednost a vývoj stylů pro webovou stránku.

5.9 Bootstrap 4

Jedná se o CSS knihovnu s řadou předdefinovaných tříd, která zjednodušší stylování webové stránky. Jeden z velkých kladů této knihovny je velmi jednoduché zajištění responzivity pro různá zařízení. Jedná se o jednu z nejvíce populárních knihoven pro CSS [16].

5.9.1 Bootstrap-Table

Bootstrap rozšíření, které zajistí speciální stylování pro tabulky. Nabízí funkcionalitu, která se u běžných tabulek nachází, jako je například: Filtrování, řazení nebo aktualizování dat v

tabulce. Použití je velmi jednoduché a nabízí 2 způsoby implementace: pomocí JavaScriptu nebo pomocí atributů v HTML [17].

5.10 JQuery

JQuery je JavaScriptová knihovna - rychlá, malá a bohatá na funkce. Správa front-endu je díky ní jednodušší. Knihovna funguje napříč všemi prohlížeči. Jedná se o jednu z nejpopulárnějších knihoven pro tvorbu webových stránek na straně front-endu [44].

5.11 Knihovna Chart.js

Jedná se o jednoduchou a velmi efektivní knihovnu pro JavaScript. Tato knihovna nabízí snadné a působivé vykreslení mnoha druhů grafů pro zobrazení dat [36].

5.12 ASP.NET Razor

Razor je programovací předpis definující pravidla pro tvorbu dynamických webových stránek v ASP.NET pomocí C#. Poprvé byl tento engine pro správu uživatelského rozhraní vydán v roce 2011 jako část Frameworku MVC 3 [13].

5.13 Xceed's WpfToolkit

Jedná se o nejpopulárnější kolekci přídavných kontrolních prvků pro WPF (5.2.2). Naleznete zde prvky jako formulářové pole pro čísla různých formátů nebo například výběr barvy. Jinými slovy se jedná o velmi důležitou součást vyvíjené aplikace, která potřebuje pokročilé vstupy od uživatele [88].

5.14 Microsoft Entity Framework

Entity Framework (EF) je open-source objektově relační mapovací (ORM) framework. Jedná se o rozšíření, které dodává vašemu kódu v .NETu možnost přidat „obálku"okolo vaší databáze. Díky tomuto řešení máte možnost snadno přistupovat k datům bez potřeby dotazovacího jazyku SQL. Přistupujete pomocí klasické objektově orientované syntaxe jazyku C# [30].

Datové tabulky v databázi se generují automaticky podle vašich nadefinovaných datových modelů. Každý datový model představuje jednu tabulka vaší databáze. Každý atribut datového modelu představuje atributy v tabulce.

5.15 IXS DNA Framework

Framework postavený v .NET Standardu obsahuje abstraktní funkcionality, které se hodí v každém projektu. Nejdůležitější součástí je logovací systém a odesílání HTTP požadavků. Jedná se o převzatý Framework [27], který jsem osobně upravil do svých vlastních požadavků a opravil drobné chyby.

Řešení je plně funkční pro nasazení do našeho projektu, ale tak zároveň i do jiných projektů. Framework je veřejně dostupný a kdokoli jej tak může použít. Framework je open-source a kdokoli jej tak může převzít a vést vlastním směrem, jak licence dovoluje.

6. Návrh systému

V této kapitole bude popsán detailní návrh aplikace, jaké technologie použít (na konkrétních místech), co by aplikace měla obsahovat, jak bude aplikace zabezpečena atd. Abychom tento návrh mohli popsat, je potřeba se obeznámit s detaily vývoje. Hlavním cílem je postavit solidní základ, ze kterého se bude moci aplikace dále jednoduše rozvíjet.

Na začátku byl plán vybudovat podobně postavenou aplikaci jako je ta, která v naší cílové firmě již řadu let funguje. Jedná se o desktop aplikaci se vzdálenou databází na serveru, kam jsou data odesílána a následně je zde webová aplikace, která data promítá do firmy. Vývoj tedy začal s plánem vybudovat WPF aplikaci. Avšak po určité době jsem osobně narazil na technologii ASP.NET, se kterou jsem doposud neměl zkušenosti. Napadlo mě využít právě této technologie a vybudovat API server, který bude data z databáze distribuovat a tím využít další .NET součást. Nicméně, tento nápad udělal z původní desktop aplikace pouze ovládací aplikaci. Jádro celého projektu tvoří právě zmíněný server.

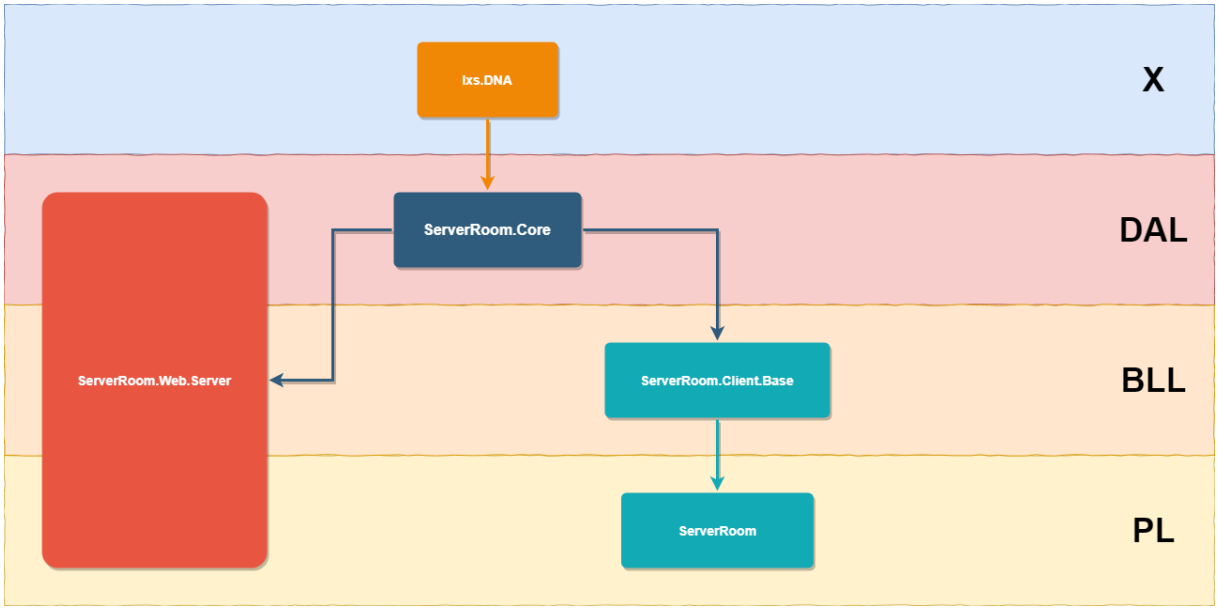
Ve výsledku je plán pro konstrukci webového serveru, který bude nabízet API rozhraní, rozhraní zobrazení dat do firemního prostředí a ovládací desktop aplikaci. Prakticky se dá uvažovat i tak, že desktop aplikace není potřeba a vše se dá udělat jako webové aplikace. Jsou zde dva hlavní důvody, proč jsem zvolil zachování desktop aplikace. Prvním je fakt, že web server bude (může být) přístupný externě z firmy. Na dálku chceme data pouze číst a právo změny chceme udržet pouze z fyzické serverové místnosti, kde bude systém běžet - neumožnit ovládat systém z venčí. Ovládací aplikaci je stále možné mít dostupnou pomocí RDP (Remote Desktop Protocol). Druhým důvodem je vlastní zkušenost. Nikdy jsem neměl příležitost postavit vlastní API server, který by zároveň obsluhoval další části aplikace.

6.1 Návrh řešení (klient/server)

Již víme, že .NET funguje na modelu MVVM a jeho serverová část - ASP.NET - na modelu MVC. Jedná se o podobné, ale zároveň lehce odlišné struktury. Hlavním cílem bude využít těchto modelů co možná nejvíce a postavit přehlednou aplikaci. Jedním z hlavních návrhů je

vystavět datovou vrstvu, kterou využije každá část aplikace - server a klient. Tuto část můžeme nazvat „Core". Tato část bude využívat mnou vytvořený DNA framework[27]. V projektu Core bychom rádi drželi všechny datové modely a rozhraní pro abstraktní funkcionalitu projektu.

Celý plánovaný návrh architektury projektu je popsán na obrázku 6.1. Projekt se rozděluje na několik menších, jak již bylo naznačeno a bude zmíněno v sekcích 6.1.1 a 6.1.2. DAL reprezentuje „Data Access Layer"- Datová vrstva, kde aplikace přistupuje k datům v databázi. BLL představuje „Business Logic Layer"- Vrstva logiky aplikace. Zde je prováděná hlavní logika, která reprezentuje naší aplikaci. PL reprezentuje „Presentation Layer"- prezentační vrstva. V našem případě to znamená GUI. Zde se v poslední řadě nachází vrstva označená písmenem X. Tato vrstva v sobě drží knihovnu Ixs.DNA[27], ze které celý projekt benefituje. Ixs.DNA poskytuje celému projektu programové struktury, které se vyskytují běžně v každém projektu - například logování. V každém projektu máme v ideálním případě logovací systém a abychom nemuseli vždy pro každý projekt definovat znovu to samé, tak implementujeme knihovnu Ixs.DNA (knihovna je stavěna pro jakýkoli projekt, není výslovně stavěna jen pro ServerRoom).



Obrázek 6.1: Plánovaná architektura projektu

6.1.1 Návrh serveru

Projekt Web.Server bude využívat projektu Core, který mu bude poskytovat veškeré datové modely, které webový projekt potřebuje k odesílání dat přes API nebo přímo k práci s databází.

Webový projekt je rozdělen do částí MVC, kde modely tvoří pouze databázový kontext, zbylé modely jsou v projektu Core. Budeme potřebovat controller pro API a další pro zobrazování dat (Viewer).

6.1.2 Návrh klienta

Klientskou část aplikace můžeme rozdělit na několik částí. Z vlastní vůle chceme rozdělit business logiku a prezentační část aplikace. Použitím WPF chceme využít modelu MVVM, který celému ekosystému aplikace pomůže udržet data, logiku a prezentaci od sebe. Dále WPF bude potřeba naimplementovat pomocí .NET Frameworku. Abychom snadno udrželi logiku stranou, můžeme vytvořit další projekt pod názvem Client.Base, který se postará o logiku klientské části. Zároveň Client.Base bude využívat projektu Core. Díky tomu, že rozdělíme klientskou část aplikace na dva projekty - .NET Framework a druhý NET Core, docílíme toho, že programátor nebude schopen využít knihovny pro GUI v části, kde má být pouze business logika. Jedná se o takovou ochranu programátora, aby organizoval projekt tak, aby psal věci tam, kde mají správně být. Proto základní projekt klienta bude obsahovat části vztažené pouze k GUI. Client.Base pak bude obsahovat klientské části, které přímo neinteragují s GUI - například view modely (controllery).

6.2 Návrh komunikace se serverem

Plánujeme server poskytující API. Pomocí tohoto API můžeme se serverem snadno komunikovat. Vytvoříme základní rozhraní API, které bude sloužit pro přihlášení nebo registraci. Vytvoříme následně druhou chráněnou část API, která bude dostupná pouze pod ověřeným přihlášeným uživatelem. Pomocí API můžeme získávat všechna data, která jsou potřebná pro plnou kontrolu aplikace.

Hlavním cílem je udržet data synchronizovaná se serverem. Proto na klientské desktop aplikaci zavedeme aktualizační smyčku, která pravidelně bude žádat testovaná data. Kdykoli bude interakce od uživatele vyžadovat nová data nebo poslat žádost o další, tak vždy vytvoříme nový HTTP požadavek a přiložíme do něj řádnou autorizaci.

6.3 ZABEZPEČENÍ

V našem případě je potřeba zabezpečit aplikaci na třech místech: Webové rozhraní, klientské rozhraní a API.

Z osobních zkušeností, dobrou volbou pro identifikaci uživatele při HTTP požadavku, je pomocí tokenu. Použijeme JWT[47]. Takový token můžeme sestavit z uživatelských dat - například uživatelské jméno nebo ID - tato data jsou snadno veřejně dostupná. Musíme ještě přidat JTI[46], který je nezbytně nutný pro identifikaci JWT. Token se přikládá do hlavičky HTTP požadavku, kde při příchozím požadavku na server je předán do rukou frameworku ASP.NET, který se již postará o validaci příchozích požadavků na server.

Přihlášení do webového prostředí bude zajištěné pomocí cookies. Celé řešení dokáže ASP.NET samostatně vyřešit za nás a my mu můžeme nastavit pouze základní parametry, jako je například expirace.

Klient má vlastní lokální databázi, kterou může použít právě pro držení dat o přihlášení. Pro přihlášení je potřeba zavolat příkaz na server přes API. API nám vrátí uživatelská data, včetně tokenu. Tento token následně uložíme do databáze a pro ověření přihlášení uživatele použijeme právě tato data. Díky tomuto řešení nebude potřeba se vždy dotazovat serveru na ověření uživatele, ale můžeme uživatele držet přihlášeného na klientské aplikaci. Při jakémkoli dotazu na server o data vždy přiložíme token uživatele do hlavičky HTTP požadavku.

Dalším plánem by bylo vytvořit role pro uživatele. Rozdělíme podle funkcí - správce a pozorovatel - kde správce má plné oprávnění na úpravu dat a pozorovatel může pouze data sledovat. Nicméně se rozdělení samotných rolí může ještě v budoucnu upravit, protože se jedná o nadstavbu projektu.

6.4 UKLÁDÁNÍ DAT V SYSTÉMU

Naším úkolem je ukládat data, která budou obsahem naší aplikace tzn. zařízení s jejich testy. Samozřejmě nesmíme zapomenout na data, která budou zajišťovat zabezpečení, jako jsou například přihlašovací tokeny.

Díky využití plné síly .NETu s pomocí ASP.NET na straně serveru se nabízí možnost využití

databázové struktury vygenerované přímo od .NETu, uváděné pod názvem Identity. Identity je část systému ASP, která zajišťuje autentizaci a autorizaci pro uživatele vytvářené aplikace v .NETu. Každá základní databáze je složena z několika tabulek, jako je například tabulka uživatelů nebo jejich rolí. Tuhle možnost můžeme rozšířit o další naše potřeby, protože je možné tabulky v jisté míře přizpůsobit. Tabulka uživatelů postrádá atributy s hodnotou křestního jména a příjmení. Z toho důvodu přidáme tyto atributy do tabulky.

Detailnější popis databázového modelu bude popsán v následujících podkapitolách.

6.4.1 Návrh databázového modelu

V celém projektu jsou využity dvě databáze - jedna na serveru a druhá je součástí klientské aplikace. První je hlavní databází běžící na Microsoft SQL Serveru. Výběr Microsoft SQL Serveru se dal očekávat a je velmi přehledné udržet vše pod stejnou střechou .NETu. Je však možné databázi snadno změnit díky použití Entity Framework. Tato databáze udržuje hlavní data aplikace. Jedná se o data firmy, jaká zařízení systém spravuje a jaké testy na daných zařízení probíhají. Druhou databází je databáze SQLite, která běží na klientské desktop aplikaci. Tato databáze slouží k uložení uživatelského nastavení aplikace a udržování dat přihlášeného uživatele. Návrh celého datového modelu je zobrazen na obrázku 6.2.

- **AspNetUsers** - Základní tabulka vygenerovaná pomocí systému Identity[49] (podrobnosti v sekci 6.4). Tato tabulka již v sobě má velké množství atributů, které využijeme, ale potřebovali jsme i další přidat. Vyjmenuji jen ty nejdůležitější: uživatelské jméno, email, hash hesla, křestní jméno, příjmení a telefonní číslo. Zbylé atributy jsou nezbytné pro funkčnost systému Identity a zajišťují autentizaci a autorizaci uživatele.

- **Devices** - Hlavní tabulkou našich dat je tabulka Devices. Tato tabulka nese informace o zařízeních a jejich nastavení. Při smazání záznamu zařízení by měly být záznamy v Routiners vztažené k danému zařízení smazány také. Nyní vyjmenuji atributy, které tabulka obsahuje: IP adresa, popis, indikace spuštěného zařízení, status jako hodnocení posledních testů, poslední vyhodnocení statusu, poslední „nezelený"status, poslední změna „nezeleného"statusu, interval testů, timeout testů, poznámka a čas editace záznamu.

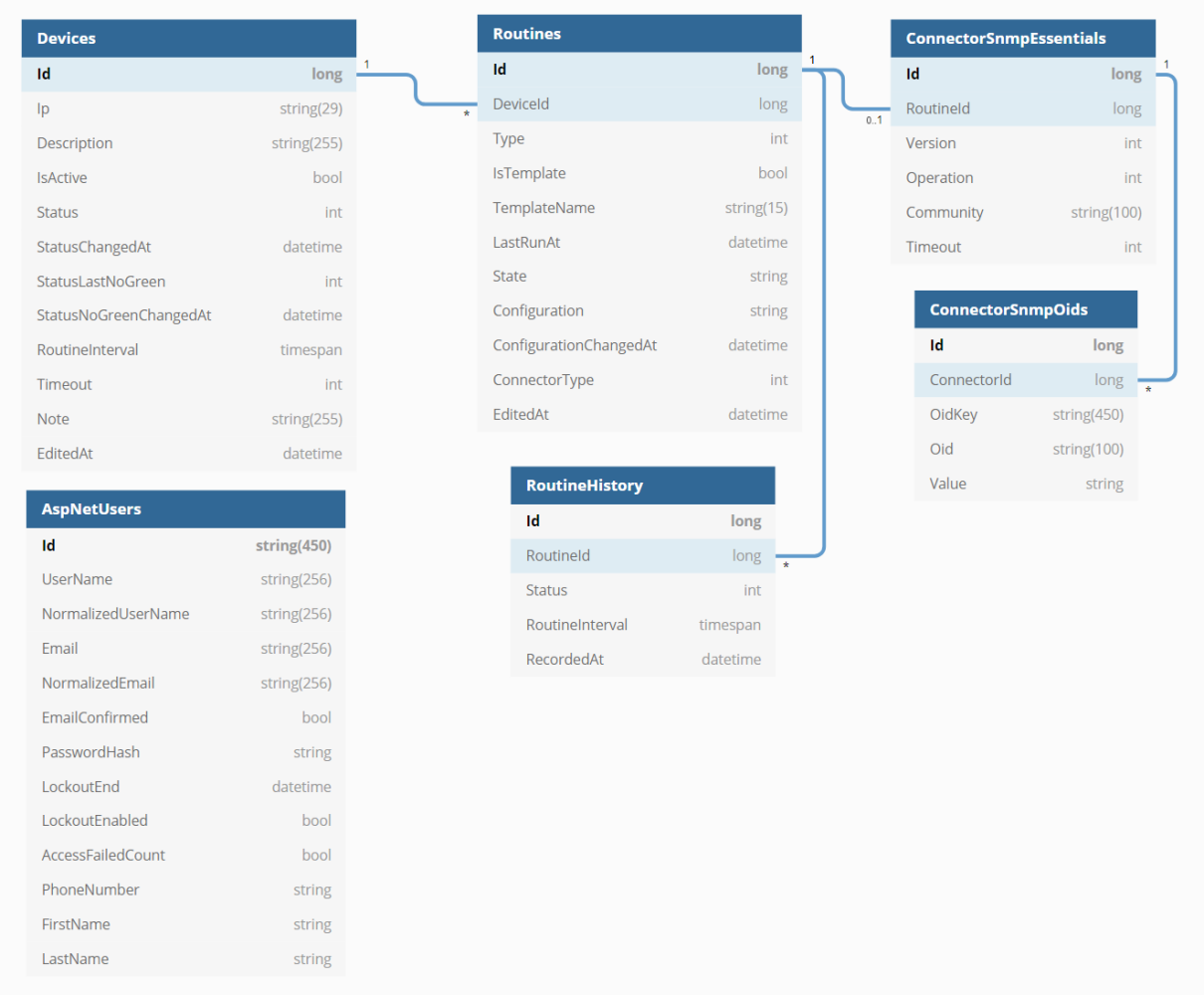
- **Routines** - Seznam všech rutin (testů) pro dané zařízení (Devices). Tabulka je ve vztahu 1:N k tabulce Devices, kde může být více rutin pro jedno zařízení, ovšem pouze jeden typ testu pro jedno zařízení. Každá rutina má svůj typ - například PING nebo UPS.

Typ rozlišuje průběh a způsob testování a vede to k tomu, že by každá rutina mohla mít vlastní tabulku, ale tato překážka je detailně popsána v sekci 6.4.1.1. Rutina může být představena také pouze jako template. To znamená, že nepatří žádnému zařízení, ale je možné vyfiltrovat template rutiny a v aplikaci je pak nabídnout jako výchozí nastavení při přidělování rutin. Tabulka rutin má následující atributy: ID zařízení, typ, bool hodnota jestli se jedná o template, poslední spuštění, stavová informace (JSON), konfigurace (JSON), čas změny konfigurace a čas editace.

- **RoutineHistory** - Záznam veškerého testování rutin, které proběhlo. Zaznamenávají se zde kladné i záporné výsledky testů. Díky těmto datům jsme schopni vytvořit statistiky o testovaných datech a zobrazovat je uživateli. Tabulka má 3 atributy: RoutineId, Status, RecordedAt, RoutineInterval. Z atributů možná již trochu vyplývá, že tabulka je spojená vazbou 1:N s tabulkou **Routines**.

- **ConnectorSnmpEssentials** - Tabulka obsahuje záznamy o tzv. „konektorech"(connectors) - speciálně se zaměřením SNMP. Konektory představují volitelnou možnost pro rutiny, pro získání dat potřebných k jejich testování, pokud připojení je potřebné. Tabulka je spojená s tabulkou **Routines**, kde může existovat žádný nebo pouze jeden záznam (ConnectorSnmpEssentials) pro jednu rutinu (Routines). Atributy tabulky jsou: RoutineId, Version, Operation, Community, Timeout. S výjimkou cizího klíče RoutineId, se jedná o nutné parametry k připojení se přes protokol SNMP.

- **ConnectorSnmpOids** - Nezbytná tabulka obsahující tz.v OID, které identifikují požadovaná data při SNMP připojení. Tabulka je spojená vazbou 1:N s tabulkou **ConnectorSnmpEssentials** a tvoří seznam OID pro daný SNMP konektor. Atributy tabulky jsou: ConnectorId, Oid, OidKey a Value. ConnectorId je cizí klíč, Oid je samotný OID pro SNMP, OidKey je identifikace OID pro programátora (systém) a v poslední řadě Value specifikuje volné pole, které poslouží při operaci SET, kde posíláme data pomocí protokolu SNMP.



Obrázek 6.2: Datový model

6.4.1.1 Řešení komplikace s databázovým modelem

Nastal bod, kde máme velmi těžké rozhodnutí ohledně modelu naší databáze. Jedná se o tabulku Routines, která obsahuje rutiny (testy) pro zařízení. Avšak ne všechny testy mají stejné parametry (atributy v databázi). Proto máme několik možností:

- **Držet se správného designu databázového modelu** - Každá rutina bude mít vlastní tabulku (např.: Routine_Ping, Routine_UPS, atd.). Pro zařízení (Devices) by následně byla vytvořena spojovací tabulka, kde bude zaznamenáno do jaké tabulky směřují zaznamenané ID rutiny.

- **Držet se správného designu databázového modelu, s menšími výjimkami** - Další možností je mít jednotnou tabulku Routines, která bude obsahovat atributy pro každý

test. To znamená, že záznam bude mít nevyužité atributy v závislosti na tom, jestli patří danému typu rutiny nebo ne.

- **Porušení pravidel datového modelu** - Pouze jedna tabulka, kde bude atribut konfigurace a stav. Tyto atributy budou hodnotou JSON. Chybné je, že ukládáme více hodnot v jedné buňce.

Jak již předchozí sekce 6.4.1 napovídá, možnost, pro kterou jsem se po dlouhém rozhodování přiklonil je třetí vybraná možnost a to porušení zásad datového modelu. Nebylo to jednoduché rozhodnutí, ale v našem řešení velmi účinné. Jde nám hlavně o jednoduchou správu aplikace. Pro deserializaci a serializaci dat vytvoříme rozšířené metody nad datovým modelem rutin a bude velmi snadné udržovat tato data přehledně uložena ve správných datových strukturách. Konfigurace i stav budou mít svoje vlastní datové modely, do kterých zmíněné metody budou převádět uložený JSON.

Hlavní nevýhodou předchozích metod je správa. Kdykoli bude potřebná změna v aplikaci (přidání atributu nebo nového typu rutiny), tak je nezbytný zásah do databáze. To by ve velké firmě nedělalo takový problém. Nicméně je potřeba podotknout, že tento software bude aktivně využívaný a bude potřeba jej nadále udržovat a aktualizovat. Správa toho produktu bude záviset pouze na jednom člověku. Druhý pohled je takový, že při každé úpravě bude potřeba zásahu do databáze a kódu. Vypustíme-li nutnost editace databáze, tak nám zbude jenom kód. Jinými slovy, významná část práce při aktualizaci je upuštěna. Další věc je, že při každém návratu ke kódu je potřeba znovu si připomenout, jak vše fungovalo a změna nemusí být okamžitá. Tím, že vynecháme zásah do databáze, se vyvarujeme možným nepříjemným okolnostem, které by mohly nastat.

Detailnější popsání implementace, které jsem zvolil, je popsáno v sekci implementace 7.6.

6.4.2 Práce s databází

Máme zde dvě databáze. SQLite na straně klienta, která je velmi jednoduchá, nenáročná a výkonná. Je možno ji použít na nejrůznějších systémech, takže s tímto řešením můžeme počítat třeba i při rozšiřování aplikace na jiné platformy.

Druhou a hlavní databází je Microsoft SQL Server, kterou lze nahradit i jinými databázovými systémy. Jelikož použijeme k řešení .NET Entity Framework, tak je velice snadné změnit

pouze jednu část kódu, která definuje použitou databázi. Entity Framework podporuje několik druhů databází, mezi které ještě patří například: MySQL nebo PostgreSQL. V budoucnu je možné přidat řešení, že si uživatel zvolí, v které databázi chce ukládat svá data, protože logika k tomuto řešení není tolik složitá, díky Entity Frameworku.

6.4.3 Ukládání uživatelského nastavení

Na straně klienta funguje model MVVM, kde můžeme vytvořit ViewModel aplikace. Tento aplikační ViewModel může být singleton (samostatná instance třídy, která se v kódu vyskytuje maximálně jednou), který bude uschovávat důležitá aplikační data včetně dat uživatelského nastavení. Tato data budou následně ukládána do zmíněné databáze SQLite, která je na straně klienta. Data, která bude potřeba ukládat jsou: poslední rozměry okna, poslední otevřená stránka aplikace a na závěr informace, jestli uživatel již prohlídnul změnu v aplikaci po aktualizaci nebo odsouhlasit různá potvrzení.

7. Popis implementace

V této kapitole bude popsán detailní popis implementace. Bude rozebrána celá kostra projektu a řádně popsána. Pokusíme se zajít i do detailů okolo kritických a klíčových částí kódu. Avšak celkový popis bude stále pouze pouhá abstrakce a nebude zabíhat přímo do detailního popisu konkrétních metod. Pro takové je vyhrazena sekce 7.8. I když se to na první pohled nezdá, zvlášť z pohledu uživatele, tak pod pokličkou je kód aplikace, který již není natolik jednoduchý jako vypadá zevnějšek aplikace.

Hlavním účelem celé kapitoly je poukázat na myšlenku v pozadí celé konstrukce a jak je možné aplikaci snadno spravovat jako programátor. ukážeme si části kódu, které využívají našeho databázového modelu a v poslední řadě si pak ukážeme, v čem je výhoda námi zvoleného databázového modelu.

7.1 PROJEKTY

Celá aplikace, jak již víme, je rozdělena do dvou částí: server a klient. Abychom docílili správného návrhu aplikace, tak celý projekt je rozdělen do několika menších částí (projektů). Toto řešení je z důvodu docílení správného vrstvení, které můžeme vidět na obrázku 7.1. Jedná se o tří-úrovňovou architekturu [9]. V našem případě funguje na podobném principu. Nachází se zde celkem 5 vrstev: X (knihovny), DAL (datová vrstva), BLL (vrstva business logiky), PL (prezentační vrstva) a Y (testy). Každá vrstva reprezentuje nějaký projekt. Nyní si tyto projekty přiblížíme:

- **Ixs.DNA** - Převzatá knihovna (5.15), kterou jsem vlastnoručně upravil do vlastních požadavků a opravil drobné chyby. Knihovna je stavěna pro široké použití v jiných projektech. Poskytuje celému projektu programové struktury, které se vyskytují běžně v každém projektu - například logování. V každém projektu máme v ideálním případě logovací systém a abychom nemuseli vždy pro každý projekt definovat znovu to samé, tak implementujeme knihovnu Ixs.DNA.

- **ServerRoom.Core** - Tento projekt funguje jako datová vrstva, kde máme definované všechny datové modely. Dále se zde nachází rozhraní, která definují základní struktury,

jako například: souborový správce nebo správce vláken (task). Nalezneme zde také užitečné rozšiřující (extension) metody. Například přidávají vlastní metody práce s textovými řetězci a najdeme je mezi možnostmi řetězení metod. Obecně, Core projekt obsahuje vše, co je sdílitelné mezi všemi projekty. V sekci 7.2.2 naleznete podrobnou implementaci projektu.

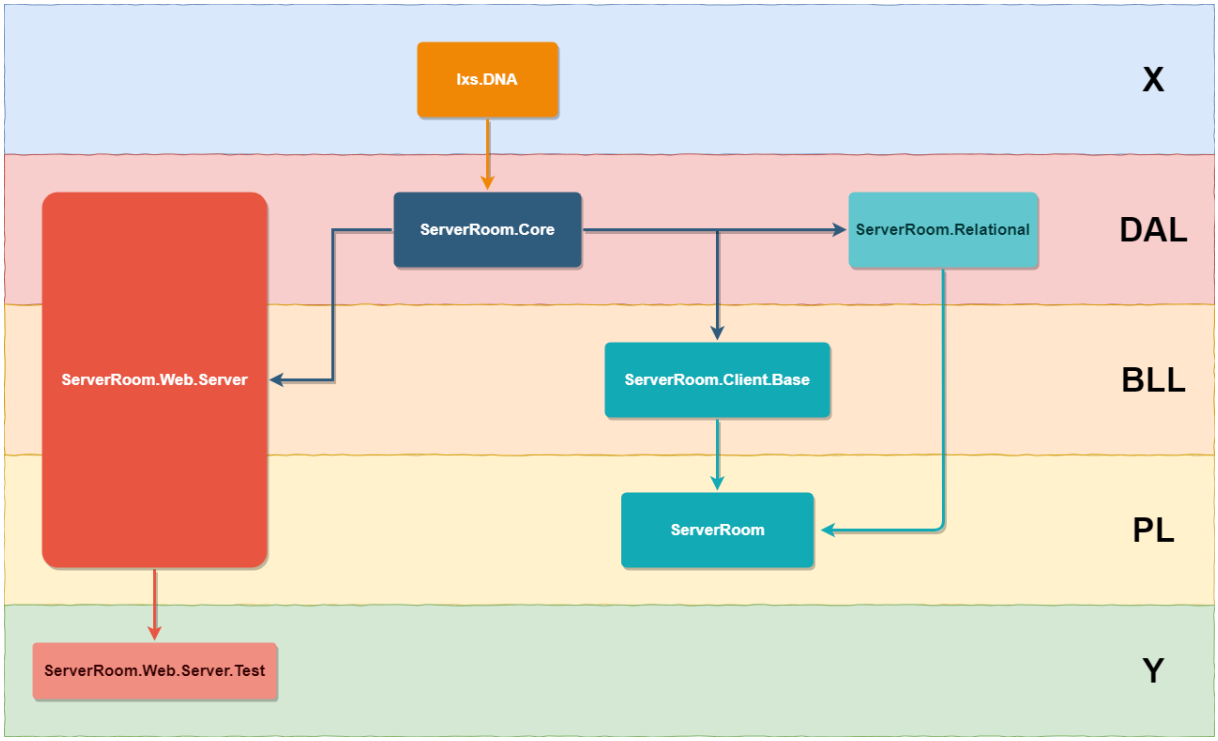
- **ServerRoom** - Jedná se o základní projekt klientské aplikace. Avšak tento projekt obsahuje pouze startup funkce projektu a programové části spjaté s uživatelským rozhraním. Podrobně rozebraná implementace celého projektu je obsažena v sekci 7.3.1.

- **ServerRoom.Client.Base** - Projekt obsahuje ovládací prvky pro klientskou aplikaci (tj. projekt ServerRoom 7.3.1). Tento projekt obsahuje to, co základní projekt klientské aplikace neobsahuje a to jsou view modely. Veškerá business logika klienta se nachází právě zde. Podrobnější implementace v sekci 7.3.2.

- **ServerRoom.Relational** - Velmi malý projekt obsahující jen několik tříd. Tento projekt pracuje s daty v rámci klientské aplikace, tj. ukládání dat do lokální databáze. Implementace projektu je zmíněna v sekci 7.3.3.

- **ServerRoom.Web.Server** - Hlavní projektová část projektu webového serveru. Projekt je stavěný na architektuře MVC, takže pokrývá 3 hlavní vrstvy naší architektury. Projekt využívá frameworku ASP.NET, který celou práci v MVC velmi ulehčuje a správa je následně velmi jednoduchá. Implementace celého projektu je rozepsána v sekci 7.4.1.

- **ServerRoom.Web.Server.Test** - Jedná se o pomocný projekt webového serveru pro vývoj, ve kterém nalezneme strukturu připravenou pro testování kritických částí webového serveru. Sekce 7.4.2 podrobněji rozebírá implementaci projektu.



Obrázek 7.1: Architektura projektu

7.1.1 Dependency Injection (DI)

Dependency Injection [26] (zjednodušeně DI) v našem projektu využíváme velmi často. Hlavním důvodem je snadná správa velmi důležitých částí kódu, které stačí jednou definovat a následně k nim můžeme snadno přistupovat.

7.2 Abstraktní projekty

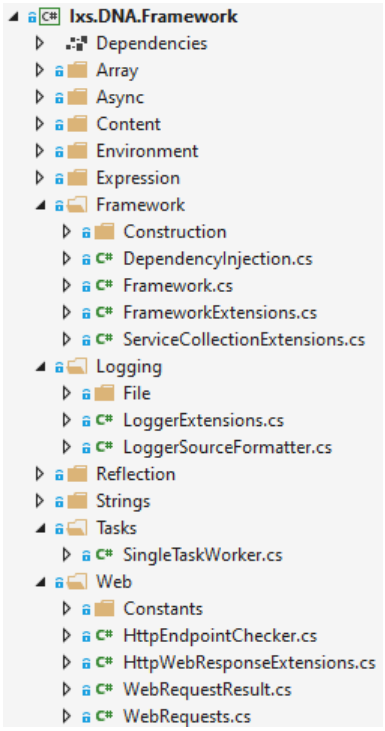
V této kapitole bude popsána struktura projektů aplikace, ze kterých vychází koncové aplikace, tj. jádro naší aplikace. Dále bude popsána architektura projektů, struktura jednotlivých částí a jejich nejvýznamnější třídy.

Abychom dokázali rozdělit vývoj aplikace, využili jsme přímo celé programové projekty k rozdělení vrstev. Jedna z těchto vrstev je vrstva datová. V našem případě se tohoto modelu držíme, ale například datová vrstva obsahuje datové modely, ale nikoli připojení do databáze. Na místo toho chceme, aby takový projekt fungoval jako projekt, který bude sloužit všem nižším projektům.

7.2.1 Projekt: Ixs.DNA

Projekt Ixs.DNA není přímou součástí projektu, ale je potřeba jej také zmínit, protože naplňuje velmi důležitou část práce v celém našem systému projektů (5.15). Popis obsahu celého projektu viz. obrázek 7.2.

Velmi důležitou informací také je, že se jedná o tzv. „fork"již zpracovaného projektu [27]. Nicméně v základu měl program několik nedostatků a chyb, které bylo potřeba upravit. Z toho důvodu jsem vlastními silami projekt rozvětvil na separátní větev, kde pokračuji s vývojem já osobně.



Obrázek 7.2: Struktura projektu Ixs.DNA

Ixs.DNA poskytuje celému projektu programové struktury, které se běžně vyskytují v každém projektu - například logování. V každém projektu máme v ideálním případě logovací systém a abychom nemuseli vždy pro každý projekt definovat znovu to samé, tak implementujeme knihovnu Ixs.DNA (knihovna je určena pro jakýkoli projekt, není výslovně stavěna jen pro ServerRoom).

Projekt je implementován pomocí .NET Standardu. Díky tomuto řešení vždy zůstane

otevřený možnosti řešení pro více platforem.

Na obrázku 7.2 je vidět celá struktura s důležitějšími částmi, o kterých si řekneme více detailů. Všechny tyto části jsou popsány v níže přiložených podsekcích s názvy, které odpovídají kořenovým složkám projektové struktury.

7.2.1.1 Dependencies

Tento adresář je speciální složkou pro .NET, která obsahuje v projektu tyto důležité informace: který framework a jeho verze jsou používány pro tento projekt a hlavně, které balíčky třetí strany jsou dále využívány v tomto projektu. Velmi oblíbený balíčkovací systém pro .NET je NuGet[65], který se používá pro balíčky třetí strany. Nejedná se ovšem pouze o balíčky třetí strany. Naleznete zde veškeré připojení, které není přímo v daném frameworku, na kterém projekt běží.

Například Ixs.DNA je pro náš projekt důležitý zejména při logování. Zde máme již navržené logovací struktury, které lze můžeme snadno implementovat v nižší vrstvě našeho projektu. Využíváme pevného základu vytvořeného přímo od společnosti Microsoft, ale o tom více v sekci týkající se přímo logování 7.2.1.8. Proto se mezi importovanými balíčky v projektu nachází například: `Microsoft.Extensions.Logging`.

7.2.1.2 Array

Jedná se o rozšiřující pravidla metod pro všechna existující pole - „extension methods"[19]. Nalezneme zde přidanou hodnotu ve formě metod pro jakákoli pole, která v projektu používáme. V našem případě zde máme metody `Append` (přidání nového prvku na konec pole) a `Prepend` (přidání nového prvku na začátek pole), které zajišťují pro nás základní funkcionalitu, která na obyčejných polích běžně chybí.

7.2.1.3 Async

V dalším popisu práce se často ještě zmíníme o asynchronním zpracování, ale abychom si to trochu ulehčili, hodilo by se mít určité struktury již hotové. Proto v našem abstraktním projektu nalézáme složku Async, ve které se nachází nejvýznamnější třída `AsyncLock`. Tato třída obsahuje metody fungující na principu semaforu. Díky těmto „polotovarům"bude psaní jakéhokoli asynchronního zadání vždy jednoduší.

7.2.1.4 Content

V adresáři `Content` se nachází třídy představující jednoduché data modely jako schránky dat. V našem případě zde máme implementovanou datovou schránku `ContentItem`, kde můžeme snadno ukládat pohromadě dvě hodnoty textového řetězce.

7.2.1.5 Environment

Zde je konkrétní specifikace proměnných prostředí pro DI (7.1.1). Jedná se o velmi důležitou část, která specifikuje několik základních informací o naší aplikaci. Najdeme zde například informaci o tom, jestli aplikace zrovna běží ve vývojovém režimu nebo se již nachází v produkčním prostředí.

Informace, které zde specifikujeme by neměli být nijak konkrétní k logice dané aplikace, ale jedná se o informace, kterými můžeme specifikovat obecně každou aplikaci. Tyto informace jsou následně snadno dostupné v použitém řešení.

7.2.1.6 Expression

Jedná se o extension metody[19] pro speciální programovou strukturu v jazyce C - Expression. Při používání „Expressions"jsou takové metody velmi důležité, viz. více externě: [18].

7.2.1.7 Framework

Jedná se o jednu z nejdůležitějších částí, kterou poskytuje Ixs.DNA Framework. Jedná se o DI (7.1.1), která pomocí této implementace může být snadno implementována do aplikace používající tento framework.

Pomocí zde implementovaných struktur dokážeme velmi snadno použít DI v aplikaci. Jako příklad můžeme poukázat na implementaci v našem serveru (Obr 7.3). Díky tomu, že implementaci DI stavíme a rozšiřujeme na základu, který předpřipravila společnost Microsoft, tak daleko snadněji můžeme využít výchozí implementaci spouštění webového serveru. Zde již DI je implementovaná a my jediné, co potřebujeme udělat je, přidat naší část DI do již implementované části ASP.NET (MVC). Technicky to je, jednoduše řečeno to, co naše implementace Framework dělá. Vytvoříme rozšíření pro existující DI a na základě toho vložíme vlastní struktury, které v kódu chceme používat.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    return WebHost.CreateDefaultBuilder()
        // Add DNA Framework
        .UseDnaFramework(construction =>
        {
            ...
        })
        .UseStartup<Startup>();
}
```

Obrázek 7.3: Implementace DNA Frameworku pomocí DI na straně serveru

Pokud se znovu podíváme na obrázek 7.3, tak je zde místo tří teček, které můžeme nahradit implementací částí, které vystavíme speciálně pro implementaci přes DI. Náš DNA framework nabízí tuto implementaci například pro logovací systém. Pomocí snadných lambda výrazů dokážeme do celého projektu vložit celý logovací nebo jiný systém, který je nadefinovaný pouze na několika řádkách.

Abychom poukázali na hlavní sílu DI, tak zde je ukázka praktického použití DI. Na obrázku 7.4 můžeme vidět použití logovacího systému k logování zprávy (více o logování v sekci 7.2.1.8).

```
// Log it
FrameworkDI.Logger.LogTraceSource("Starting background device service...");
```

Obrázek 7.4: Praktického použití logování v kódu pomocí DI

Samozřejmě to není jen tak. Pro takový jednoduchý přístup je potřeba udělat ještě jeden mezikrok. Nativně můžeme získat uložená data v DI pomocí generické metody Get přímo z DI. Můžeme si však programování trochu zjednodušit, a to platí i pro testování a debugování aplikace. Pomocí „zkratkové třídy"můžeme velmi často používaná data z DI dostávat daleko jednodušeji. V projektu Ixs.DNA se jedná o třídu `FrameworkDI`. Jejím obsahem jsou podobné zkratky podobné těm na obrázku 7.5. Zároveň se u následujících projektů můžeme všimnout, že každý projekt má svoji zkratkovou třídu jako je právě `FrameworkDI` a to s jediným rozdílem, a to že na místo názvu „Framework"připadá jméno daného projektu, popřípadě samotné „DI", pokud se jedná o kořenový projekt (klient/server).

```
/// <summary>
/// Gets the default logger
/// </summary>
10 references
public static ILogger Logger => Framework.Provider.GetService<ILogger>();
```

Obrázek 7.5: Ukázka zkratky pro získání dat z DI

7.2.1.8 Logging

Další velkou část tohoto frameworku je logovací systém. Základem je logovací systém poskytovaný společností Microsoft a jeho funkcionalitu rozšiřujeme směrem, kterým potřebujeme. Přidáváme vlastní metody, které rozšiřují funkcionalitu logovacího systému o možnost logovat zdroj, kde byla logovací zpráva vyvolána. Jedna z takových rozšiřovacích metod je na obrázku 7.6. S takovým rozšířením také rozšiřujeme výstupní formátování logovací zprávy.

```
/// <summary>
/// Logs a debug message, including the source of the log
/// </summary>
/// <param name="logger">The logger</param>
/// <param name="message">The message</param>
/// <param name="eventId">The event ID</param>
/// <param name="exception">The exception</param>
/// <param name="origin">The callers member/function name</param>
/// <param name="filePath">The source code file path</param>
/// <param name="lineNumber">The line number in the code file of the caller</param>
/// <param name="args">The additional arguments</param>
0 references
public static void LogDebugSource(
    this ILogger logger,
    string message,
    EventId eventId = new EventId(),
    Exception exception = null,
    [CallerMemberName] string origin = "",
    [CallerFilePath] string filePath = "",
    [CallerLineNumber] int lineNumber = 0,
    params object[] args
    ) => logger?.Log(LogLevel.Debug, eventId, args.Prepend(origin, filePath, lineNumber, message, LogLevel.Debug), exception, LoggerSourceFormatter.Format);
```

Obrázek 7.6: Extension metoda logovacího systému pro logování zprávy úrovně TRACE

Hlavním rozšířením logovacího systému je logování do souboru. Kdykoli zalogujeme zprávu, tak je výstup poslán na konzoli a taktéž do souboru. Samozřejmě všechny tyto hodnoty jsou nastavitelné z prvotní konfigurace, která je vedená pomocí DI. S logováním do souboru také přichází riziko přeplnění disku, na kterém je soubor uložen. Abychom předešli pomalému zaplňování disku, tak je možné logovacímu systému říci limit, při jakém se má log začít postupně promazávat (rotační log).

7.2.1.9 Reflection

Jedná se o extension metody pro speciální programovou strukturu v jazyce C. Reflection může být užitečný zvláště ve chvíli, kdy potřebujete dynamicky vytvářet instance nebo typy v

kódu.

7.2.1.10 Strings

Zde definujeme rozšiřující metody pro jeden ze základních datových typů - string.

7.2.1.11 Tasks

Obsahuje struktury zamýšlené k práci s vlákny. Task v jazyce C je jednou z hlavních tříd, se kterou se pracuje na vícevláknové úrovni. V aktuálním stavu zde máme připravenou logiku pro tzv. pracovníka (worker), který dokáže obsloužit práci (task), pokud mu ji přidělíme a následně nás o konečném stavu informovat.

7.2.1.12 Web

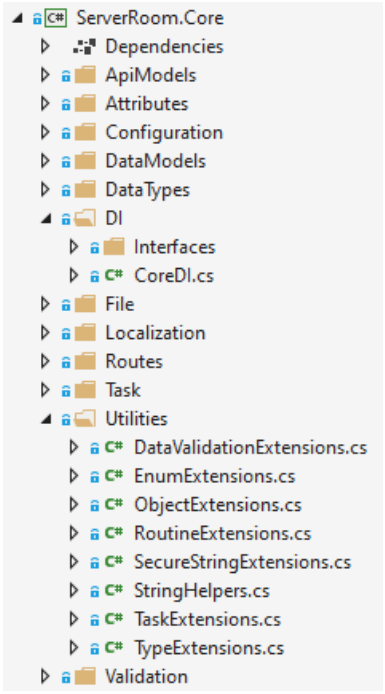
V naší aplikaci je klíčová komunikace mezi serverem a klientem. Tento adresář je velmi důležitou částí k této komunikaci. Nabízí velmi jednoduché rozhraní pro odesílání a přijímání HTTP zpráv bez nutnosti řešit procesy, jako například převod dat do přenositelné formy atp. Pouze definujeme v jakém formátu chceme dat přenášet a máme na výběr JSON nebo XML.

Více informací o implementaci komunikaci klienta se serverem naleznete v sekci 7.8, kde bude popsán podrobně celý průběh komunikace na příkladu.

7.2.2 Projekt: ServerRoom.Core

Projekt ServerRoom.Core je naším hlavním projektem pro klientskou aplikaci. Obsahuje funkcionalitu využitou ve všech částech našeho systému. Pokud je něco natolik obecné, že to není specifické pro server ani pro klienta, může to sloužit všem a zároveň je to logika spjatá pouze s tímto naším systémem, tak právě takový kód patří do tohoto projektu. Díky vybudované struktuře 7.1 výše položené projekty nedokáží přistupovat k níže položeným projektům jako je tento. Projekt je založený tak, aby si vystačil obsahem sám bez pomoci prezentační vrstvy nebo vrstvy business logiky. Popis obsahu celého projektu viz. obrázek 7.7.

Projekt je implementován pomocí .NET Standardu. Jako náš klíčový projekt je potřeba udržet aplikaci dostupnou pro všechna možná budoucí rozšíření. Z toho důvodu .NET Standard tvoří vhodnou možnost z výběru pro naše řešení.



Obrázek 7.7: Struktura projektu ServerRoom.Core

Podrobné rozebrání všech částí projektů je v následujících podsekcích. Každá podsekce prezentuje jednu kořenovou složku dle obrázku 7.7.

7.2.2.1 Dependencies

Existence této části má stejný význam jako pro projekt Ixs.DNA - viz popsáno v sekci 7.2.1.1.

7.2.2.2 ApiModels

Zde se nachází všechny aplikací použité data modely, které slouží k přenosu dat pomocí API. Modely jsou rozdělené do několika kategorií dle jejich funkce - například žádost o data nebo odeslání dat.

API modely se celkem dělí na 5 skupin:

- **Add** - API modely takto označené slouží jako požadavek na server pro přidání nového prvku do databáze pomocí dat přiložených v tomto API data modelu.
- **Get** - Slouží jako požadavek k vrácení dat s požadovanými parametry, které nesou tento API data model.

- **Put** - Odeslání dat na server za účelem akce, jako je například přihlášení uživatele nebo registrace.

- **Result** - Model takto označený směřuje pouze ze serveru pryč. Tento data model přenáší data, která jsou vyžádána.

- **Update** - Podobné jako modely označené pomocí **Add** s tím rozdílem, že účelem je data v databázi aktualizovat podle přiložených dat v data modelu, nikoli je přidat.

Podrobný popis implementace komunikace přes API je podrobně popsáno v sekci 7.8.

7.2.2.3 Attributes

Jedná se o definici všech vlastních atributů pro třídy, metody, properties[22] nebo fields[20]. Pomocí atributů dokážeme specifikovat přídavnou funkcionalitu daného prvku. V našem případě je primárně využíváme k validaci dat. Blíže se k validaci dostaneme v kapitole 7.8.2.

7.2.2.4 Configuration

Složka obsahuje seznam konstant, které specifikují funkčnost celého systému.

7.2.2.5 DataModels

Hlavní část celého projektu tvoří právě data modely. Zde jsou nadefinovány data modely a také prakticky, jak budou vypadat databázové tabulky jednotlivých modelů.

Data modely mají vnitřní rozdělení, kde přímo v dané složce se nachází základní data modely. Zde se nachází i složka `Connectors` a `Routines`. V případě složky `Connectors` se zde nachází data modely spjaté se strukturami, které nazýváme DataConnectors (7.6.1). Pro `Routines` platí podobné pravidlo s rozdílem, že rutina je rozdělena vždy na 3 části: Základ rutiny, stav a konfigurace (více v 7.6). Každá rutina je tedy složena ze tří data modelů. Diskuse o návrhu již problematiku probírala v sekci 6.4.1.1.

7.2.2.6 DataTypes

Složka obsahuje definice všech vlastních datových typů používaných v tomto celém systému.

7.2.2.7 DI

Jak již název napovídá, nacházejí se zde specifikace týkající se DI. Zde se vyskytuje hlavní DI třída, která specifikuje zkratky pro data vložené do DI na dané vrstvě (více informací již bylo řečeno v sekci 7.2.1.7). Dále zde můžeme najít rozhraní struktur, na základě kterých můžeme vytvářet příslušná data.

7.2.2.8 File

Ve složce specifikujeme obecného souborového správce, který dokáže číst a zapisovat do souboru.

7.2.2.9 Localization

Tato složka se nachází v každém projektu, ve kterém je potřeba na dané úrovni vytvářet textový obsah pro uživatele. Tento obsah je zapisován do tzv. resource (.resx) souborů, které jsou podporovány přímo .NET Frameworkem a přináší vývojáři jedinečnou možnost skladovat textové řetězce pod automaticky spravovanými identifikátory. My využíváme právě těchto souborů, abychom měli vždy otevřenou možnost uvést aplikaci vícejazyčně.

7.2.2.10 Routes

Routes je klíčové místo pro shromažďování všech uživatelsky dostupných cest na serveru. Kamkoli se může uživatel nebo API dostat pomocí GET požadavku HTTP, tak je zde definováno jako konstanta. Pomocí takového seznamu můžeme snadno opravit, ale hlavně používat všechny cesty bez nutnosti vypisovat jeden a ten samý řetězec stále dokola na nespočet míst.

Je to velmi důležité pro server, aby věděl jaká data zobrazovat pro jakou adresu, a zároveň na straně klienta je to velmi důležité ze strany API. Je potřeba vědět, kam odesílat požadavky pro API a právě zde máme seznam všech dostupných cest.

7.2.2.11 Task

Adresář Task specifikuje obecného Task manažera, který implementuje výchozí metody Task. S tím rozdílem, že přidává logování pro snadnější debugování při vývoji.

7.2.2.12 Utilities

Adresář Utilities obsahuje směs mnoha extension metod rozdělených do tříd podle datového typu, pro který jsou navržené.

7.2.2.13 Validation

V této složce definujeme třídy poskytující validaci. Nachází se zde primárně obecný validátor, který slouží k validaci celého zařízení (device) a následně pomocné třídy, jako je například výsledek testování, který v sobě dokáže nést data typu - list chyb nebo stav validace.

Více o validaci v kapitole 7.8.2, kde ji rozebereme podrobněji.

7.3 KLIENT

V této kapitole bude popsána struktura klientské aplikace, mimo jiné: architektura projektů, struktura jednotlivých částí a jejich nejvýznamnější třídy.

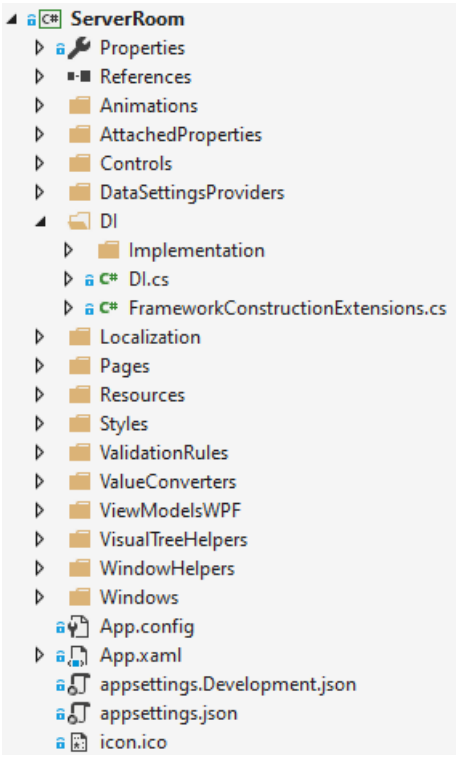
Klientská aplikace využívá celkem 3 projekty: `ServerRoom`, `Server.Room.Client.Base` a `ServerRoom.Relational`. Projekt `ServerRoom.Relational` stále ještě spadá do datové vrstvy, ale zbylé 2 projekty tvoří již nižší vrstvu. Klient využívá architektury MVVM[55] a celá implementace je rozložena do obou dvou projektů. Z praktického i teoretického hlediska může být implementace pouze v jednom projektu. V našem řešení jsme se však rozhodli tomuto zabránit z jednoho prostého důvodu. Hlavní projekt `ServerRoom` je implementovaný pomocí .NET Frameworku, který je podporován pouze operačním systémem Windows. .NET Framework proto implementuje knihovny fungující pouze na operačním systému Windows. Především se jedná o knihovny poskytující grafické prvky nebo práci s nimi. Rozhodně se chceme vyhnout používání těchto knihoven v naší business logice, a proto veškerou business logiku odsuneme do projektu `Server.Room.Client.Base` a v základním projektu zůstane pouze to, co se přímo týká grafiky nebo uživatelského rozhraní.

7.3.1 Projekt: ServerRoom

Projekt `ServerRoom` je koncovým projektem pro naší klientskou aplikaci. Obsahuje logiku vztaženou pouze k uživatelskému rozhraní a nutným prvkům, které s uživatelem interagují.

Popis obsahu celého projektu viz. obrázek 7.8.

Projekt je implementován pomocí .NET Frameworku, tudíž je podporovaný pouze systémem Windows. Pro nás to do budoucna netvoří žádnou překážku. Jedná se o hlavní projekt, ze kterého dále můžeme odvíjet další a díky chytré architektuře 7.1 můžeme kdykoli vytvořit dalšího klienta pro jinou platformu.



Obrázek 7.8: Struktura projektu ServerRoom

Podrobný rozbor všech částí projektů je v následujících podsekcích. Každá podsekce prezentuje jednu kořenovou složku dle obrázku 7.8.

7.3.1.1 Properties

Obsahuje soubory pro speciální funkčnost pro .NET Framework. Nachází se zde nastavení projektu, jako je například: kdy a co se má vytvářet při kompilaci projektu nebo jaké soubory se mají generovat pro nasazení aplikace.

7.3.1.2 References

Funguje obdobně jako je popsána sekce Dependencies (7.2.1.1) u projektů fungujících na .NET Standard.

7.3.1.3 Animations

Definice všech animací se nalezne na tomto místě. Jedná se o animace, které nám umožňují například plynule přecházet z jedné stránky aplikace na jinou. I když se to na první pohled nezdá, implementovat animace do WPF, není jednoduchá akce. Alespoň ne za podmínky, že chceme strukturu udržet přehlednou a snadno rozšířitelnou.

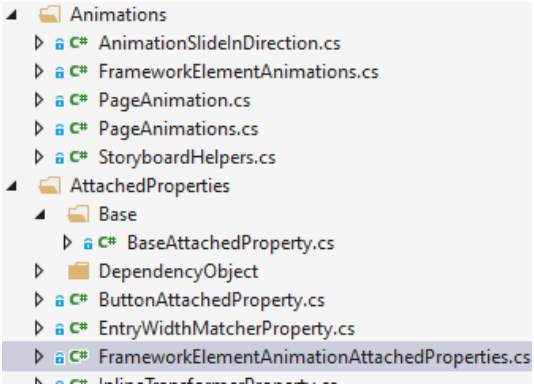
Na ukázku se zde nachází metoda (Obr 7.9), která dokáže přidat animace objektu přijíždějícího z levé strany v přechodu ze ztracena. Toto je přímo samotná metoda, která obstarává logiku animace, ale zbylé metody na obrázku 7.10 se starají o logiku, jak animace lépe distribuovat pro jejich implementaci. Vidíme zde také označenou část v AttachedProperties, která je koncovou částí pro aplikaci animací přímo ve WPF šablonách. Více o AttachedProperties v sekci 7.3.1.4.

```
public static void AddSlideInFromLeft(this Storyboard storyboard, float seconds, double offset, float decelerationRatio = 0.9f, bool keepMargin = true)
{
    // Create the margin animate from right
    var animation = new ThicknessAnimation
    {
        Duration = new Duration(TimeSpan.FromSeconds(seconds)),
        From = new Thickness(-offset, 0, keepMargin ? offset : 0, 0),
        To = new Thickness(0),
        DecelerationRatio = decelerationRatio
    };

    // Set the target property name
    Storyboard.SetTargetProperty(animation, new PropertyPath("Margin"));

    // Add this to the storyboard
    storyboard.Children.Add(animation);
}
```

Obrázek 7.9: Ukázka animační metody



Obrázek 7.10: Animační a Attached Properties struktura

7.3.1.4 AttachedProperties

Attached Properties [87] v našem projektu využíváme velmi hojně. Jsou velmi užitečné, pokud potřebujete implementovat funkčnost uživatelského rozhraní, která nativně není podporována.

Jako příklad attached property můžeme na obrázku 7.11 vidět aplikace jedné takové attached property. Jedná se o aplikaci animace, které předáváme boolean hodnotu z kódu o stavu editace - jestli uživatel v danou chvíli edituje formulář nebo ne. Dále můžeme vidět tzv. „Cenverter". Více o konvertorech v sekci 7.3.1.13.

Attached properties mohou být netriviální pro nové uživatele ve WPF. V důsledku toho je zde implementovaná výchozí třída `BaseAttachedProperty` (můžeme vidět na obr. 7.10), která implementuje výchozí logiku pro attached properties. Díky tomu jsou naše attached properties velmi jednoduché pro implementaci a jakoukoli budoucí změnu. Pokud se attached properties implementují všechny bez hlubší myšlenky k použití, nastává zde problém s nehezkým opakováním v kódu.

```
<Grid
    Grid.Column="2"
    HorizontalAlignment="Right"
    VerticalAlignment="Center"
    local:AnimateFadeInNoWaitProperty.Value="{Binding Editing, Converter={local:BooleanInvertConverter}}">
```

Obrázek 7.11: Attached Property, ukázka aplikace

7.3.1.5 Controls

Obsahuje části definic uživatelského rozhraní. UserControls [89] máme použité například na uživatelské vstupy nebo na rozčlenění stránek do sekcí. Obecně jsou použité na prvky uživatelského rozhraní, které se v kódu mohou opakovat nebo zjednoduší přehled kódu uživatelského rozhraní.

7.3.1.6 DataSettingsProviders

Vlastní psaná logika pro rozšíření funkcionality ukládání dat lokálně do souboru. Rozšiřujeme logiku přímo .NET Frameworku. Nicméně se jedná o logiku převzatou z jiných projektů a v momentální fázi aplikace není využita.

7.3.1.7 DI

Jak již název napovídá, nacházejí se zde specifikace týkající se DI. Zde se vyskytuje hlavní DI třída, která specifikuje zkratky pro data vložená do DI na dané vrstvě (více informací již bylo řečeno v sekci 7.2.1.7). Dále, co se zde nachází, jsou implementace rozhraní struktur, které jsou implementovány na základě rozhraní z vyšších projektových vrstev.

7.3.1.8 Localization

Stejně definována jako v sekci 7.2.2.9.

7.3.1.9 Pages

Obsahuje definic stránek aplikace. Zde jsou nadefinované šablony celé aplikace.

7.3.1.10 Resources

Zde naleznete externí média potřebná pro chod aplikace. V našem případě zde máme obrázky (loga) a fonty použité při konstrukci klientské aplikace.

7.3.1.11 Styles

V této složce definujeme rozšiřující XAML šablony pro naše uživatelské rozhraní. Tyto rozšiřující šablony definují opakující se styly v šablonách a převádí je na jedno místo.

7.3.1.12 ValidationRules

V této složce definujeme obálky okolo funkčnosti atributů pro validování uživatelských vstupů na straně klienta [90]. Více o samotné validaci v kapitole 7.8.2.

7.3.1.13 ValueConverters

Value Converters [91] slouží pro konverzi dat v šablonách. Například, pokud máme datovou boolean hodnotu, kterou potřebujeme pouze pro zobrazení uživatelského rozhraní, tak použijeme converter. Dynamicky pak můžeme transformovat data pouze pro šablony.

Je zde implementováno nemalé množství obecných konvertorů pro různé datové typy pro široké použití.

7.3.1.14 ViewModelWPF

V této složce nalezneme view modely [55] specifické pouze pro WPF. V našem případě zde máme například `WindowViewModel`, který definuje parametry, jak by mělo vypadat běžné okno klientské aplikace.

Velmi významný je také pomocník pro view modely `ViewModelLocator`. V tomto scriptu definujeme zkratky z celého klientského projektu, kterými lze přistoupit k datům v šablonách uživatelského rozhraní.

7.3.1.15 VisualTreeHelpers

Speciální pomocné třídy, které na straně jazyka C# dokáží číst XAML, kterým jsou definovány šablony ve WPF. Pomocí těchto pomocných tříd dokážeme na straně kódu procházet šablony.

7.3.1.16 WindowHelpers

Zde se nachází pouze jedna neměnná třída `WindowResizer`. Tato metoda je implementována podle .NET standardů a pomáhá aplikaci řešit změnu měřítka na operačním systému Windows.

7.3.1.17 Windows

Adresář definuje všechna okna aplikace. Jsou zde nadefinována i okna pro stahování aktualizací, ale ty jsou zatím nefunkční a jejich implementace se bude řešit v budoucím vývoji.

Hlavní okno aplikace je pouze jen jedno a to se používá pro všechen obsah klientské aplikace.

7.3.1.18 App.xaml / App.xaml.cs

Jedná se o klíčový skript, který je hlavním vstupem celé klientské aplikace. Zde máme nadefinováno, co se stane při startu a před ukončením aplikace. Máme zde připravené skripty pro aktualizaci naší klientské aplikace, kterou plánujeme implementovat v budoucnu.

7.3.1.19 appsettings.json

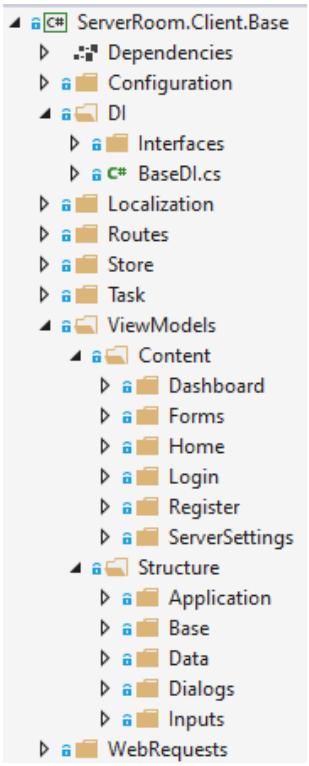
Definice důležitých informací, které mohou být nastaveny uživatelem pro modifikaci klientské aplikace.

Specifikujeme zde úroveň výpisu logování, výchozí připojení serveru a uložení lokální databáze.

7.3.2 Projekt: ServerRoom.Client.Base

Projekt `ServerRoom.Client.Base` je nedílnou součástí koncového projektu `Server.Room`. Obsahuje business logiku pro uživatelské rozhraní na straně klienta. Popis obsahu celého projektu viz. obrázek 7.12.

Projekt je implementován pomocí .NET Standardu. U tohoto projektu není důležité, jestli je implementace v .NET Standardu nebo .NET Core. Hlavním důvodem je rozdělení business logiky a uživatelského rozhraní do dvou projektů a tím zamezit používání grafických knihoven v business logice (view modelech).



Obrázek 7.12: Struktura projektu ServerRoom.Client.Base

Podrobné rozebrání všech částí projektů je v následujících podsekcích. Každá podsekce prezentuje jednu kořenovou složku dle obrázku 7.12.

7.3.2.1 Dependencies

Existence této části má stejný význam jako pro projekt Ixs.DNA - viz popsáno v sekci 7.2.1.1.

7.3.2.2 Configuration

Obsahuje seznam konstant, které specifikují klíčové hodnoty pro funkčnost klientské aplikace.

7.3.2.3 DI

Jak již název napovídá, nacházejí se zde specifikace týkající se DI. Zde se vyskytuje hlavní DI třída, která specifikuje zkratky pro data vložená do DI na dané vrstvě (více informací již bylo řečeno v sekci 7.2.1.7). Dále, co se zde nachází, jsou rozhraní struktur, na základě kterých můžeme vytvářet příslušná data.

7.3.2.4 Localization

Stejně definována jako v sekci 7.2.2.9.

7.3.2.5 Routes

Obsahuje třídu `RouteHelpers`, která je pomocnou třídou pro sestavování URL adres pro odesílání požadavků na server.

7.3.2.6 Store

Zde se nachází logika pro práci s lokálním uložištěm dat. Zde je místo pro definování logiky ukládání nebo načítání konkrétních dat pomocí speciálních view modelů.

Jako příklad zde máme třídu `AppStateDataWrapper`, která je view modelem pro data udržující informace o posledním stavu klientské aplikace (například rozměr otevřeného okna aplikace při posledním zavření).

7.3.2.7 Task

Zde ukládáme hlavní logiku týkající se servisních „tasků". Zde také implementujeme naší hlavní část aplikace, která udržuje ve spojení klientskou aplikaci se serverem. Více o aktualizaci dat na klientské aplikaci v sekci 7.3.4.

7.3.2.8 ViewModels

View Modely jsou naším jádrem business logiky pro klientskou aplikaci. View modely rozdělujeme do dvou sekcí:

- **Content** - Jedná se o view modely, které obsluhují aktivní část aplikace, tj. stránky a jejich součásti (UserControls[89]).
- **Structure** - Tyto view modely obsluhují pasivní část aplikace. Část, kterou uživatel nevidí jako hlavní prvek. Jsou zde view modely obsluhující notifikace, uživatelské vstupy, základní data klientské aplikace a v neposlední řadě základové view modely.

Základové view modely (Base) jsou rodičem všech ostatních view modelů. Je zde několik typů, ze kterých lze view modely dědit s tím, že zde je jeden hlavní `BaseViewModel`, ze kterého dědí i ostatní základové modely.

BaseViewModel

Základem každého view modelu je logika, která umožní snadno aktualizovat data v našich šablonách. Pokud změníme data do našich šablon bez této logiky, tak tato data nebudou v šablonách aktualizovaná v reálném čase. Z toho důvodu implementujeme rozhraní `INotifyPropertyChanged`, které přidá do takové instance akce, které se automaticky spustí při změně konkrétní property naší třídy. Tato akce informuje šablonu, že šablona je potřeba aktualizovat.

Dále se zde nachází pomocné metody, které se s velkou pravděpodobností mohou vyskytovat v potomcích tohoto základu. Klíčovou metodou je zde `RunCommandAsync`, která implementuje spuštění asynchroního „tasku"chráněného identifikací spuštění. To znamená, že pokud task je spuštěn, tak ve stejný okamžik nemůže být spuštěn paralelně ten samý. Tato metoda je vhodná pro použití pro uživatelské vstupy (např. tlačítka). Tato metoda je představena na obrázku 7.13.

```
protected async Task RunCommandAsync(Expression<Func<bool>> updatingFlag, Func<Task> action)
{
    // Lock to ensure single access to check
    lock (updatingFlag)
    {
        // Check if the flag property is true (meaning the function is already running).
        if (updatingFlag.GetPropertyValue())
            return;

        // Set the property flag to true to indicate we are running.
        updatingFlag.SetPropertyValue(true);
    }

    try
    {
        // Run the passed in action.
        await action();
    }
    finally
    {
        // Set the property flag back to false now it's finished.
        updatingFlag.SetPropertyValue(false);
    }
}
```

Obrázek 7.13: Metoda RunCommandAsync ve třídě BaseViewModel

7.3.2.9 WebRequests

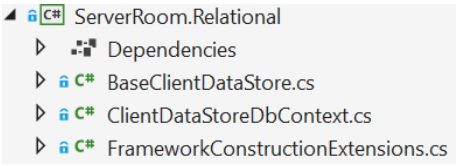
Rozšířená logika pro systém odesílání HTTP požadavků z našeho projektu Ixs.DNA. Konkrétní logika, která je specifická právě pro klientskou aplikaci na základě tohoto systému.

7.3.3 Projekt: ServerRoom.Relational

Projekt `Server.Room.Relational` je velmi malou částí celého našeho systému. Projekt definuje připojení do databáze a databázi samotnou pomocí řešení SQLite. Databáze má sloužit na straně klienta, kde bude ukládat data vztažená ke stavu aplikace (např. velikost okna, ve

kterém se má aplikace znovu otevřít při dalším spuštění). Popis obsahu celého projektu viz. obrázek 7.14.

Projekt je implementován pomocí .NET Standardu. Pomocí tohoto řešení vždy zůstaneme otevřené možnosti řešení pro více platforem.



Obrázek 7.14: Struktura projektu ServerRoom.Relational

Podrobný rozbor všech částí projektů je v následujících podsekcích. Každá podsekce prezentuje jednu kořenovou složku dle obrázku 7.14.

7.3.3.1 Dependencies

Existence této části má stejný význam jako pro projekt Ixs.DNA - viz popsáno v sekci 7.2.1.1.

7.3.3.2 BaseClientDataStore.cs

`BaseClientDataStore.cs` skript je základním správcem, který obsahuje logiku pro ovládání dat v databázi (ukládání, načítání nebo dotazování, zda existují). Tento správce je připravený pro to, aby se implementoval na dané klientské aplikace. Je možné, že zrovna tento skript v budoucnu bude přemístěn do jiného projektu.

7.3.3.3 ClientDataStoreDbContext.cs

Tak jako třeba na úrovni serveru, tak i zde je implementace databáze zajištěna pomocí EntityFramework[30]. Základem je právě tato definice databázového kontextu. Tento skript tvoří definici naší databázové struktury.

7.3.3.4 FrameworkConstructionExtensions.cs

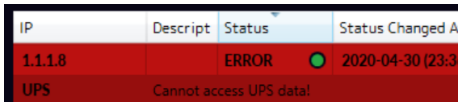
Construction extension skripty obecně v každé části projektu rozšiřují funkcionalitu o navázání konkrétní logiky do DI. Zde tomu není výjimkou a tento skript dodává možnost přidat databázového správce do části našeho programu.

7.3.4 Update Worker

V této sekci si přiblížíme, jak funguje aktualizace dat ze serveru do klientské aplikace. Ukážeme si některé klíčové části a pokusíme se vysvětlit základní logiku.

Základem je `DataViewModel`, který obsahuje informace o všech datech, která je potřeba v pravidelných cyklech aktualizovat pro klientskou aplikaci. Z toho důvodu tento data model implementuje dvě metody: `StartUpdateWorkerAsync` (pro vytvoření a start workera) a `StopUpdateWorkerAsync` (pro zastavení a zrušení workera) a také si drží referenci na pracovníka (workera), který vykonává aktualizovací rutinu. Tento data model je zamýšlen tak, aby se v celé aplikaci vyskytovala pouze jedna instance tohoto data modelu, a proto jej můžeme implementovat do DI, ze které následně snadno k datům přistupujeme.

`DataViewModel` následně implementuje i metody pro samotné aktualizace dat. V momentální verzi aplikace zde přenášíme 2 struktury: Seznam všech zařízení včetně jejich rutin a uživatelský data log. Z tohoto důvodu zde máme dvě metody pro aktualizaci těchto dat: `UpdateDeviceListAsync` a `UpdateDataLogAsync`.



Obrázek 7.15: Identifikátor chybného statusu při desynchronizaci

Při spuštění workera přes metodu `StartUpdateWorkerAsync` se vytvoří worker podle třídy `DataUpdateWorker`. Tento worker je implementován pomocí třídy `SingleTaskWorker` z projektu Ixs.DNA, kde je pro nás klíčová metoda `WorkerTaskAsync`. Metoda `WorkerTaskAsync` je implementována z třídy `SingleTaskWorker` a pro nás znamená samotný jeden průběh aktualizace. Proto zde vytvoříme nekonečnou smyčku, která může být přerušena jen a pouze na žádost zastavení workera. Následně tato metoda bude v cyklu s intervalem spouštět metodu `UpdateAsync` (obr. 7.16), kde je definovaná samotná aktualizovací rutina.

```
/// <summary>
/// Update routine
/// Automatically stops the worker if user is not logged in
/// </summary>
/// <param name="cancellationToken">Cancelation token of the worker</param>
/// <returns>Bool of state if the update passed</returns>
1 reference
protected async Task<bool> UpdateAsync(CancellationToken cancellationToken)
{
    // Store single transcient instance of client data store
    var scopedClientDataStore = CoreDI.ClientDataStore;

    // Get the user token
    var token = (await scopedClientDataStore.GetLoginCredentialsAsync())?.Token;

    // If we don't have a token (so we are not logged in...)
    if (string.IsNullOrEmpty(token))
    {
        // Stop update worker - we are not logged in
        _ = mDataVM.StopUpdateWorkerAsync();
        // Then do nothing more
        return false;
    }

    // Log it
    FrameworkDI.Logger.LogDebugSource("Updating data...");

    // Update data (device list)
    if (!await UpdateDataAsync(token))
        return false;

    // Update data log
    if (!await UpdateDataLogAsync(token))
        return false;

    // Log it
    FrameworkDI.Logger.LogDebugSource("Data successfully updated!");

    // Successfully updated
    return true;
}
```

Obrázek 7.16: Hlavní metoda aktualizace dat ve třídě DataUpdateWorker

Bylo myšleno i na ztrátu synchronizace dat, která může nastat. Řekněme, že máme aktualizovací cyklus nastavený na každých 60 sekund a je zde testovací rutina, která probíhá každých 28 sekund. Tato testovací rutina teoreticky může proběhnout třikrát za jeden aktualizovací cyklus klientské aplikace (zmiňuji, že testování probíhá na serveru). Pokud by nastala chyba během prvního nebo druhého testu a následně další test by byl v pořádku, tak uživatel nedokáže rozpoznat a ani nebude upozorněn, že chyba nastala. V takovou dobu tuto informaci nese pouze datový log. Z tohoto důvodu má data model `DeviceDataModel` implementovanou pomocnou property `StatusMasqueraded`, která slouží metodě `CheckForDesyncedStatus` v třídě `DataViewModel` pro nastavení statusu rutiny, která byla označena jako chybná. Následně lze v klientské aplikaci dle těchto informací určit, zda od poslední aktualizace došlo k chybnému stavu nebo ne. Pokud k takové situaci dojde, systém zobrazí uživateli vždy tu nejhorší variantu, která od poslední aktualizace nastala. Zároveň se zobrazí identifikátor, který zobrazuje aktuální správný stav, aby uživatel věděl, že nastala chyba, ale v daném čase je vše již v pořádku. Zobrazení takového identifikátoru je ukázáno na obrázku 7.15.

7.4 SERVER

V této kapitole bude popsána struktura serveru, jako například: architektura projektů, struktura jednotlivých částí a jejich nejvýznamnější třídy.

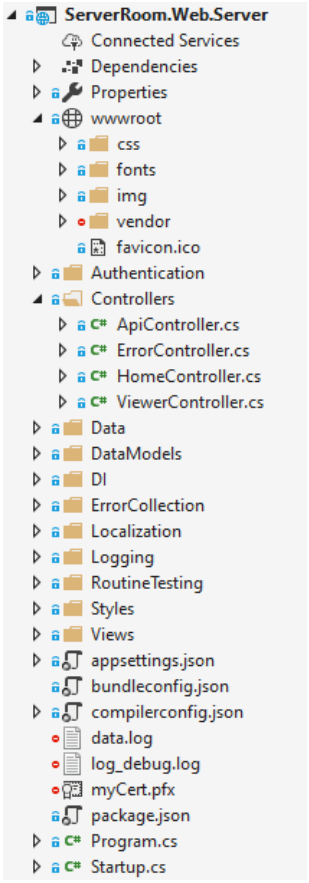
Server je založen pouze na jednom hlavním projektu na rozdíl od klientské aplikace, která je složena ze tří projektů. Server je postavený na architektuře MVC[53], kde veškeré řízení funkčnosti v této struktuře velmi dobře spravuje ASP.NET framework.

7.4.1 Projekt: ServerRoom.Web.Server

Projekt `ServerRoom.Web.Server` je koncovým projektem a tím i startovacím projektem pro náš server. Popis obsahu celého projektu viz. obrázek 7.8.

Projekt je implementován pomocí ASP.NET Core (MVC), tudíž je podporovaný pro více možných platforem.

Podrobný rozbor všech částí projektů je v následujících podsekcích. Každá podsekce prezentuje jednu kořenovou složku dle obrázku 7.17.



Obrázek 7.17: Struktura projektu ServerRoom.Web.Server

7.4.1.1 Dependencies

Existence této části má stejný význam jako pro projekt Ixs.DNA - viz popsáno v sekci 7.2.1.1.

7.4.1.2 Properties

Nachází se zde nastavení projektu ve formě souboru `launchSettings.json`. Toto nastavení je i jednou z věcí, která je potřeba upravit při nasazování aplikace ve finálním prostředí.

7.4.1.3 wwwroot

Adresář `wwwroot` specifikuje naši kořenovou strukturu pro finální webovou aplikaci. Nalezneme zde soubory CSS, zdrojové fonty, obrázky použité na naší web. stránce, ikonu stránky a také složku vendor, která obsahuje styly a skripty třetí strany použité na naší webové stránce.

Veškerý obsah tohoto adresáře je složen pouze z produkční verze těchto skriptů, tzn. soubory skriptů a stylů jsou minimalistické (označení `min`). Do této složky obsah pouze generujeme, netvoříme ho sami. Naše styly (CSS) jsou generovány pomocí SCSS[72]. Generovací pravidla jsou definována v souboru `compilerconfig.json`, přiloženém v kořenovém adresáři. Dále zde máme externí knihovny (vendor), které je potřeba nejdříve specifikovat. V našem řešení používáme automatické řešení pomocí NPM[64] a jejich definici uvádíme v souboru `package.json`.

Externí knihovny (vendor) jsou importovány, ale je potřeba jejich začlenění do projektu. Musíme tedy specifikovat, které soubory potřebujeme pro naší aplikace a vložit je do cílového adresáře odkud je aplikace zvládne načítat. Tuto specifikaci definujeme v souboru `bundleconfig.json`.

Na závěr potřebujeme vložit všechna tato data do webové stránky. Pro takové řešení lze snadno vložit příslušné odkazy do hlavičky HTML, naší kořenové šablony `Views/Shared/_Layout.cshtml` (více o šablonách v sekci 7.4.1.14).

7.4.1.4 Authentication

Zde se nachází definice autentizace a autorizace uživatele. Jsou zde dvě hlavní třídy `AuthorizeTokenAttrib`
(vytváří atribut pro třídy a metody specifikující konkrétní autentizaci pomocí JWT[47]) a `JwtTokenExtensionMethods` (přidává schopnost generovat JWT token dle daných hodnot). Více o JWT tokenech v sekci 5.5.

7.4.1.5 Controllers

„Controllery"jsou klíčovou částí naší webové aplikace. Tvoří hlavní business logiku. Nachází se zde celkem 4 „controllery":

- **API** - Zajišťuje veškerou logiku pro API, které náš webový server poskytuje.
- **Error** - Zajišťuje zobrazení chyb, pokud na serveru dojde k jakékoli chybě, kterou potřebujeme předat uživateli.
- **Home** - Vytváří business logiku pro obrazovky uživatelského zobrazení přímo na webu a jeho základní funkce (například přihlášení).
- **Viewer** - Zajišťuje logiku stejně jako `HomeController`, ale pouze pro zobrazování dat (viewers).

Speciálním případem je `ApiController`, který implementuje námi vytvořený atribut `AuthorizeToken` a tím přidává nutné oprávnění pro použití tohoto „controlleru". Zároveň tento controller vyžaduje oprávnění ve výchozím stavu a pouze označené metody mají právo být použity bez oprávnění. Takové řešení je bezpečnější z pohledu programátora a při přidávání nové metody jsme vždy zabezpečeni.

7.4.1.6 Data

Jedná se o velmi významný adresář, který definuje strukturu naší databáze. Nachází se zde databázový kontext `ApplicationDbContext`, který je „základním kamenem"naší databáze. Podobně jako v sekci 7.3.3, tak i zde definujeme databázovou strukturu pomocí EntityFramework[30].

7.4.1.7 DataModels

Adresář reprezentuje datovou vrstvu uprostřed modelu MVC. Podobně jako v projektu `ServerRoom.Core`, tak i zde máme datové modely, ale pouze potřebné na straně serveru.

Příkladem je zde datový model `ApplicationUser`, který rozšiřuje databázovou tabulku uživatelů o vlastní nová pole, která výchozí řešení .NET nenabízí.

7.4.1.8 DI

Jak již název napovídá, nachází se zde specifikace týkající se DI. Zde se vyskytuje hlavní DI třída, která specifikuje zkratky pro data vložená do DI na dané vrstvě (více informací již bylo řečeno v sekci 7.2.1.7).

7.4.1.9 ErrorCollection

Jedná se o extension metody pro konkrétní vytvořenou strukturu v našem projektu. Díky těmto rozšířením můžeme pracovat s polem `IEnumerable<IdentityError>`. Toto pole ukládá chybné hodnoty o správě uživatelů (přidávání/odebírání/aktualizace). Pomocí snadných metod můžeme připravit tato data pro výstup a přehledně je zobrazit uživateli.

7.4.1.10 Localization

Stejně definována jako v sekci 7.2.2.9.

7.4.1.11 Logging

Podobně jako v sekci 7.2.1.8, tak i zde definujeme logování. Tentokrát to ovšem není klasické logování pro programátora, ale pro uživatele. Je zde vytvořená struktura datového logovacího systému, který je postavený na stejném základě jako v sekci 7.2.1.8.

Hlavní problém, který nastává při logování je velikost a kam data ukládat. Máme zde implementované logování do souboru, které s sebou nese i další problémy. Je dobré zmínit, že klientské aplikace aktualizují data v cyklech a není nejrozumnější vždy každý cyklus posílat všechny zprávy logu (při objemnějších logových souborech by to mohlo mít vliv na rychlost přenosu dat). Z toho důvodu přenášíme vždy pouze jen ty nové zprávy. Abychom tohoto docílili, tak s ostatními daty přenášíme místo v souboru, odkud byl naposledy aktualizován log (velikost souboru). Při následném požadavku o log vždy žádáme pomocí počtu požadovaných řádků logu, velikostí souboru, ve kterém byl log naposledy aktualizován a času poslední aktualizace pro ochranu proti ztrátě synchronizace.

7.4.1.12 RoutineTesting

Adresář obsahuje struktury pro testování rutin zařízení. Nachází se zde klíčové skripty jako `BackgroundDeviceService`, `BaseDeviceWorkerManager` nebo `DeviceWorker`. O testování rutin bude více řečeno v sekci 7.4.3. Na obrázku 7.18 se můžeme všimnout celé adresářové struktury, kde se nachází i podadresář `Routines`. Podadresář `Routines` obsahuje všechny existující metody testů, které jsou v našem systému dostupné. Na zmíněném obrázku jsou vidět dvě rutiny:

- **Ping (PingRoutineTest.cs)** - Dostupná rutina v našem systému, která testuje zařízení pomocí jednoduchého příkazu PING[68]. Naše implementace metody PING nabízí několik parametrů, kterými lze PING přizpůsobit uživatelským požadavkům. Tyto parametry jsou:
  - **Multi-Check** - Specifikuje záložní PING požadavky, pokud první selže. Vždy je definováno testování pomocí jednoho požadavku PING při nastavené hodnotě na nulu. Při hodnotě vyšší než 0, pokud by první PING požadavek selhal, se přistoupí na další PING a stejným principem pokračuje, dokud nevyčerpá zásobu PING požadavků na jeden test.
  - **Multi-Check Delay** - Určuje časovou mezeru mezi koncem předchozího PING požadavku a tím následujícím (platí pouze, pokud je nutné testovat více PING

požadavků v jednom testu - Multi-Check).

  - **Timeout** - časový limit pro jeden PING požadavek.

- **UPS (UpsRoutineTest.cs)** - Tato testovací rutina poskytuje testování pro UPS zařízení. Metoda dokáže testovat, jestli se UPS zařízení vybíjí nebo nabíjí s indikací změny stavu. Dále uživateli předává informaci o momentálním stavu baterie (na kolik procent je baterie nabitá) a její stav (nabíjí/vybíjí). Pro toto řešení je potřeba externích dat, která UPS zařízení poskytuje. Abychom tato data mohli získat, je potřeba použít jednu z dostupných metod (data connectors 7.6.1). Definice dostupných konektorů pro daný test je definována v data modelu `Core.DataModels.Routines.UpsConfigurationDataModel` pomocí property `AllowedDataConnectorTypes`.

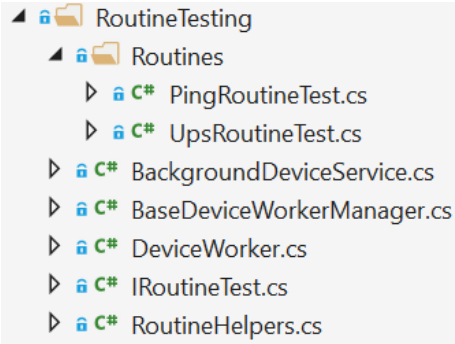
  - **WarningBracket** - Procentuální rozsah (0-100%) definující hranici pro stav označený jako WARNING (oranžový). Pokud procento UPS baterie klesne pod tuto hodnotu, bude výsledný stav testu vyhodnocen jako WARNING.

  - **DangerBracket** - Procentuální rozsah (0-100%) definující hranici pro stav označený jako ERROR (červený). Pokud procento UPS baterie klesne pod tuto hodnotu, bude výsledný stav testu vyhodnocen jako ERROR.

  - **WarningBracketInterval** - Definuje rozpětí v procentech, ve kterém je uživatel znovu upozorněn, i když je upozornění již pod hranicí **WarningBracket**. Jinými slovy definuje procentuální krok pro znovu upozornění pro uživatele, když se UPS nachází stavem baterie v rozmezí **WarningBracket**.

  - **DangerBracketInterval** - Definuje rozpětí v procentech, ve kterém je uživatel znovu upozorněn, i když je upozornění již pod hranicí **DangerBracket**. Jinými slovy definuje procentuální krok pro znovu upozornění pro uživatele, když se UPS nachází stavem baterie v rozmezí **DangerBracket**.

  Nachází se zde i další parametry, které uživatel může nastavit, ale jedná se pouze o experimentální vlastnost, která bude implementována v novější verzi systému (viz. kapitola 9).



Obrázek 7.18: Obsah RoutineTesting adresáře

Samotné testování rutin probíhá ve službě běžící na pozadí webového serveru. Tuto službu spouští a kontroluje třída `BackgroundDeviceService` a vytváří zde instance pro jednotlivé testy zařízení pomocí třídy `DeviceWorker`.

7.4.1.13 Styles

Obsahuje soubor definic stylů pomocí SCSS[72] pro šablony webového serveru. Zde je definovaný kořenový stylový soubor `app.scss`, který je následně kompilován do produkční verze CSS (viz. sekce 7.4.1.3).

Celý adresář je dále strukturován do podadresářů specifikující konkrétní části definic stylů. Můžeme si všimnout, že všechny ostatní stylové soubory mají název začínající s (prefix) podtržítkem. Takové označení programátorovi říká, že takový soubor nemá být kompilován do CSS, ale má být vložen do jiného stylového souboru. Každý adresář obsahuje speciální stylový soubor `_module.scss`, který do sebe vkládá všechny ostatní stylové soubory v daném adresáři. Díky tomuto řešení pak snadno udržíme přehled ve struktuře stylů a následně do kořenového souboru `app.scss` vkládáme už jen pouze tyto „module"soubory z jednotlivých adresářů.

7.4.1.14 Views

Zde se nachází definice všech dostupných šablon (webových stránek). Organizace je velmi jednoduchá, kde každý pod-adresář reprezentuje jeden „controller"(7.4.1.5). V každém podadresáři pak nalezneme šablony odpovídající metodám v daném „controlleru", které definují obsah webové stránky, se kterým uživatel interaguje (metoda má svoji šablonu jen tehdy, kdy má být logika této metody zobrazována).

Nachází se zde i speciální podadresář označený jako `Shared`. Tento podadresář obsahuje sdílený obsah mezi šablonami. Hlavním je `_Layout.cshtml`, který definuje hlavní kostru HTML stránky. Do této šablony je pak následně obsah vkládán do místa pro obsah (body).

7.4.1.15 appsettings.json

Jedná se o klíčový soubor, který je potřeba nastavit při nasazení v cílovém prostředí. Tento soubor definuje základní informace pro server, jako je například databázové připojení a informace pro zabezpečovací token JWT[47].

7.4.1.16 Program.cs

Tento soubor je nejvíce počátečním kódem naší aplikace. Zde vytváříme hlavní konstrukci celého webového serveru a zároveň tak do této části přidáváme náš Ixs.DNA framework (5.15) s nastavením logovacího systému.

Příkladem může být následný plán do budoucího vývoje našeho systému. Budeme chtít umožnit webovému serveru běžet jako pouhá služba Windows na pozadí systému. Pro takové řešení bude potřeba upravit tento soubor. Více o plánovaných změnách v kapitole 9.

7.4.1.17 Startup.cs

Jedná se o výchozí třídu ASP.NET, ve které blíže specifikujeme nastavení celého webového serveru. Třída obsahuje dvě metody: `ConfigureServices` (konfiguruje služby webového serveru) a `Configure` (konfiguruje HTTP požadavky).

Nastavení použití cookies[24] autorizace a automatické přesměrování je definováno v metodě `Configure`. A na druhé straně máme metodu `ConfigureServices`, ve které specifikuje politiku hesel, specifikaci cookies[24] (GDPR), definice všech služeb (včetně naší testovací služby zařízení - 7.4.1.12) a definice autorizace pomocí tokenů JWT[47].

7.4.2 Projekt: ServerRoom.Web.Server.Test

Projekt `ServerRoom.Web.Server.Test` je speciálním projektem, který definuje prostředí pro testování jednotkovými testy.

Projekt používá framework jednotkových testů nazývaný xUnit[92]. Pomocí tohoto frameworku dokážeme vytvořit speciální strukturu jednotkových testů. Další nedílnou součástí

jednotkových testů jsou tzv. Mock objekty [51]. „Mockování"zajišťujeme pomocí volně dostupného frameworku MOQ[52] pro C#.

V současné verzi našeho systému se jedná pouze o počáteční nastavení, kde v budoucí fázi hodláme nasadit jednotkové testy na všechny kritické části kódu. Více o testování v kapitole 8.

7.4.3 Device Worker

Ve své podstatě se jedná o nejdůležitější část celé aplikace. Celá struktura testování rutin se nachází v projektu `ServerRoom.Web.Server` v adresáři `RoutineTesting`.

Hlavní strukturou je zde třída `BackgroundDeviceService`, která implementuje dvě metody: `StartAsync` (spustí službu) a `StopAsync` (zastaví službu). Tato třída implementuje rozhraní `IHostedService`, které obecně definuje službu běžící na pozadí webového serveru. Při spuštění webového serveru se chová jako nekonečně dlouho běžící služba na pozadí. Služba je zamýšlena pouze k jednomu spuštění a to na při spuštění webového serveru a následném vypnutí služby při vypnutí webového serveru. Při startu této služby se vytvoří samostatný databázový kontext, kterým získáme data o všech zařízeních, která mají být spuštěna. Následně data o zařízeních předáme správci, který se stará o pracovníky zařízení (`BaseDeviceWorkerManager`).

`BaseDeviceWorkerManager` je správce uchovávající informaci o všech běžících pracovnících pro testování zařízení. Správce implementuje veřejné ovládací metody z rozhraní `IDeviceWorkerManager` jako je například `AddAndStartWorkerAsync` nebo `RemoveAndStopWorkerAsync`. Správce uchovává list aktivních pracovníků zařízení, kteří jsou testováni. Pokud je některé ze zařízení odebráno z testování, tak je také z tohoto listu smazán jeho pracovník. Při přidávání přes veřejné metody rozhraní je volána privátní metoda `RunAdnStartWorkerAsync`, která zajistí přidání zařízení do testování. Jedna z věcí je vytvoření pracovníků, do kterých jsou vložena jednotlivá zařízení. Metodu pro přidání zařízení k testování můžeme vidět na obrázku 7.19. Tito pracovníci přidávají funkcionalitu pro testování jednotlivých zařízení.

```
private async Task<bool> RunAddAndStartWorkerAsync(DeviceDataModel device)
{
    // CHeck if any previous worker is running uder the same device ID
    if (mDeviceWorkers.ContainsKey(device.Id.ToString()))
    {
        // Log it
        FrameworkDI.Logger.LogDebugSource($"Ignoring. Device worker already exists... (Device: {device.Id})");
        // Worker exists, we cannot add new one to the same ID
        return false;
    }

    // Log it
    FrameworkDI.Logger.LogTraceSource($"Starting new device worker... (Device: {device.Id})");

    // Create a new device worker
    var worker = new DeviceWorker(device);

    // Start worker routine
    // If the start failed...
    if (!await worker.StartAsync())
    {
        // Log it
        FrameworkDI.Logger.LogErrorSource($"Device worker failed to start. (Device: {device.Id})");
        // Worker failed to start, return failure
        return false;
    }

    // Add worker into active worker list
    mDeviceWorkers.Add(device.Id.ToString(), worker);

    // Successfully added and started
    return true;
}
```

Obrázek 7.19: Metoda pro přidání zařízení do testování

Nejnižší vrstvou, která se stará přímo o samotné testování zařízení je `DeviceWorker`. `DeviceWorker` implementuje `SingleTaskWorker` z projektu `Ixs.DNA` (5.15), kde naší klíčovou metodou je `WorkerTaskAsync`. Podobně jako u aktualizačního pracovníka v klientské aplikaci (7.3.4), tak i zde máme definovaný cyklus a následně metodu `ProcessDeviceAsync`, která se stará o konkrétní test zařízení.

Metoda `ProcessDeviceAsync` vždy z databáze načte všechny rutiny k testování pro dané zařízení. Pokud by uživatel během testování odebral všechny rutiny, tak se testování daného zařízení vypne automaticky. Nicméně metoda prochází všechny rutiny zařízení a spouští na těchto rutinách testy na základě jejich typu a parametrů. Jak přesně se rutiny spravují a umožňují spouštět testy je podrobně popsáno v sekci 7.6.

Důležitou informací je také, že všechny testy běží asynchronně jako zbytek celé aplikace vůči vláknu uživatelského rozhraní. To znamená, že každý jednotlivý test může běžet bez závislosti na jiném testu.

7.5 DATABÁZE

Hlavní databáze celého systému se nachází na straně serveru. Databáze postupuje podle našeho návrhu ze sekce 6.4.

Velikou zajímavostí je implementace konkrétního databázového systému. V momentálním stavu je implementován Microsoft SQL server, ale to pro nás nečiní žádnou zábranu toto řešení změnit. Navrhli jsme systém tak, aby pro změnu databázového systému byla zapotřebí změnit pouze jedna řádka kódu. Tato změna je potřeba provést v projektu `ServerRoom.Web.Server` ve třídě `Data/DbContextOptionsBuilderExtensions`, kde se nachází specifikace databázového systému použitého pro EntityFramework[30]. Na obrázku 7.20 můžeme vidět aktuální použití Microsoft SQL serveru na našem serveru.

```
/// <summary>
/// Use series of options intended for this app
/// </summary>
/// <typeparam name="TBuilder">Builder specification (generic optionally)</typeparam>
/// <param name="builder">The builder</param>
/// <returns>The builder for chaining</returns>
2 references
public static TBuilder UseServerRoomOptions<TBuilder>(this TBuilder builder)
    where TBuilder : DbContextOptionsBuilder
{
    // Setup connection
    builder.UseSqlServer(Framework.Construction.Configuration.GetConnectionString("DefaultConnection"));

    // Return builder for chaining
    return builder;
}
```

Obrázek 7.20: Specifikace použitého databázového systému (MSSQL)

Jedinou změnou metody „UseSqlServer"(obr. 7.20) a textového řetězce v konfiguraci webového serveru lze snadno zcela vyměnit databázový systém a jeho připojení k němu.

7.5.1 Ukládání rutin

Již v sekci 6.4.1.1 bylo naznačeno, že náš databázový model bude v něčem speciální. Tato úprava se dotýká testových rutin. V této sekci si řekneme a ukážeme, jak provádíme ukládání testových rutin bez závažného dopadu na bezpečnost a ztrátu dat.

Každá rutina (bez ohledu na její typ) se nachází v jedné databázové tabulce `Routines`. Zde je definovaný atribut `Type`, který definuje třídu podle programové struktury ENUM -

DeviceStatus. Každá rutina se ovšem liší parametry, které jsou pro každý typ specifické. Pro takové řešení se zde nachází dva atributy tabulky definované pomocí technologie JSON[45]:

- State - Definuje parametry, které se uchovávají z předešlého testu pro test nový.
- Configuration - Definuje parametry rutiny, podle kterých se rutina řídí.

V projektu ServerRoom.Core je definována klíčová třída RoutineExtensions pro možnost práce s těmito JSON řetězci. Nachází se zde několik metod a mezi ty nejvýznamnější patří: SerializeConfiguration, DeserializeConfiguration, SerializeState a DeserializeState. Od názvů těchto metod se dá snadno odvodit jejich funkce, kde *Serialize* znamená převedení programového objektu do textového řetězce a *Deserialize* je opakem. Existují vždy obě dvě metody právě pro State a Configuration. Jako příklad zde na obrázku 7.21 můžeme vidět metodu pro deserializaci konfigurace rutiny. Jedná se o extension metody[19] pro datový model rutiny (RoutineDataModel). Konkrétní použití těchto metod je blíže specifikováno v sekci 7.6.

```
/// <summary>
/// Compose routine configuration into string for routine details
/// </summary>
/// <typeparam name="T">Specific routine type configuration data model</typeparam>
/// <param name="routine">The routine</param>
/// <param name="configuration">The configuration</param>
/// <returns>The routine for chaining</returns>
0 references
public static RoutineDataModel SerializeConfiguration<T>(this RoutineDataModel routine, T configuration)
    where T : ARoutineConfigurationDataModel
{
    // Save new configuration
    routine.Configuration = JsonConvert.SerializeObject(configuration);
    // Update time of configuration change
    routine.ConfigurationChangedAt = DateTime.Now;

    // Return routine for chaining
    return routine;
}
```

Obrázek 7.21: Metoda deserializace konfigurace rutiny

Při ukládání rutin je vždy datový model konfigurace deserializován za pomoci zmíněných metod a při opětovném načítání dat z databáze je obsah konfigurace zase opětovně serializován do objektů. Programové struktury automaticky počítají, že ve výchozím stavu nemusí konfigurace (Configuration) ani stavové hodnoty (state) existovat, z toho důvodu lze snadno předpokládat, že nemůže dojít k závažným problémům. Může se však najít případ, kdy bude upraven datový model konfigurace nebo stavových hodnot (přidané nebo odstraněné pole). V takovém případě to pro nás nečiní překážku, protože pokud je identifikována hodnota navíc,

tak je ignorována a na druhou stranu, pokud některá z hodnot chybí, tak je nahrazena výchozí hodnotou. Uživatel má vždy možnost vstupy upravit. Posledním možným problémem je nekonzistence dat. S daty není nijak manipulováno než přes již zmíněné metody a putují přímo do datových modelů. Pokud by i přes tyto zábrany zde nastala nekonzistence dat, tak bude hodnota ignorována a opět nahrazena výchozí hodnotou.

7.6 SPRÁVA RUTIN

V této sekci si řekneme a ukážeme, jak snadno je možné spravovat a používat rutiny, jak tyto rutiny rozdělujeme do data modelů, kde uchováváme jejich data a nastíníme případy možných přidání definice nové rutiny.

Rutina se dělí celkem na tři data modely:

- `RoutineDataModel` - Specifikuje databázovou tabulku a tím i kopíruje všechna data, která se v datové tabulce nachází.
- `[Ping|Ups]StateDataModel` - Definuje strukturu stavových hodnot jednotlivých typů rutin. Tento model je následně ukládán jako JSON řetězec do databázové tabulky `Routines` (`RoutineDataModel.State`). Tento data model implementuje abstraktní třídu `ARoutineStateDataModel`, která dodává data modelu nutnou konfiguraci a zároveň vynucuje implementaci nezbytných metod pro správnou funkčnost stavových hodnot (například metoda `ToString`, která převede stavové hodnoty uživateli do prezentující formy).
- `[Ping|Ups]ConfigurationDataModel` - Definuje strukturu konfigurace jednotlivých typů rutin. Tento model je následně ukládán jako JSON řetězec do databázové tabulky `Routines` (`RoutineDataModel.Configuration`). Tento data model implementuje abstraktní třídu `ARoutineConfigurationDataModel`, která dodává data modelu nutnou konfiguraci a zároveň vynucuje implementaci nezbytných metod a properties[22] pro správnou funkčnost této konfigurace. Hlavní přidanou hodnotou je property `AllowedDataConnectorTypes`, která definuje pole povolených „data konektorů"pro tento typ rutiny pomocí programové struktury ENUM `DataConnectorType`.

Pokud se tedy znovu podíváme na naše řešení, máme zde atribut databáze obsahující řetězec JSON, který pomocí metod zmíněných v sekci 7.5.1 snadno dokážeme převést do konkrétního datového modelu podle typu rutiny. Pokud by nastala situace potřeby změny

konfiguračních hodnot, tak jediné místo, kde potřebujeme upravit tato data, je právě data model `ARoutineConfigurationDataModel` (nepočítáme uživatelské formuláře). V běžné situaci bychom museli upravit i samotnou databázi.

7.6.1 Data Connectors

Nedílnou součástí jsou tzv. „data konektory (data connectors)". Jedná se o nadstavbu pro rutiny, která rutinám umožní získávat externí data nezbytná pro testování. Při definici data modelu `ARoutineConfigurationDataModel` jsme se seznámili s property `AllowedDataConnectorTypes`, která nám umožní specifikovat u každého typu rutiny, jaké data konektory může daný typ rutiny použít. Na základě toho se v celé aplikaci pro tento typ rutiny zobrazí příslušné uživatelské vstupy (nebo nezobrazí).

Máme zde vytvořený zatím jeden data konektor, kterým je SNMP[73]. Vytvořili jsme podporu pro SNMP verze 1 a 2. Existuje i verze 3, která zatím není implementována z důvodu potřeby otestovat software nejdříve se základními verzemi. Verze 3 se liší uživatelskými vstupy verzemi, z tohoto důvodu jsme se nejdříve rozhodli implementovat verzi SNMP bez nutnosti změny uživatelských vstupů.

Každý data konektor má definovanou svoji databázovou tabulku podle jeho data modelu (`SnmpConnectorDataModel`). V případě SNMP zde potřebujeme i vedlejší tabulku pro uchování tzv. OID[66], která definují data, se kterými manipulujeme pomocí SNMP.

7.7 TESTOVÁNÍ RUTIN

Pro každou rutinu existuje speciální třída implementující rozhraní `IRoutineTest`. Tyto třídy se nacházejí v projektu webového serveru v adresáři *RoutineTesting/Routines*. Zde implementujeme testovací třídy, které se starají jen a pouze o samotné testování. Tyto třídy implementují metodu `RunAsync`, která představuje hlavní veřejnou metodu této třídy spouštějící testování. Strukturu a proces tohoto testování můžeme vidět na obrázku 7.22.

```
#region Constructor

/// <summary>
/// Default constructor
/// </summary>
1 reference
public PingRoutineTest(RoutineDataModel routine)
{
    Routine = routine;
}

#endregion

#region Run Test

/// <inheritdoc/>
4 references
public async Task<DeviceStatus> RunAsync(CancellationToken workerCancellationToken,
    CancellationToken deviceTimeoutCancellationToken = default)
{
    Get State

    Get Configuration

    Test

    Save State

    Return
}

#endregion
```

Obrázek 7.22: Základní struktura testovací třídy

Když se pozorně podíváte na strukturu zobrazenou na obrázku 7.22, můžeme vidět strukturu testu skládající se z pěti kroků: **Get State** (načtení stavových hodnot), **Get Configuration** (načtení konfigurace rutiny), **Test** (testovací proces), **Save State** (připraví stavové hodnoty pro uložení - serializace) a **Return** (vrátí volanému procesu hodnocení testu).

Na obrázku 7.23 můžeme vidět jednoduchost načítání a následného uložení stavových hodnot během testování. Tato metoda `RunAsync` se dá definovat také jako jádro, kde definujeme testovací logiku, zbylé části aplikací pouze zprostředkovávají správné přenesení dat k uživateli a jeho interakci s těmito daty.

```
#region Get State

// Initialize state
PingStateDataModel state = null;
// Get routine state
state = Routine.DeserializeState<PingStateDataModel>();
// If state does not exist, create a new empty one
state ??= new PingStateDataModel();

#endregion

Get Configuration

Test

#region Save State

// Serialize state (save)
Routine.SerializeState(state);

#endregion

Return
```

Obrázek 7.23: Načítání a ukládání stavových hodnot v testovací metodě PING

7.7.1 Postup přidání nové rutiny

V této sekci si ukážeme velmi jednoduše kroky, které je potřeba podstoupit pro přidání nového typu rutiny. Tato sekce by měla nastínit jednoduchost správy našeho systému.

- Přidání nové ENUM položky v `RoutineType` nacházející se v projektu `ServerRoom.Core` v adresáři `DataTypes`
- Vytvořit nový datový model `<TYPE>StateDataModel` pro stavové hodnoty tohoto typu rutiny v projektu `ServerRoom.Core` v adresáři `DataModels/Routines`
- Vytvořit nový datový model `<TYPE>ConfigurationDataModel` pro konfiguraci tohoto typu rutiny v projektu `ServerRoom.Core` v adresáři `DataModels/Routines`
- Vytvořit třídu `<TYPE>RoutineTest` v serverovém projektu `ServerRoom.Web.Server` v adresáři `RoutineTesting/Routines`
- Upravit programovou strukturu SWITCH ve třídě `RoutineExtensions` nacházející se v projektu `ServerRoom.Core` v adresáři `Utilities`

- Upravit programovou strukturu SWITCH ve třídě `RoutineHelpers` nacházející se v projektu `ServerRoom.Web.Server` v adresáři `RoutineTesting`
- Úprava uživatelských vstupů v klientské aplikaci
  - Vytvoření nového UserControl[89] (definice uživatelských vstupů pro konfiguraci) pro nový typ rutiny v projektu `ServerRoom` v adresáři `Controls/Forms/RoutineConfigurationInputs`
  - Vytvoření nového view modelu pro uživatelské vstupy konfigurace `Routine<TYPE>ConfigurationInputViewModel` v projektu `ServerRoom.Client.Base` v adresáři `ViewModels/Content/Forms/RoutineConfigurationInputs`
  - Přidat definici o novém UserControl na vyznačené místo do stránky přidávacího a editovacího formuláře rutiny (`RoutineAddFormPage/RoutineEditFormPage`) v projektu `ServerRoom` v adresáři `Pages/Forms`
  - Přidat definici vstupů do view modelů přidávacího a editovacího formuláře (`RoutineAddFormPageViewModel/RoutineEditFormPageViewModel`) v projektu `ServerRoom.Client.Base` v adresáři `ViewModels/Content/Forms`

Podle výše uvedeného postupu můžeme vidět, že se zde lehce opakují editace programové struktury SWITCH. V budoucím vývoji je to jedna z hlavních priorit, kterou chceme redukovat, ale i v aktuálním případě se jedná pouze o velmi klíčové struktury, ve kterých by byl programátor řádně upozorněn při zapomenuté implementaci.

7.8 VYBRANÉ FUNKCIONALITY SYSTÉMU

V této sekci si přiblížíme vybraná řešení, která aplikace používá. Provedeme detailní rozbor konkrétních řešení a ukážeme si jejich užitečnost přímo v našem systému.

7.8.1 Komunikace Klient-Server

Pro zajištění komunikace klientské aplikace se serverem využijeme API, které poskytuje náš server. Již v projektu Ixs.DNA (7.2.1) jsme zmiňovali programovou strukturu pro HTTP požadavky, kterou chceme pro tyto účely použít. Tato programová struktura pod názvem

`WebRequests` obsahuje několik metod, kde pro nás nejdůležitější je `PostAsync`, která odešle požadavek pomocí HTTP POST[69]. Pro její funkčnost vyžaduje parametr vytvářející následnou odpověď od serveru, kterou implmentujeme jako `ApiResponse<T>`, kde jejím generickým[21] parametrem je API datový model typu `Result` (ovšem tento návratový API datový model není vyžadovaný).

```
#region Data Server Request

// Load get data log response from server
var result = await WebRequests.PostAsync<ApiResponse<Result_DataLogApiModel>>(
    // Set URL
    RouteHelpers.GetAbsoluteRoute(ApiRoutes.GetDataLog),
    // Pass API model
    new Get_DataLogApiModel()
    {
        LastGetLogTime = mDataVM.LastServerGetLogTime,
        LimitFileSize = mDataVM.ServerLogFileUpdateBookmark,
        Select = mDataVM.LogLinesToSelect
    },
    // Pass in user Token
    bearerToken: token
    );

// If the response has an error...
if (await result.HandleErrorIfFailedAsync(Localization.Resource.DataUpdateWorker_LoadDataErrorTitle))
    // We are done
    return false;

#endregion
```

Obrázek 7.24: HTTP požadavek pro získání nových logů

```
public static async Task<bool> HandleErrorIfFailedAsync(this WebRequestResult response, string title)
{
    // If there was no response, bad data or a response with an error message...
    if (response == null || response.ServerResponse == null || (response.ServerResponse as ApiResponse)?.Successful == false)
    {
        // Default error message
        var message = Localization.Resource.WebRequestResultExtensions_HandleUnknownError;

        // Default api response
        ApiResponse apiResponse = null;
        // Check if API response exists
        if (response?.ServerResponse is ApiResponse)
            // Get API response
            apiResponse = (ApiResponse)response.ServerResponse;

        Get Response Message

        Handle Error

        // Return that we had an error
        return true;
    }

    // All was OK, so return false for no error
    return false;
}
```

Obrázek 7.25: Metoda pro zpracování chybného HTTP požadavku

Na obrázku 7.24 můžeme vidět běžnou strukturu webového požadavku, který se v kódu klientské aplikace často vyskytuje pro získání vzdálených dat. Můžeme zde vidět již popsanou strukturu `WebRequests`, která jako parametr používá `ApiResponse`. Také můžeme vidět, že `ApiResponse` získává dva další parametry, kde prvním je adresa API metody zpracovávající tento požadavek, druhým je odesílaný API data model ze strany klienta a třetím je autorizační token. V našem případě žádáme server o doplnění informací datového logu pro klientskou aplikaci. Můžeme tedy vidět odesílání `Get_DataLogApiModel` obsahující informace, které předáme serveru pro správné vyhodnocení požadavku a následné navrácení dat API datovým modelem `Result_DataLogApiModel`.

```
#region Handle Error

// If this is an unauthorized response...
if (response?.StatusCode == System.Net.HttpStatusCode.Unauthorized)
{
    // Log it
    FrameworkDI.Logger.LogInformationSource("Logging user out due to unauthorized response (401) from server");

    // Automatically log the user out
    await BaseDI.ViewModelSettings.LogoutAsync();
}
// If the user exists, but unauthorized to server...
else if (apiResponse != null && apiResponse.HasUnauthorizedUserError)
{
    // Log it
    FrameworkDI.Logger.LogInformationSource("Logging user out due to unauthorized user");

    // Automatically log the user out (delete saved client credentials)
    await BaseDI.ViewModelSettings.LogoutAsync();
}
else
{
    // Display error
    await BaseDI.UI.ShowSnackbar(new SnackbarDialogViewModel()
    {
        Title = title,
        Message = message
    });
}

#endregion
```

Obrázek 7.26: Možné případy zpracování chyby HTTP požadavku

Druhou částí odeslaného požadavku je validace následně příchozí zprávy. Na obrázku 7.24 je metoda `HandleErrorIfFailedAsync`, která se postará o možné chyby, které mohou při HTTP požadavku nastat. Na obrázku 7.25 můžeme vidět již zmíněnou metodu `HandleErrorIfFailedAsync`. Obrázek 7.26 ukazuje řešení, jak naložit s chybným HTTP požadavkem. Můžeme na něm vidět, že v první řadě testujeme, jestli odpověď od serveru vůbec existuje. Pokud odpověď existuje, pokračujeme v jejím rozboru a hledáme záznam o neautorizovaném přístupu, při kterém je potřeba uživatele odhlásit z klientské aplikace. Můžeme

zde vidět dva typy ošetření neautorizovaného přístupu. První přístup, který se kontroluje, je autorizace samotného požadavku. Druhým způsobem šetření je autorizace konkrétního uživatele, který požadavek posílá.

Na straně serveru zde máme `ApiController` obsahující metody, které představují naše API rozhraní. Jak můžeme na obrázku 7.27 vidět, metoda má definováno, že má být dostupná pouze přes HTTP POST[69] na adrese serveru s relativní adresou specifikovanou pomocí atributu `Route`.

```
/// <summary>
/// Get data log messages
/// </summary>
/// <param name="model">The model containing about how to get log messages</param>
/// <returns>Returns successful response containing desired log message list with necessary variables</returns>
[HttpPost]
[Route(ApiRoutes.GetDataLog)]
0 references
public async Task<ApiResponse<Result_DataLogApiModel>> GetDataLog([FromBody]Get_DataLogApiModel model)
{
    #region Get/Check User

    // Get user by the claims
    var user = await mUserManager.GetUserAsync(HttpContext.User);
    // If user does not exit in database...
    if (user == null)
        // Return user auth error response
        return GetAuthorizationErrorApiResponse<Result_DataLogApiModel>(Localization.Resource.AuthError_UserNotFound);

    #endregion

    // Get last file clean from data logger
    var lastFileClean = DI.DataLogger.LastFileCleanDate;
    // Init default value for file size limitation
    var limitFileSize = default(long);

    // Check log file consistance by last clean time
    // If log file clean NOT happend during the time from last log get...
    if (DateTime.Compare(lastFileClean, model.LastGetLogTime) < 0)
        // Set limit file size
        limitFileSize = model.LimitFileSize;

    // Get log list
    var logList = DI.DataLogger.Tail(model.Select, out long currentFileSize, limitFileSize, skipFirstOccurance: true);

    // Return routine to user
    return new ApiResponse<Result_DataLogApiModel>
    {
        // Pass back the device list
        Response = new Result_DataLogApiModel
        {
            LogMessages = logList,
            FileSize = currentFileSize,
            ServerGetLogTime = DateTime.Now
        }
    };
}
```

Obrázek 7.27: Metoda API pro získání data logů

Na obrázku 7.27 se dále můžeme všimnout regionu `Get/Check User`, který autorizuje

uživatele (všechny metody automaticky vyžadují autorizaci, pokud není dáno jinak). Při chybě je vrácena odpověď klientovi o chybné autorizaci. Zbytek metody definuje rozbor všech s sebou přinesených parametrů ze strany klienta a podle těchto parametrů jsou vybrány řádné logové zprávy, které jsou následně odeslány zpět na tohoto klienta.

7.8.2 Validace dat

Validace dat je nedílnou součástí každého systému či aplikace a vždy může být řešená jinak. V našem systému validaci řešíme pomocí atributů tříd a metod, kde definujeme svoje vlastní atributy s vlastní logikou.

Definujeme rozhraní `IValidableAttribute`, které tvoří základ naší implementace validačního atributu. Příkladem může být atribut `ValidateIntegerAttribute` zobrazený na obrázku 7.28. Třída je zde definována jako atribut použitelný striktně pro properties[22] s implementovanými metodami `Validate`. Vstupním parametrem těchto metod je vždy hodnota, která má být validována. Výstupem je třída `DataValidateResult` představující výsledek validace nebo přímo seznam validačních chyb (třída `DataValidateResult` kopíruje logiku třídy `IdentityResult`[39]).

V každé takové třídě atributu se nachází properties, které udávají validační parametry a zároveň jsou povinnými parametry konstruktoru.

```
/// <summary>
/// Validation, property attribute of type <see cref="int"/>
/// </summary>
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
19 references
public class ValidateIntegerAttribute : BaseValidateAttribute, IValidableAttribute<int>
{
    Public Properties

    Constructor

    #region Interface Methods

    /// <inheritdoc/>
    7 references
    public DataValidationResult Validate(int value)...

    /// <inheritdoc/>
    20 references
    public ICollection<DataValidationError> Validate(int value, object none)...

    #endregion
}
```

Obrázek 7.28: Třída validace typu integer

S vytvořenými atributy pro validaci jednotlivých datových typů můžeme provést aplikaci v datových modelech. Pro aplikování atributu a jeho parametrů potřebujeme vytvořit tzv. limitní konstanty v daném datovém modelu. Na obrázku 7.29 můžeme vidět příklad přiřazení těchto konstant do validační atributu. Konstanty pojmenujeme podle názvu samotné property s přidáním pojmenování limitního pravidla - např. minimální hodnota (MinValue).

Tyto validační limitní konstanty bychom mohli přímo definovat do validačního atributu. Výhodou samostatné definice je snadný přístup k těmto datům, pokud je potřebujeme použít mimo samotnou validaci (například pro výpis pravidel uživateli).

```
/// <summary>
/// The device timeout for all routines
/// </summary>
[ValidateInteger(nameof(Timeout), typeof(DeviceDataModel),
    pMinValue: nameof(Timeout_MinValue),
    pMaxValue: nameof(Timeout_MaxValue))]
12 references
public int Timeout { get; set; } = 0;
```

Obrázek 7.29: Přiřazení validačního atributu
(Parametry validačního atributu jsou: označení, typ třídy dané property, minimální dovolená hodnota a maximální dovolená hodnota)

Časem bychom došli do situace, kdy bude potřeba validovat celý datový model. Pro takové případy můžeme definovat pro celou třídu validační atribut `ValidateModelAttribute`

(obr. 7.30). Tento atribut implementuje stejné rozhraní jako atributy jednotlivých datových typů. Na místo validace se vkládá celý datový model, který prochází po všech properties a validuje ty, které jsou označeny pomocí validačních atributů. Takové řešení pro nás činí velmi jednoduchou validaci, která je vždy přístupná pomocí reference datového modelu.

```
/// <summary>
/// Mark the class (model) as validable and add functinality into it to validate it
/// </summary>
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
5 references
public class ValidableModelAttribute : Attribute, IValidableAttribute<object>
```

Obrázek 7.30: Validační atribut datového modelu

V případě vložení validace na stranu klienta pro uživatelské vstupy musíme přidat definici do jednotlivých vstupních polí. V klientské aplikaci máme vytvořené celé struktury pro jednotlivé vstupní pole podle typu. Abychom modifikovali tyto vstupy a přidali validaci na stranu klienta, tak vytvoříme validační referenci, kterou budeme moci přidat při specifikaci jednotlivých uživatelských polí (obr. 7.31).

```
/// <summary>
/// A base view model that describes abstract items for entry view models
/// </summary>
/// <typeparam name="T">Type of the value</typeparam>
5 references
public abstract class BaseEntryViewModel<T> : BaseEntryViewModel
{
    Private Members

    #region Public Properties

    /// <summary> Custom setter process for OriginalValue and EditedValue Does not s ...
    8 references
    protected abstract Func<T, T> ValueCustomSetterProcess { get; }

    /// <summary> The current saved value (selected)
    99+ references
    public T OriginalValue...

    /// <summary> The current non-commit edited value --- Make it behave as Original ...
    9 references
    public T EditedValue...

    /// <summary>
    /// Validation attribute container to bind specific validation into the entry
    /// </summary>
    48 references
    public IValidableAttribute<T> Validation { get; set; }

    #endregion

    Constructor

    Public Methods
}
```

Obrázek 7.31: Reference validace pro vstupní pole (entries)

Implementaci z obrázku 7.31 následně použijeme při vytváření jednotlivých uživatelských polí pro naše šablony (příklad na obrázku 7.32). Zde můžeme vidět definici uživatelského pole ve view modelu formuláře pro zařízení o hodnotě Timeout.

```
// Set initial unloaded values / Timeout
Timeout = new IntegerEntryViewModel
{
    DisabledCommitMechanism = true,
    Label = Localization.Resource.DeviceXFormPageViewModel_TimeoutFieldTitle,
    OriginalValue = 0,
    MinValue = DeviceDataModel.Timeout_MinValue,
    MaxValue = DeviceDataModel.Timeout_MaxValue,
    Validation = typeof(DeviceDataModel).GetPropertyAttribute<ValidateIntegerAttribute>(nameof(DeviceDataModel.Timeout))
};
```

Obrázek 7.32: Vytvoření nového uživatelského pole (Device/Timeout)

Dále potřebujeme provést poslední úkon, a tak vložíme validaci do samotné šablony

vstupního pole. Obrázek 7.33 nám ukazuje definici této validace pro uživatelský vstup očekávající číselnou hodnotu. Tato definice na první pohled nevypadá jednoduše, ale zároveň je potřeba ji definovat pouze jednou v celém systému. Implementace je zajištěna pomocí AttachedProperties[87], díky kterým je možné předat data do struktury ValidationRules[90] (obr. 7.34), která zajišťuje výpis validačních chyb pro WPF v reálném čase.

```
<xctk:IntegerUpDown
    Maximum="{Binding MaxValue}"
    Minimum="{Binding MinValue}">
    <xctk:IntegerUpDown.Resources>
        <local:BindingProxy x:Key="TargetProxy" Data="{Binding Validation}" />
    </xctk:IntegerUpDown.Resources>
    <xctk:IntegerUpDown.Value>
        <Binding Path="EditedValue" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <local:DataIntegerValidationRule ValidationStep="UpdatedValue">
                    <local:ValidationIntegerAttributeProperty Value="{Binding Data, Source={StaticResource TargetProxy}}" />
                </local:DataIntegerValidationRule>
            </Binding.ValidationRules>
        </Binding>
    </xctk:IntegerUpDown.Value>
</xctk:IntegerUpDown>
```

Obrázek 7.33: Použití validace v šabloně

```
/// <summary>
/// Validation rule for data of type <see cref="int"/>
/// </summary>
[ContentProperty(nameof(ValidationProperty))]
1 reference
public class DataIntegerValidationRule : BaseValidationRule
{
    /// <summary>
    /// Validation dependency property reference
    /// </summary>
    4 references
    public ValidationIntegerAttributeProperty ValidationProperty { get; set; }

    /// <inheritdoc/>
    2 references
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        // Get value
        var val = (int)GetBoundValue(value);

        // Check if validation is NOT attached...
        if (ValidationProperty == null || ValidationProperty.Value == null)
            // Validation does not exist, leave it without error
            return ValidationResult.ValidResult;

        // Run validation
        var errors = ValidationProperty.Value.Validate(val, null);
        // If validation has any errors...
        if (errors.Count > 0)
            return new ValidationResult(false, errors.AggregateErrors());

        return ValidationResult.ValidResult;
    }
}
```

Obrázek 7.34: ValidationRules[90] obsluhující validaci

8. Testování

V následující kapitole je popsáno testování našeho systému. Nejprve popíšeme zařízení, na kterých byl náš systém testován. Dále následuje popis jednotlivých testovacích scénářů spolu s jejich výsledky.

8.1 Testovací zařízení

Mezi testovací zařízení, která byla použita pro testování našeho systému, se řadí i můj osobní počítač, na kterém byl celý systém vyvíjen. Další zařízení jsou z naší cílové firmy, na kterých jsme systém testovali. Seznam všech testovaných zařízení naleznete níže (8.1, 8.2, 8.3, 8.4).

Stolní počítač	
Operační systém	Windows 10 Pro, 1903
CPU	Intel Core i7-6700k
RAM	32GB, 2400 MHz, DDR4
Disk	SATA SSD
Rozlišení obrazovky	2560 x 1440 / 3840 x 2160

Tabulka 8.1: Testované zařízení 1

Stolní počítač	
Operační systém	Windows 10 Pro, 1909
CPU	Intel Core i5-3470
RAM	16GB, 3200 MHz, DDR4
Disk	SATA SSD
Rozlišení obrazovky	1920 x 1080

Tabulka 8.2: Testované zařízení 2

UPS	
Typ	APC Smart-UPS SRT 5000VA LCD RM 230V

Tabulka 8.3: Testované zařízení 3

UPS	
Typ	APC Smart-UPS SRT 192V 5kVA and 6kVA RM Battery Pack

Tabulka 8.4: Testované zařízení 4

8.2 TESTOVACÍ SCÉNÁŘE

V této kapitole budou popsány testovací scénáře spolu s jejich výsledky, kterými byla aplikace testována na testovacích zařízeních uvedených v kapitole 8.1.

Při těchto testech předpokládáme, že uživatel již má spuštěný server, klientskou aplikaci (dostupnou ve stejné počítačové síti) a má vytvořeného výchozího uživatele `admin`.

8.2.1 Odpojení a připojení k serveru

Scénář:
Na hlavní obrazovce klientské aplikace vyberte možnost `SERVER SETTINGS`. Zde pomocí tlačítka označeného „X"zastavte aktuální připojení k serveru. Pokud se přesunete zpět na hlavní obrazovku a pokusíte se o přihlášení, tak nebude možné akci uskutečnit (pokud jste již přihlášeni, tak při přechodu na obrazovku `DASHBOARD` nebude možné vidět žádná data - zařízení). Nyní se vraťte zpět na obrazovku `SERVER SETTINGS`, kde nastavte připojení na smyšlený server (simulace chybného připojení). Ve stejném okamžiku byste měli být informováni, že připojení k serveru není dostupné (tato zpráva se aktualizuje v cyklu 20ti sekund). Následně zpět nastavte připojení na správný server. Do 20ti sekund by upozornění na chybné připojení mělo zmizet a data na obrazovce `DASHBOARD` by měla být znovu dostupná.

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Server správně zareagoval při chybném nastavení serveru a aplikace uživatele informovala o chybném připojení a nedovolila mu přístup k datům.

8.2.2 Přihlášení/Odhlášení uživatele (klient)

Scénář:
Na hlavní obrazovce vyberte možnost LOGIN a na nadcházející obrazovce se přihlaste na uživatele admin (Admin123456789). Vyzkoušejte chybné údaje, které povedou k chybnému přihlášení a aplikace Vás o chybných údajích upozorní prostřednictvím notifikace. Po úspěšném přihlášení budete přesunuti na obrazovku DASHBOARD. Pokuste se odhlásit a znovu přihlásit. Pro odhlášení vyberte možnost USER SETTINGS (pravý horní roh) na obrazovce DASHBOARD. Na obrazovce USER SETTINGS naleznete možnost LOGOUT. Po odhlášení budete automaticky přesunuti na přihlašovací stránku klientské aplikace. Pokuste se celý proces ještě jednou zopakovat s očekávaným stejným výsledkem.

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Klient správně brání přenosu dat, pokud není uživatel přihlášen. Uživatel je vždy po odhlášení přesunut mimo obrazovky, na které v tu dobu nemá oprávnění.

8.2.3 Přihlášení/Odhlášení uživatele (web)

Scénář:
Ve webové části serveru se přesuňte z hlavní obrazovky (index) do přihlašovacího formuláře LOGIN. Přihlaste se na uživatele admin (Admin123456789). Ověřte přihlášení s chybnými údaji a ověřte, že chybné údaje nebudou považovány jako správné. Po úspěšném přihlášení budete přesměrováni na stránku Viewer, která obsahuje všechna zařízení, které testujete. V pravém horním rohu naleznete box s uživatelským jménem, který po rozbalení nabídne možnost odhlášení Logout. Po odhlášení budete přesměrováni na výchozí stránku webové aplikace bez možnosti přístupu zpět na data (Viewer - <ADRESA_SERVERU>/viewer). Po přesměrování zpět na adresu stránky Viewer (bez oprávnění) budete okamžitě přesměrováni na formulář pro přihlášení.

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Server správně brání uživateli k přístupu na stránky s nutným oprávněním.

8.2.4 Vytvoření nového zařízení s rutinami

Scénář:
Přihlaste se v klientské aplikaci na uživatele admin (Admin123456789). Na obrazovce DASHBOARD vyberte možnost ADD NEW DEVICE. Vyplňte formulář dle vlastního uvážení a vyzkoušejte i chybné hodnoty (například speciální znaky nebo písmena v poli IP adresy). Pokud některé z hodnot budou mimo dovolený rozsah než nabízí vstupní pole nebo hodnota nebude v požadovaném tvaru, budete upozorněni prostřednictvím notifikace. Při úspěšném vytvoření zařízení budete přesměrováni na obrazovku DASHBOARD, kde by se vaše zařízení mělo objevit v seznamu do 20ti sekund. Zařízení bude označené žlutým podbarvením se statusem NOSTATUS a bude zastavené (Stopped). U tohoto zařízení naleznete možnost ozubeného kola, kterým se dostanete do editace tohoto zařízení. Změňte některý z údajů tohoto zařízení a ujistěte se na obrazovce DASHBOARD, že údaje byly změněny.
Při návratu do editace zařízení zde můžete vidět možnost přidání nové rutiny (ADD NEW ROUTINE). Pomocí toho se přesunete do formuláře pro vytvoření nové rutiny. Vytvořte novou rutinu typu UPS. Po nastavení typu rutiny se rozevřou přídavná pole konfigurace, která vyplňte dle vlastního uvážení. Vyzkoušejte také možnost odeslání formuláře pro vytvoření nové rutiny bez nastaveného typu rutiny. Takové odeslání by Vás mělo upozornit na chybu. Pro úspěšné odeslání nastavte všechny hodnoty konfigurace a aplikujte konfiguraci APPLY CONFIGURATION. Poslední částí nastavení je možnost získání externích dat UPS. Budete mít možnost výběru SNMP a jeho nastavení, které toto připojení běžně poskytuje. Vytvořte tuto rutinu s nevyplněnými hodnotami (nebo nastavte vlastní) a rutinu vytvořte. Vytvoření by mělo proběhnout úspěšně a budete přesměrováni zpět do editace zařízení. Zde můžete ověřit, že seznam rutin již obsahuje vámi vytvořenou rutinu typu UPS a při editaci této rutiny můžete ověřit hodnoty konfigurace, zda se shodují s těmi, které jste nastavili. Zároveň při návratu do obrazovku DASHBOARD uvidíte vaše zařízení ve stavu běžícím (Running).

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Zařízení a rutina na chybná data reagovala bez problémů a uživatele řádně upozornila. Po vytvoření zařízení a rutiny testování začalo, jak bylo očekáváno.

8.2.5 Start/Stop zařízení

Scénář:
Přihlaste se v klientské aplikaci na uživatele admin (Admin123456789). Na obrazovce DASHBOARD

si vyberte zařízení a pomocí Start/Stop tlačítka zařízení zastavte (pokud tomu tak již není). Zařízení by se mělo do 20ti sekund zastavit a změnit stav na `NOSTATUS`. Po opětovném spuštění by se do 20ti sekund mělo spustit a přesunout do stavu `Running` s jiným statusem než `NOSTATUS`. Pokud spustíte zařízení, které neobsahuje žádné rutiny, tak zařízení nenastartuje ani nezmění stav.

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Spouštěním a zastavováním testování zařízení fungovalo bez problémů.

8.2.6 Odstranění rutiny zařízení

Scénář:
Přihlaste se v klientské aplikaci na uživatele `admin` (`Admin123456789`). Na obrazovce `DASHBOARD` si vyberte zařízení (nebo vytvořte), které již obsahuje jakoukoli rutinu. Ujistěte se, že zařízení je v provozu (`Running`). Následně s přesuňte do nastavení tohoto zařízení, přes které se dostanete do nastavení konkrétní rutiny. Odstraňte všechny tyto rutiny. Výsledkem by mělo být zastavené zařízení se statusem `NOSTATUS` (bude aktualizováno do 20ti sekund na obrazovce `DASHBOARD`).

Vyhodnocení:
Aplikace reagovala na scénář dle očekávání. Zařízení je správně uvedeno do stavu zastavení, pokud neobsahuje žádné rutiny k testování.

8.3 JEDNOTKOVÉ TESTY

Náš systém má definovaný projekt `ServerRoom.Web.Server.Test`, který je plně připraven pro jednotkové testování. V kódu se vyskytují celkem dvě hlavní části, které považujeme za kritické a je potřeba dbát vyšší obezřetnost při jejich implementaci. Mezi tyto části se řadí API „controller"a jednotlivé testy rutin. V aktuální fázi vývoje máme splněný a vyzkoušený fungující „prototyp"systému, ale je potřeba provést několik úprav, které umožní snadný budoucí vývoj. V kapitole implementace 7 jsme definovali zkratkové třídy pro snadné získávání dat z DI[26]. Krok, který je potřeba upravit, se týká použití těchto zkratkových metod. V aktuální fázi používáme tyto zkratkové reference přímo bez uchování referencí ve třídách, které tato data používají. Takové řešení je pohodlné pro programátora, ale činí jistou překážku pro

možnost jednotkového testování.

V dalším vývoji chceme pročistit kód od nepotřebných struktur a přidat správné reference, abychom umožnili jednotkovému testování „mockovat"[51] objekty. V mnoha částech kódu se nachází logovací systém, který je právě pomocí DI implementován a následně používán. Z důvodu přímého použití z DI se jedná o statickou referenci, kterou nelze „mockovat", tudíž je to překážkou pro naše testování. Nejedná se však o nijak závažný zákrok a po řádné kontrole kódu bude jednotkové testování možné uskutečnit.

Další možnou potíží je fakt, že nelze „mockovat"jiné než veřejné objekty. Z tohoto důvodu to může na jistých místech kódu činit obtíž, protože zde máme hodně externího připojování - klient server nebo třeba SNMP[73]. Některá z těchto spojení jsou implementována s logikou, která by neměla být správně veřejně dostupná programátorovi.

9. Budoucí vývoj

Tato kapitola je zaměřena na popis plánovaného budoucího vývoje našeho systému a zároveň na jeho možná další rozšíření. Existuje celá řada možností, kterými můžeme aplikaci rozšiřovat. V této kapitole se budeme věnovat hlavně těm, které jsou přednostně důležité.

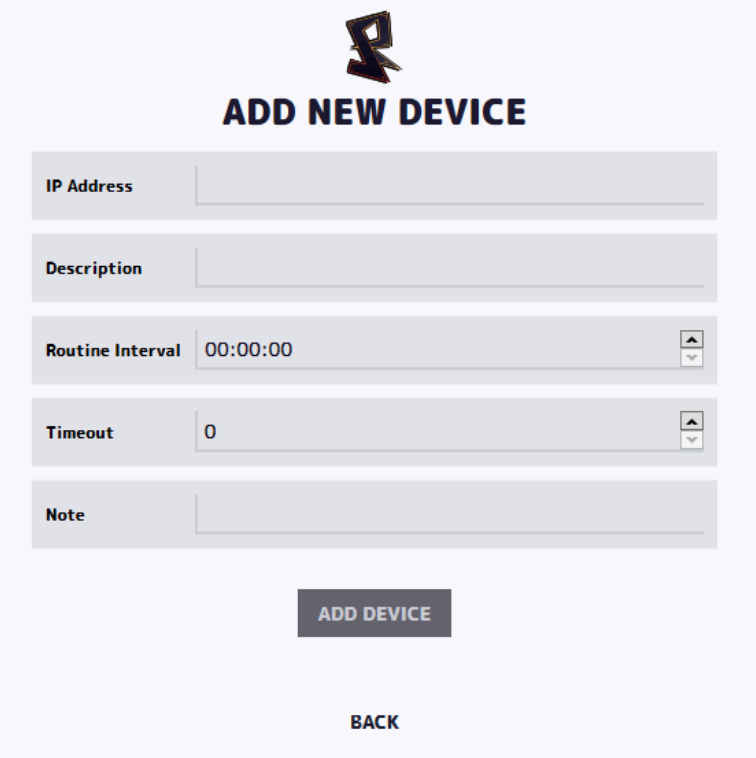
9.1 Plánovaná rozšíření aplikace

V současné době jsou plánována další rozšíření systému, které umožní detailnější monitorování firemní sítě.

9.1.1 Uživatelské rozhraní

Jedna z hlavních věcí, která je potřeba doladit je vzhled uživatelského rozhraní. Na některých místech, jako jsou třeba formuláře, se jedná již o hotový design. Jsou však části aplikace, které chceme v budoucnu navrhnout lépe.

Příkladem může být formulář pro přidání nového zařízení, který je již ve finální podobě (obr. 9.1).



Obrázek 9.1: Formulář pro přidání nového zařízení

9.1.2 Vlastní platforma pro aktualizace

Jednou z hlavních věcí, kterou chceme implementovat je možnost automatické aktualizace ze serveru vývojáře. S tímto řešením již mám osobně zkušenost a pro uživatele bude velmi přívětivé mít systém vždy aktualizovaný bez nutnosti manuální aktualizace.

9.1.3 Zpětná vazba uživateli

Aplikace nabízí možnost validace dat na straně klienta a i upozornění klienta na chybně zadané údaje. Nicméně chceme rozšířit zpětnou vazbu ohledně instrukcí těchto uživatelských vstupů. Pokud se jedná například o uživatelské vstupy vyžadující měrné jednotky, uživatel není dostatečně informován, ve kterých jednotkách má být hodnota zadávána.

Chceme vylepšit tyto uživatelské vstupy o speciální informační blok, ve kterém tyto informace můžeme zmínit a tím zjednodušit uživateli definice svých vstupních informací.

9.1.4 Další testovací metody

V aktuální verzi našeho systému máme pouze dvě hlavní testovací metody. Do budoucna tento počet chceme zvednout a rozšířit testovací metody o metody uvedené v kapitole 2. V aktuálním stavu pro nás naše dvě hlavní testovací metody činí měřítko, podle kterého můžeme snadno upravit chyby v systému a připravit jej pro možný nárůst těchto metod.

9.1.5 Zvukové a SMS upozornění

Náš systém nabízí pouze grafické upozornění. Již v aktuální době má připravené programové struktury, aby zvládl i zvukové nebo SMS upozornění, ale chybí samotná implementace těchto modulů.

9.1.6 Prázdninový kalendář

Do budoucna chceme přidat do našeho systému možnost výběru prázdninového kalendáře, který bude sloužit pro možnou odstávku zařízení po danou dobu.

Pokud uživatel ve své firmě má testy, které závisí na fyzické přítomnosti, tak po dobu prázdnin by mohlo nastat, že takové testy budou hlásit chybu. Pokud uživatel bude chtít tomuto zabránit, tak mu umožníme vybrat dny, ve kterých se testování nebude provádět.

9.1.7 Priorita rutin

Chceme přidat možnost nastavit rutinám prioritu, aby bylo snadno nastavitelné, která rutina má být vykonána jako první a která jako další.

9.1.8 Import/Export zařízení

Další vlastností, kterou chceme implementovat je možnost exportovat nebo importovat celou strukturu nastavených zařízení. Díky tomuto řešení umožníme uživateli snadno přenášet nastavení, zálohovat je nebo nastavovat zařízení pomocí textového souboru.

9.1.9 Správa uživatelů

Chceme rozšířit možnosti správy uživatelů. Aktuálně se v celém systému nachází pouze výchozí jeden uživatel - admin. Plánem je přidat možnost pro vytvoření uživatele a přidělit

mu role, podle kterých případně zamezí přístup do některých sekcí systému (například právo čtení bez zápisu - uživatel bude moci pouze data číst, nikoli měnit).

9.1.10 Jednotkové testy / codereview

Potřebujeme znovu projít celou strukturu a ucelit řešení, zkontrolovat konvenci pojmenování programových struktur a upravit místa kódu, která neodpovídají standardům MVVM[55]. Na základě těchto úprav budeme moci implementovat jednotkové testy, které momentálně v systému chybí. V aktuální době se nejedná o závažný problém, ale s možným přírůstkem testovacích metod by hrozilo riziko složitějších úprav při nalezení chyby.

9.1.11 Viewer rozhraní pro grafovou prezentaci dat

Chceme rozšířit náš webový zobrazovač dat o informační stránku prezentace dat, která bude uvádět statistiky celého systému s možností přizpůsobení zobrazených dat.

9.2 Možná rozšíření aplikace

V této sekci uvedeme rozšíření, o kterých zvažujeme jejich implementaci a na budoucím posudku hotového systému bude teprve rozhodnuto, jestli jsou tato řešení potřebná nebo ne.

9.2.1 Speciální test pro I/O moduly

Již v sekci analýzy (2.3) jsme zmínili možná rozšíření o I/O moduly. Tyto moduly by v budoucnu mohly snadno řešit vlastní problematiku testování. Uživatel by snadno mohl testovat vstupy těchto zařízení, na základě kterých upraví hardware těchto I/O modulů do svého požadovaného stavu.

9.2.2 Správa uživatelů

Rozšíření uživatelů můžeme vzít ještě trochu globálnějším směrem. Pokud firma využívající náš systém spravuje uživatele pomocí Microsoft řešení (ActiveDirectory[10]), tak se zde dostáváme ke značné výhodě našeho designu databázového modelu. Totiž, databázový model je generovaný z výchozího Microsoft designu, který umožňuje snadnější napojení do systému ActiveDirectory. Systém je na toto řešení připraven a rozhodně je to jedno z možných rozšíření, které umožní uživateli snadněji spravovat své zaměstnance s automatickým přístupem ke kontrole zařízení.

10. Závěr

Cílem této bakalářské práce bylo navrhnout a vytvořit monitorovací systém pro malé a střední firmy, který bude nabízet řešení bez nutných poplatků s podporou platformy Windows.

Nejprve byl proveden rozbor potřebných vlastností, které jsou na systém kladeny cílovou firmou, pro kterou je systém přednostně vyvíjen. Seznámili jsme se s moderními nástroji, které v dnešní době jsou populární pro monitorování počítačové sítě a porovnali jsme možné použití takového systému s našimi požadavky. Dalším důležitým krokem bylo vybrání vhodných technologií pro implementaci. Důrazně jsme dbali na možnost vývoje systému pro více platforem a zároveň na jednoduchou rozšiřitelnost a použitelnost pro programátora i uživatele.

V další části práce byla popsána implementace jednotlivých projektů celého systému. Popisy obsahují struktury daných projektů s obeznámením jednotlivých sekcí kódu a jeho funkce v systému. Popsali jsme způsob sofistikované implementace správy rutin, které tvoří testování pro zařízení. Uvedli jsme zde i ukázka možného scénáře pro přidání nových testů do celého systému. Následuje rozbor implementace databázového systému s datovým modelem, který elegantně řeší minimalistické řešení a snadnou správu tohoto systému. V závěru této části práce jsou uvedeny ukázky některých částí kódu, které jsou pro systém klíčové. Jejich implementace vyžadovala navrhnutí sofistikované struktury tak, aby implementace byla pro zbytek systému co nejjednodušší a snadno rozšiřitelná.

V následujících kapitolách je popsáno uskutečněné testování aplikace s jejími scénáři a výsledky. Dále je zde zmíněný plánovaný budoucí vývoj a možné rozšíření tohoto systému.

Výsledný monitorovací systém se nachází teprve na počátku svého možného nasazení. Práce na celém projektu však zdaleka nekončí. Aktuální verze pro nás představuje velmi silný základní projekt. Díky sofistikované struktuře můžeme snadno vytvářet další testovací metody pro jeho širší použitelnost. Jako zajímavost může posloužit informace, že celý projekt (projekt Ixs.DNA[27] není počítán) obsahuje 29,500 řádků C# kódu (363 souborů `.cs`) a 4,500 řádků značkového jazyka XAML (36 souborů `.xaml`).

Seznam zkratek

API Application Programming Interface. 14, 25, 27, 43, 45, 75, 77, 78, 88

ENUM Enumerated type. 69, 71, 74

HTTP Hypertext Transfer Protocol. 19, 21, 24, 28, 42, 45, 55, 66, 75, 77, 102

HTTPS Hypertext Transfer Protocol Secure. 102

JSON JavaScript Object Notation. 30, 32, 42, 70, 71, 102

MSSQL Microsoft SQL Server. 69, 102

MVC Model View Controller. 10, 19, 20, 25, 27, 35, 39, 59, 62

MVVM Model View ViewModel. 25, 33, 46, 93

PHP Hypertext Preprocessor. 10, 12, 13, 19

SQL Structured Query Language. 21, 23, 29, 32

URL Uniform Resource Locator. 54

USB Unversal Serial Bus. 3

UWP Universal Windows Platform. 15, 17, 19, 20

WPF Windows Presentation Foundation. 17, 18, 20, 23, 27, 48, 51, 83

XAML Extensible Application Markup Language. 18, 50, 51

XML Extensible Markup Language. 42

Reference

[1] *.NET*. 2020-05-05. URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet.

[2] *.NET Abstract*. 2020-05-05. URL: https://code-maze.com/differences-between-net-framework-net-core-and-net-standard/.

[3] *.NET Abstract iamge*. 2020-05-05. URL: https://kaushalsubedi.com/wp-content/uploads/2018/04/1_-bQofDO6WBkiru3Tu5VpMg.png.

[4] *.NET bundle*. 2020-05-05. URL: https://dotnet.microsoft.com/download.

[5] *.NET Core Bundle*. 2020-05-05. URL: https://dotnet.microsoft.com/download/dotnet-core/thank-you/runtime-aspnetcore-3.1.3-windows-hosting-bundle-installer.

[6] *.NET Standard*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

[7] *.NET Standard- versions*. 2020-05-05. URL: https://dotnet.microsoft.com/platform/dotnet-standard.

[8] *.NET: Ecosystem*. 2020-05-05. URL: https://www.c-sharpcorner.com/article/difference-between-net-framework-and-net-core/.

[9] *3TierArchitecture*. 2020-05-05. URL: https://searchsoftwarequality.techtarget.com/definition/3-tier-application.

[10] *ActiveDirectory*. 2020-05-05. URL: https://searchwindowsserver.techtarget.com/definition/Active-Directory.

[11] Philip Japikse Andrew Troelsen. *Pro C 7: With .NET and .NET Core*. 2017. ISBN: 9781484230176.

[12] *ASP.NET*. 2020-05-05. URL: https://dotnet.microsoft.com/apps/aspnet.

[13] *ASP.NET Razor*. 2020-05-05. URL: https://www.w3schools.com/asp/razor_intro.asp.

[14] *ASP.NET vs ASP Core*. 2020-05-05. URL: https://www.ifourtechnolab.com/blog/differences-between-asp-net-and-asp-net-core-asp-net-vs-asp-net-core.

[15] *Bearer Token*. 2020-05-05. URL: https://swagger.io/docs/specification/authentication/bearer-authentication/.

[16] *Bootstrap*. 2020-05-05. URL: https://getbootstrap.com/.

[17] *BootstrapTable*. 2020-05-05. URL: https://getbootstrap.com/.

[18] *C# Expressions*. 2020-05-05. URL: https://livebook.manning.com/book/c-sharp-in-depth-third-edition/chapter-9/.

[19] *C# Extension Methods*. 2020-05-05. URL: https://www.tutorialsteacher.com/csharp/csharp-extension-method.

[20] *C# Fields*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/fields.

[21] *C# Generics*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/.

[22] *C# Properties*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties.

[23] *COM*. 2020-05-05. URL: https://www.pcmag.com/encyclopedia/term/com-port.

[24] *Cookies*. 2020-05-05. URL: https://kb.iu.edu/d/agwm.

[25] *CSS*. 2020-05-05. URL: https://www.w3.org/Style/CSS/Overview.en.html.

[26] *Dependency Injection*. 2020-05-05. URL: https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/.

[27] *DNA Framework*. 2020-05-05. URL: https://github.com/Frixs/DNA-Framework.

[28] *Electron*. 2020-05-05. URL: https://www.electronjs.org/.

[29] *Electron: The best*. 2020-05-05. URL: https://www.intertech.com/Blog/why-major-companies-are-using-electron-to-build-cross-platform-apps/.

[30] *Entity Framework*. 2020-05-05. URL: https://docs.microsoft.com/en-us/ef/.

[31] *Ethernet*. 2020-05-07. URL: https://searchnetworking.techtarget.com/definition/Ethernet.

[32] Jonas Fagerberg. *Asp.net Core 2.0 Mvc Razor Pages for Beginners*. How to Build a Website. 2017. ISBN: 9781979759953.

[33] Adam Freeman. *Pro ASP.NET Core MVC 2*. 2017. ISBN: 9781484231500.

[34] *HAML*. 2020-05-05. URL: http://haml.info/.

[35] *HTML*. 2020-05-05. URL: https://developer.mozilla.org/en-US/docs/Web/HTML.

[36] *Chart.js*. 2020-05-05. URL: https://www.chartjs.org/.

[37] *Chromium*. 2020-05-05. URL: https://www.chromium.org/.

[38] *I/O Quido*. 2020-05-05. URL: https://papouch.com/io-moduly/quido/.

[39] *IdentityResult class*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identityresult?view=aspnetcore-3.1.

[40] *Ionic*. 2020-05-05. URL: https://ionicframework.com/.

[41] *JavaFX*. 2020-05-05. URL: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm.

[42] *JavaFX: Styles*. 2020-05-05. URL: https://www.callicoder.com/javafx-css-tutorial/.

[43] *Javascript*. 2020-05-05. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[44] *JQuery*. 2020-05-05. URL: https://jquery.com/.

[45] *JSON*. 2020-05-05. URL: https://www.json.org/json-en.html.

[46] *JTI*. 2020-05-05. URL: https://tools.ietf.org/html/rfc7519#page-10.

[47] *JWT*. 2020-05-05. URL: https://tools.ietf.org/html/rfc7519.

[48] *Laravel*. 2020-05-05. URL: https://laravel.com/.

[49] *Microsoft Identity*. 2020-05-05. URL: https://docs.microsoft.com/en-us/microsoft-identity-manager/microsoft-identity-manager-2016.

[50] *Microsoft SQL Server connection strings*. 2020-05-05. URL: https://www.connectionstrings.com/sql-server.

[51] *Mocking*. 2020-05-05. URL: https://github.com/Moq/moq4/wiki/Quickstart.

[52] *MOQ*. 2020-05-05. URL: https://github.com/moq/moq.

[53] *MVC*. 2020-05-05. URL: https://webmail.zcu.cz/SOGo/so/kvetont/Mail/view.

[54] *MVP*. 2020-05-05. URL: https://www.agilealliance.org/glossary/mvp/#q=~(infinite~false~filters~(tags~(~'mvp))~searchTerm~'~sort~false~sortDirection~'asc~page~1).

[55] *MVVM*. 2020-05-05. URL: https://www.raywenderlich.com/34-design-patterns-by-tutorials-mvvm.

[56] *Nagios Free vs Paid*. 2020-05-05. URL: https://www.nagios.com/products/nagios-xi/edition-comparison/.

[57] *Nagios Log tests*. 2020-05-05. URL: https://exchange.nagios.org/directory/Plugins/Log-Files/check_logfiles/details.

[58] *Nagios UPS tests*. 2020-05-05. URL: https://exchange.nagios.org/directory/Plugins/Hardware/UPS/SNMP-UPS-Check/details.

[59] *Nagios: Configuration*. 2020-05-05. URL: https://www.digitalocean.com/community/tutorials/how-to-install-nagios-4-and-monitor-your-servers-on-centos-7.

[60] *Nagios: Popularity*. 2020-05-05. URL: https://www.trustradius.com/it-infrastructure-monitoring.

[61] *Nagios: Pricing*. 2020-05-05. URL: https://www.capterra.com/p/152793/Nagios-XI/.

[62] *Nagios: UI*. 2020-05-05. URL: https://www.nagios.com/products/nagios-xi/.

[63] *NodeJS*. 2020-05-05. URL: https://nodejs.org/en/.

[64] *npm*. 2020-05-05. URL: https://www.npmjs.com/.

[65] *NuGet*. 2020-05-05. URL: https://www.nuget.org/.

[66] *OID*. 2020-05-05. URL: https://www.dpstele.com/snmp/what-does-oid-network-elements.php.

[67] *PHP*. 2020-05-05. URL: https://www.tutorialspoint.com/php/index.htm.

[68] *Ping*. 2020-05-05. URL: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/ping.

[69] *POST (HTTP)*. 2020-05-05. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.

[70] *React.js*. 2020-05-05. URL: https://reactjs.org/.

[71] *RS232*. 2020-05-07. URL: https://circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications.

[72] *SASS*. 2020-05-05. URL: https://sass-lang.com/.

[73] *SNMP*. 2020-05-05. URL: https://www.manageengine.com/network-monitoring/what-is-snmp.html.

[74] *SolarWinds: Comparison*. 2020-05-05. URL: https://www.pcwdld.com/solarwinds-vs-nagios.

[75] *SolarWinds: Features (log)*. 2020-05-05. URL: https://thwack.solarwinds.com/t5/Geek-Speak-Blogs/Logfile-Monitoring-I-Do-Not-Think-That-Word-Means-What-You-Think/ba-p/441691.

[76] *SolarWinds: Features (UPS)*. 2020-05-05. URL: https://thwack.solarwinds.com/t5/Alert-Lab-Discussions/UPS-monitoring-in-Solarwinds/td-p/415821.

[77] *SolarWinds: Pricing*. 2020-05-05. URL: https://www.solarwinds.com/downloads.

[78] *SQL Server*. 2020-05-05. URL: https://www.microsoft.com/cs-cz/sql-server/sql-server-2019.

[79] *SQLite*. 2020-05-05. URL: https://www.sqlitetutorial.net/what-is-sqlite/.

[80] *Symfony*. 2020-05-05. URL: https://symfony.com/.

[81] *UPS*. 2020-05-05. URL: https://www.hardwaresecrets.com/ups/.

[82] *USB*. 2020-05-07. URL: https://computer.howstuffworks.com/usb.htm.

[83] *UWP*. 2020-05-05. URL: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.

[84] *Windows Forms*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/framework/winforms/.

[85] *WPF*. 2020-05-05. URL: https://www.wpf-tutorial.com/about-wpf/what-is-wpf/.

[86] *WPF*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/.

[87] *WPF Attached Properties*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/attached-properties-overview.

[88] *WPF Toolikt*. 2020-05-05. URL: https://github.com/xceedsoftware/wpftoolkit.

[89] *WPF UserControls*. 2020-05-05. URL: https://stackoverflow.com/questions/1379493/what-are-the-purpose-of-user-controls-in-visual-c.

[90] *WPF ValidationRules*. 2020-05-05. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-implement-binding-validation.

[91] *WPF ValueConverters*. 2020-05-05. URL: https://www.c-sharpcorner.com/UploadFile/87b416/wpf-value-converters/.

[92] *xUnit*. 2020-05-05. URL: https://xunit.net/.

[93] *Zabbix Agents*. 2020-05-05. URL: https://www.zabbix.com/zabbix_agent.

[94] *Zabbix: Comparison*. 2020-05-05. URL: https://www.itcentralstation.com/product_reviews/zabbix-review-32935-by-it_user174738.

[95] *Zabbix: Overview*. 2020-05-05. URL: https://www.zabbix.com/features.

[96] *Zabbix: UI*. 2020-05-05. URL: https://www.zabbix.com/documentation/3.0/manual/introduction/whatsnew300.

Přílohy

A Instalační příručka

V této příručce jsou popsány postupy instalace serveru a klientské aplikace tohoto systému.

A.1 Server

Ujistěte se, že na cílovém zařízení máte instalovaný balíček .NET Core (Runtime) verze 3.1. Tento balíček můžete stáhnout na adrese [5] a následně nainstalovat.

Na CD, které bylo v příloze práce, otevřete adresář `Server`.

V tomto adresáři otevřete soubor `appsettings.json`. Zde je potřeba nastavit výchozí nastavení pro připojení k databázi a zabezpečení serveru. Můžete zde vidět strukturu JSON[45], ve které se vyskytuje klíč `DefaultConnection`. Hodnotu tohoto klíče upravte, aby odpovídal standardům zmíněným na webové stránce [50]. Vyplňte jej s vašimi údaji k vytvořené databázi (MSSQL). Jako další klíč se zde nachází `SecretKey`, u kterého změňte hodnotu na Vámi libovolně zvolenou (obsahující 16 nebo více znaků). Jedná se o privátní klíč, který bude reprezentovat Váš server. Posledním volitelným nastavením je klíč `Expires`, kde změnou hodnoty (sekundy) můžete specifikovat délku trvání do odhlášení uživatele z webové části aplikace při jeho neaktivitě.

Druhou částí nastavení je nastavení adresy serveru a jejího portu. Navraťte se do adresáře `Server` a nalezněte složku `Properties` a následně ji otevřete. V této složce nalezněte a otevřete soubor `launchSettings.json`. Obdobným způsobem editujte tento soubor pro klíč `applicationUrl`. Hodnotu tohoto klíče nastavte pro vámi zvolenou adresu serveru ve formátu `https://<adresa>:<port>`. Adresa je nastavená pro komunikaci pomocí HTTPS protokolu. Pokud chcete umožnit přístup pomocí HTTP, definujte dvě adresy oddělené znakem „;"(středník). Druhá adresa bude ve tvaru `http://<adresa>:<port>` s jiným portem (pokud je adresa stejná jako první).

Na závěr můžete server spustit pomocí souboru `ServerRoom.Web.Server.exe` a server je připravený testovat jakékoli zařízení s přidělenou IP adresou v místní síti.

A.2 Klient

Ujistěte se, že na cílovém zařízení máte instalovaný balíček .NET Framework (Runtime) verze 4.8 a .NET Core (Runtime) verze 3.1. Tyto balíčky můžete stáhnout na adrese [4] a následně nainstalovat.

Na CD, které bylo v příloze práce, nalezněte a otevřete adresář `Client`. V tomto adresáři nalezněte soubor soubor `ServerRoom.exe` a spusťte jej.

A.3 Překlad

Pro překlad server a klientské aplikace platí stejný postup. Stáhněte (`visualstudio.microsoft.com`) a nainstalujte vývojové prostředí VisualStudio. Importujte nejdříve klientskou aplikaci pomocí souboru `ServerRoom.sln`, který se nachází na přiloženém CD v adresáři `Source`. Ujistěte se, že jste nezměnili strukturu adresáře `Sources`. Pomocí nástroje VisualStudio přeložte kód do spustitelné verze (`CTRL+SHIFT+B`). Ujistěte se, zda aplikace běží bez případných chyb (v takovém případě se ujistěte, jestli jsou importovány všechny balíčky NuGet[65]). V úspěšném případě můžete přepnout překlad z verze „Debug"na verzi „Release"v konfiguraci řešení a provést překlad znovu. Vaše publikovatelná verze bude dostupná v adresáři projektu `bin/Release`. Stejný postup opakujte pro projekt serveru (`ServerRoom.Web.Server.sln`).

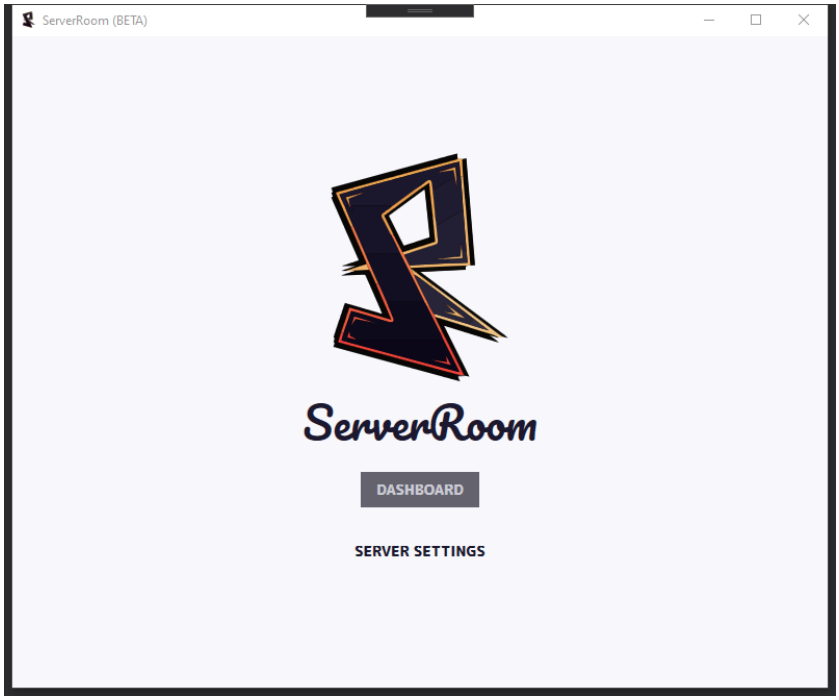
B UŽIVATELSKÁ PŘÍRUČKA

V této příručce je vysvětleno, jak ovládat server a klientskou aplikaci našeho systému.

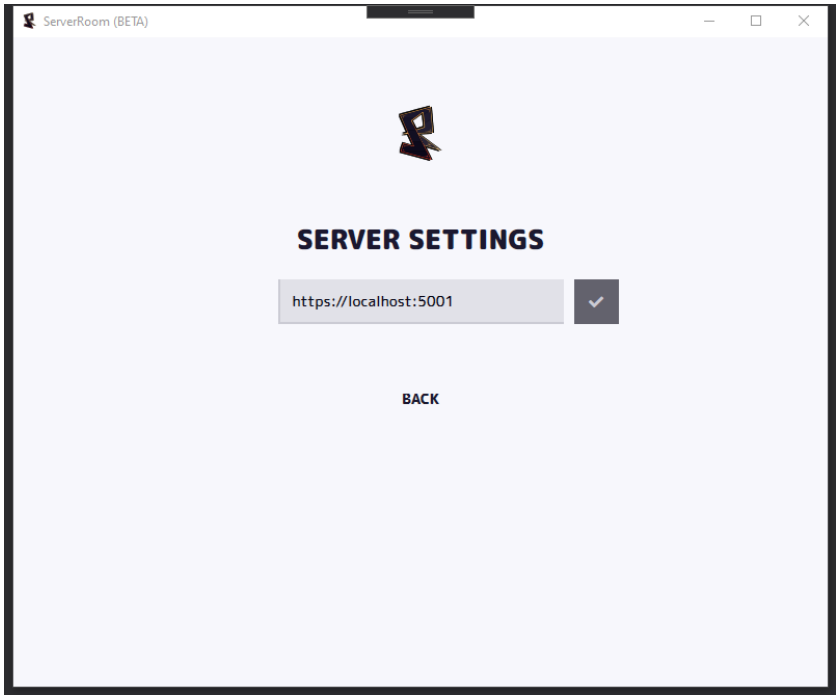
B.1 Nastavení připojení k vašemu serveru (klient)

Jednou z první věcí, která je potřeba provést na serveru je nastavení připojení k serveru. To lze samozřejmě kdykoli v průběhu změnit. Pro změnu připojení k serveru se přesměrujte z hlavní obrazovky klientské aplikace do nastavení serveru (tlačítko `SERVER SETTINGS` na hlavní stránce - obr. 10.1).

Náhled stránky nastavení můžete vidět na obrázku 10.2. Pro možnost editace je zde potřeba nejdříve zastavit připojení k aktuálnímu serveru pomocí tlačítka označeného pomocí „X"(pokud již není zastaveno). Jakmile se server odpojí, je možné editovat uživatelské pole, do kterého vložte adresu vašeho serveru. Pomocí tlačítka odeslání se můžete pokusit o připojení k serveru. Při úspěšném připojení zmizí varovná hláška o chybném připojení k serveru (obr. 10.3) a při nedostupné adrese serveru budete upozorněni pomocí notifikace (obr. 10.4).

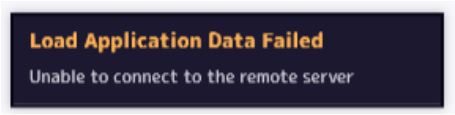


Obrázek 10.1: Hlavní stránka klientské aplikace



Obrázek 10.2: Nastavení připojení serveru

Obrázek 10.3: Upozornění o nedostupném připojení k serveru



Obrázek 10.4: Upozornění o chybném připojení k serveru

B.2 Přihlášení/Odhlášení uživatele

V celém systému existuje pouze jeden uživatel a to je uživatel `admin`. Výchozím heslem pro tohoto uživatele je `Admin123456789`, které doporučujeme po prvním použití změnit.

Při prvním použití našeho systému je potřeba výchozího uživatele vygenerovat. Vygenerovat uživatele můžete pomocí webového rozhraní běžící na serveru. Na výchozí stránce webového rozhraní (obr. 10.5) je viditelné tlačítko `CREATE USER`, které po použití vygeneruje

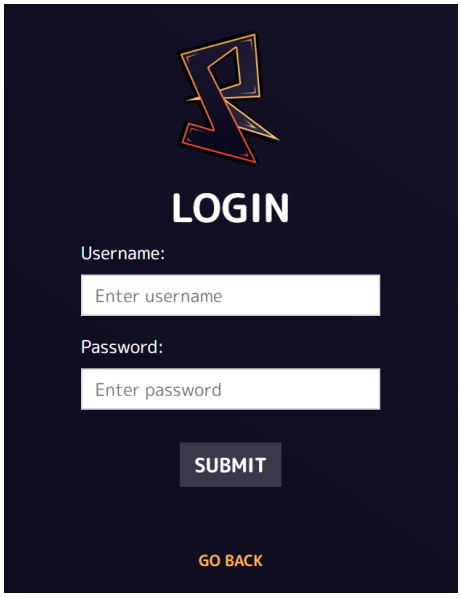
již zmíněného uživatele s názvem `admin`.

Pokud se znovu přesunete do webového rozhraní serveru na výchozí stránku, tak je zde možnost přihlášení (tlačítko `LOGIN`), pomocí kterého se dostanete na stránku přihlašovacího formuláře (obr. 10.6). Po úspěšném přihlášení na uživatele `admin` budete přesměrování do rozhraní „Viewer", které je zobrazením vašich zařízení.

Pro odhlášení uživatele můžete snadno přistoupit k boxu nesoucí jméno přihlášeného uživatele, které je umístěné v pravém horním rohu webové aplikace (obr. 10.7). Pomocí rozbalovacího menu lze vybrat možnost `Logout`, která přihlášeného uživatele řádně odhlásí od systému. Odhlášení v prostředí klientské aplikace lze provést na stránce uživatelského nastavení (obr. 10.8), do které se můžete přesměrovat z ovládacího panelu (obr. 10.10) pomocí tlačítka `LOGOUT`.

Můžete se všimnout, že webové rozhraní kopíruje design klientské aplikace (obr. výchozí stránky klientské aplikace 10.1), které funguje obdobným způsobem jako je tomu na webové části serveru.

Obrázek 10.5: Hlavní stránka webového rozhraní



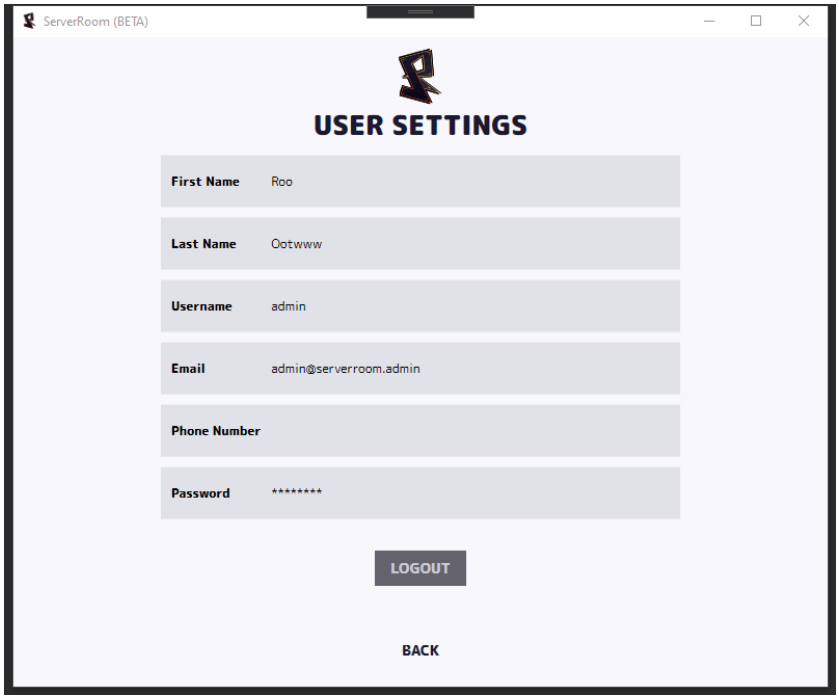
Obrázek 10.6: Přihlašovací formulář webového rozhraní

Obrázek 10.7: Odhlašovací box webového rozhraní

B.3 Správa uživatele (klient)

Pro úpravu uživatelských údajů lze použít stránku klientské aplikace - `USER SETTINGS`. Na tuto stránku se lze dostat prostřednictvím ovládacího panelu (tlačítko v pravém horním rohu - obr. 10.10), na který jste automaticky přesměrováni po přihlášení do klientské aplikace.

Každé pole uživatelských dat je upravitelné nezávisle na ostatních, kde můžete upravit třeba pouze jednu hodnotu a neodesílat celý formulář.

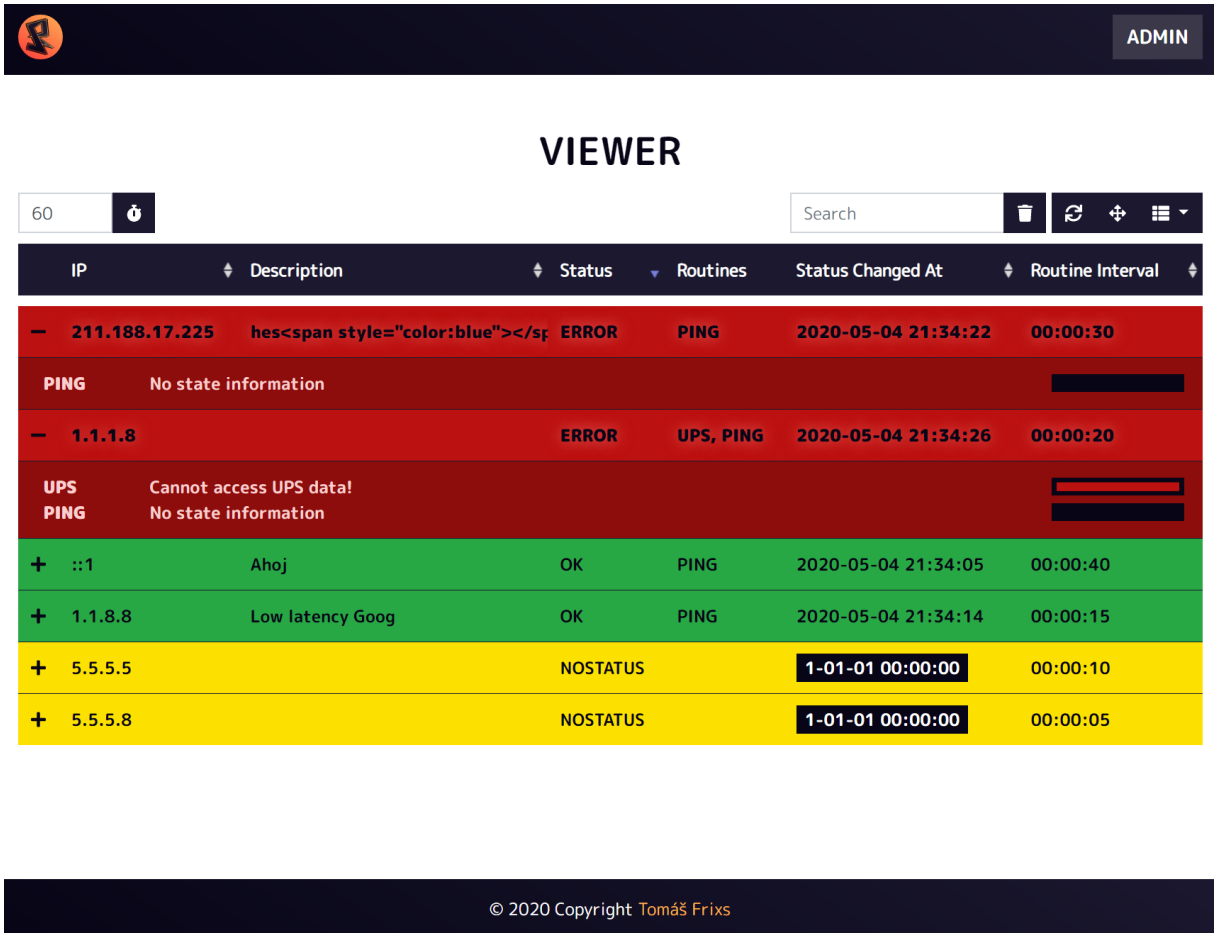


Obrázek 10.8: Uživatelské nastavení

B.4 Obrazovka sledování zařízení (web)

Uživatel přihlášený do webového rozhraní má oprávnění vstoupit na stránku „Viewer", která je projektorem nastavených zařízení a zobrazuje jejich průběh testování. Již od názvu této stránky je zřejmé, že je stránka pouze pro sledování (read-only). Můžete zde vidět průběh všech zařízení včetně sedmi denní historie vyhodnocených testů jednotlivých rutin.

Datová tabulka zobrazující data umožňuje uživateli přizpůsobení na celou obrazovku nebo vybrat jen tak pole, která chce zobrazovat. Dále uživatel může upravovat cyklus aktualizace dat tohoto zobrazení.

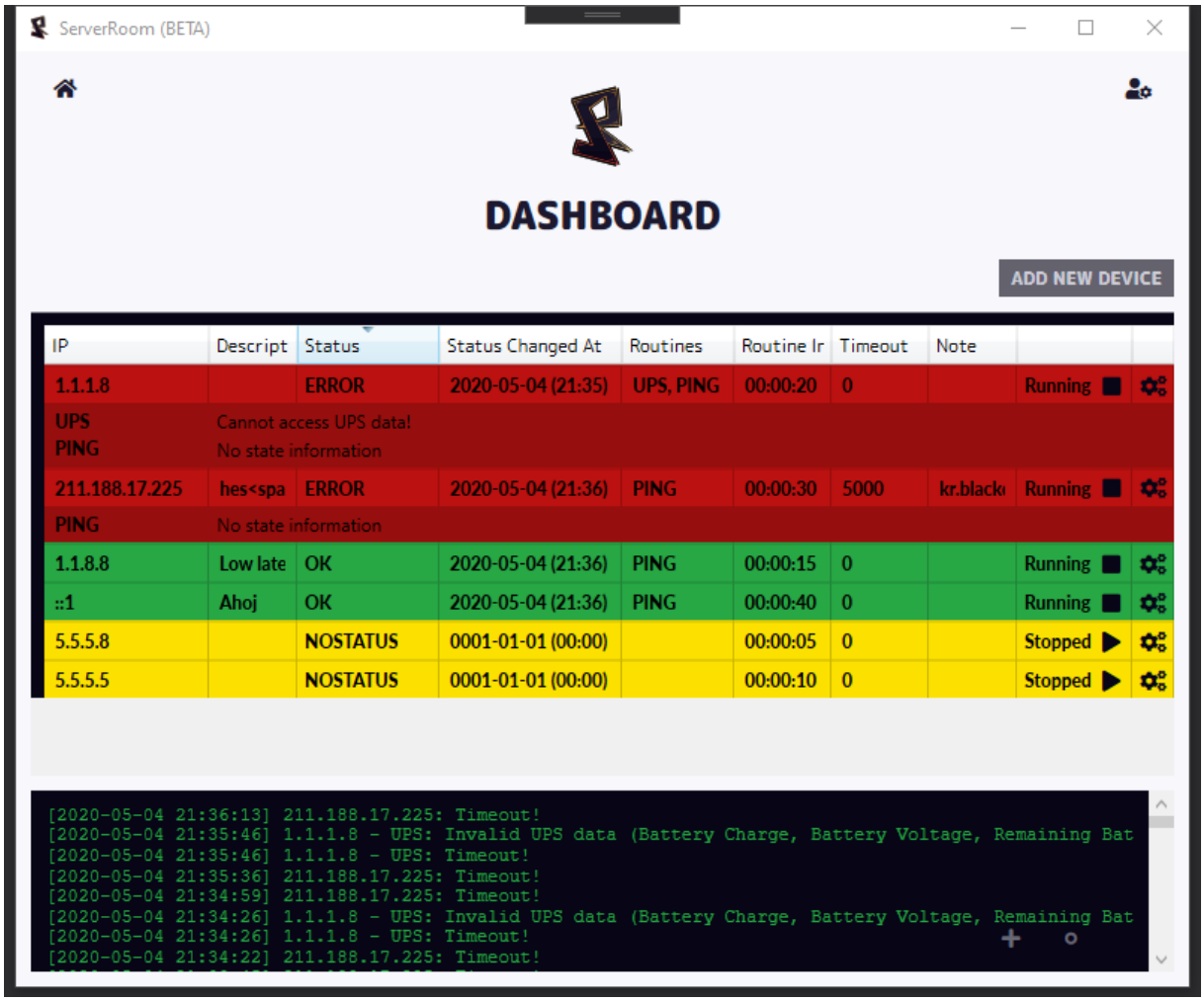


Obrázek 10.9: Viewer

B.5 Ovládací panel (klient)

Ovládací panel slouží pro ovládání celého systému. Hlavní část stránky (obr. 10.10) tvoří seznam všech zařízení včetně jejich rutin. U každého zařízení existují dvě tlačítka: Start/Stop a nastavení (tlačítko ozubeného kolečka). V dolní části můžete vidět uživatelský log, kterým můžete snadněji kontrolovat, co se v systému děje nebo co se v systému stalo.

Pomocí tlačítka Start/Stop můžete zastavit vykonávání testů na daném zařízení nebo je naopak spustit. Pomocí tlačítka nastavení se přesunete do editace daného zařízení. Pokud chcete přidat nové zařízení, stiskněte tlačítko `ADD NEW DEVICE`, které Vás přesune do formuláře pro vytvoření nového zařízení.



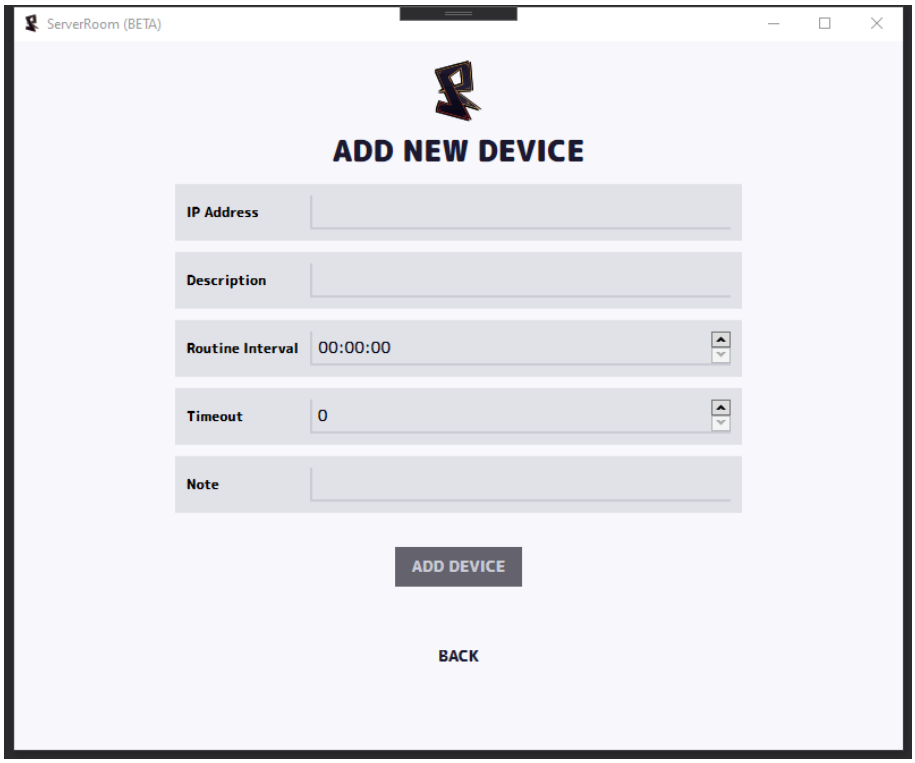
Obrázek 10.10: Ovládací panel klientské aplikace

B.6 Správa zařízení s rutinami

Na obrázku 10.11 můžete vidět formulář pro vytvoření nového zařízení obsahující tato uživatelská pole:

- **IP Address** - IP adresa zařízení.
- **Description** - Volitelný popis zařízení.
- **Routine Interval** - Časový interval, po kterém se pravidelně spouští testy. Jedná se o časovou mezeru, která trvá od konce předchozího testu do začátku dalšího.
- **Timeout** - Definuje maximální dobu, po kterou zařízení čeká na vyhodnocení testu. Po uplynutí této doby se automaticky testy vyhodnotí jako chybné a testování dále pokračuje. Hodnota je zadávána v milisekundách.
- **Note** - Volitelná poznámka pro administrátora.

Přístup k tomuto formuláři naleznete na stránce ovládacího panelu, do kterého se automaticky přesunete po úspěšném přihlášení do klientské aplikace (viz. B.5).

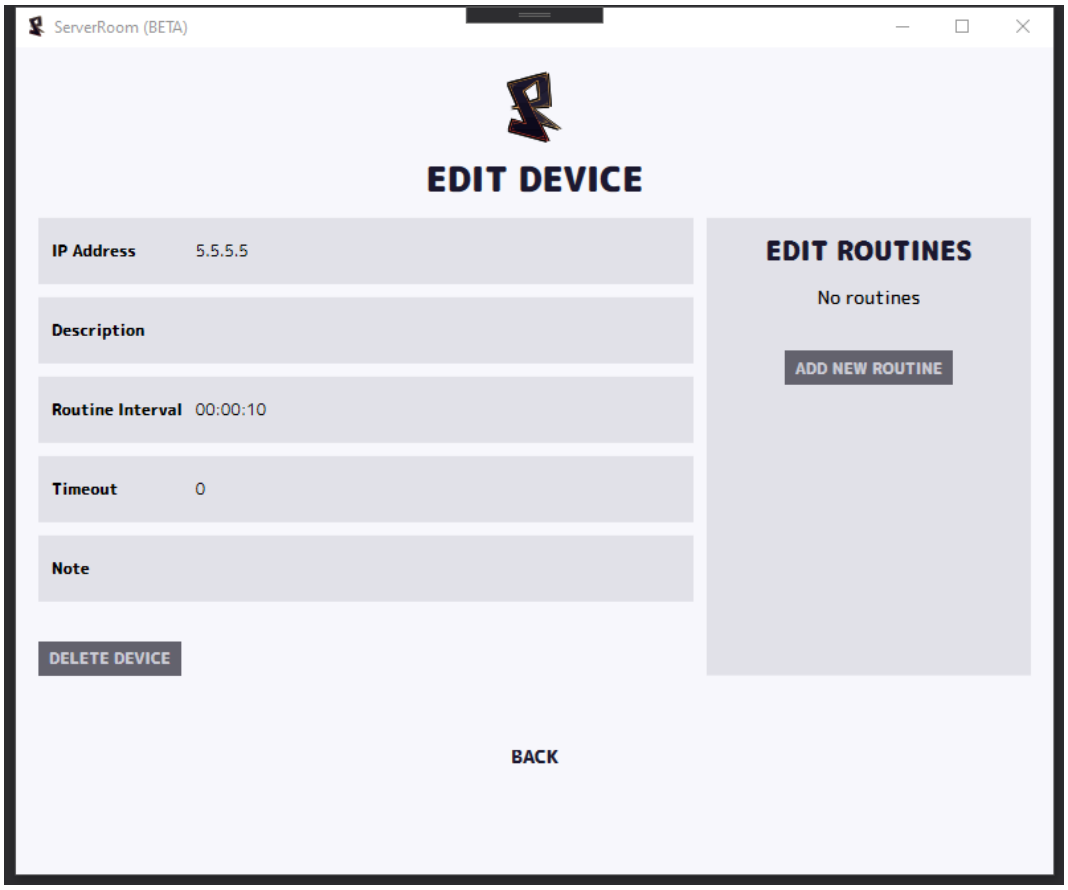


Obrázek 10.11: Formulář pro vytvoření nového zařízení

V případě editace zařízení (obr. 10.12) zde můžete všechny předchozí pole upravit a navíc zde máte možnost vytvoření rutin, které slouží pro testování zařízení. Pomoc tlačítka `ADD NEW ROUTINE` se přesunete do formuláře pro vytvoření nové rutiny pro dané zařízení.

Může existovat pouze tolik rutin zařízení jako je maximální existující počet typů rutin. Rutina, která byla vytvořena dřív bude vykonávána před tou, která byla vytvořena po ní.

Přístup k tomuto formuláři naleznete na stránce ovládacího panelu, do kterého se automaticky přesunete po úspěšném přihlášení do klientské aplikace (viz. B.5).



Obrázek 10.12: Formulář pro editaci existujícího zařízení

V době tvorby nové rutiny (obr. 10.13) máme jako první na výběr typ rutiny. Na základě typu rutiny se dále rozevře konkrétní konfigurace reprezentující daný typ. Na obrázku 10.14 můžete vidět formulář s parametry typu PING. Po vyplnění těchto parametrů je potřeba aplikovat konfiguraci (tlačítko `APPLY CONFIGURATION`). Pomocí tohoto kroku vytvoří formulář přídavná uživatelská pole pro získání externích dat, pokud rutina tuto možnost vyžaduje. V tomto kroku má uživatel možnost použít tlačítko `ADD ROUTINE` pro vytvoření rutiny.

Nachází se zde také uživatelské pole pod názvem „IsTemplate". Pokud je toto pole nastaveno na hodnotu „True", tak dostane uživatel možnost vytvořit pojmenování pro svoji rutinu. Proces vytvoření rutiny je následně již stejný. Hlavním rozdílem je, že se rutina nevytvoří pro zvolené zařízení, ale je nezávislá na jakémkoli zařízení. Tato rutina slouží jako šablona, která může být při příští tvorbě daného typu rutiny použita pro rychlé vložení hodnot.

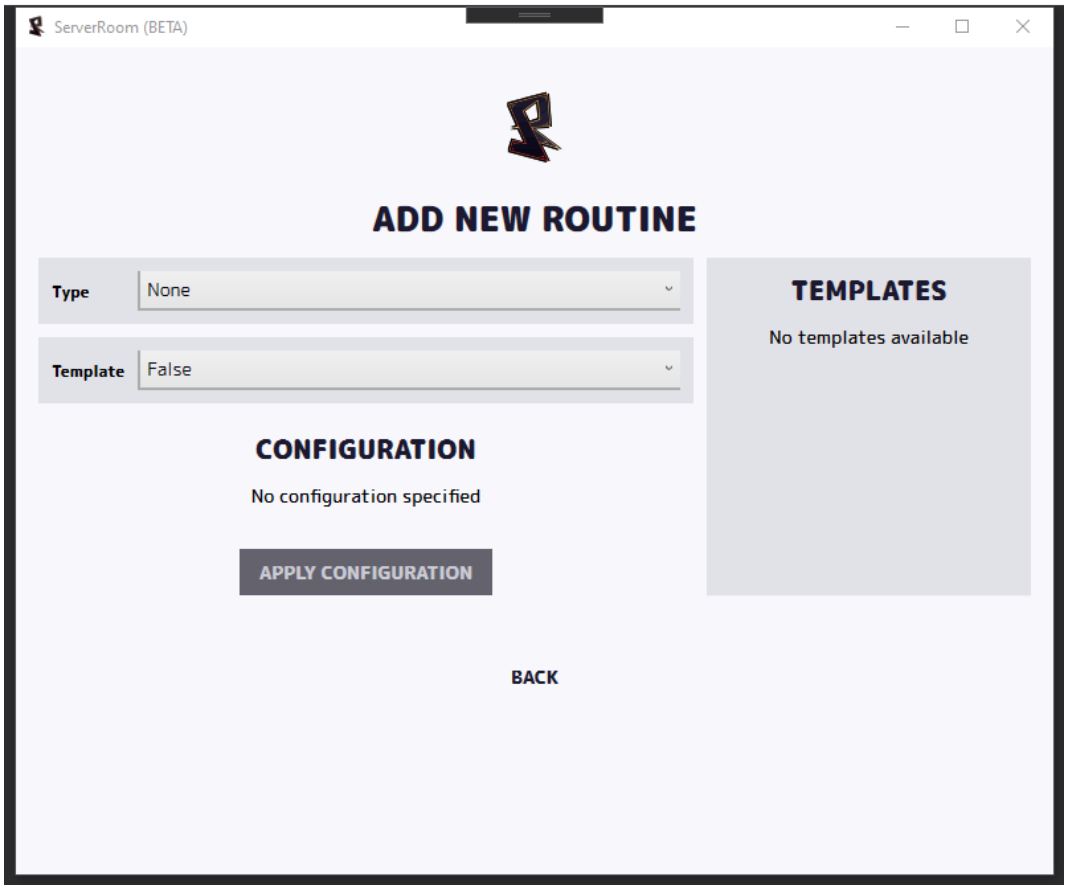
Existují dva typy možných rutin s následujícími uživatelskými vstupy konfigurace:

- **PING**
  - **Multi-Check** - Definuje počet přídavných testování pomocí PING[68]. Pokud první PING selže a je nastavena vyšší hodnota než 0, tak se testuje tolikrát podle nastavené hodnoty, dokud nebude úspěšný alespoň jeden test. Pokud všechny testy selžou, rutina je vyhodnocena jako chyba.
  - **Multi-Check Delay** - Slouží pro nastavení časové mezery od posledního pokusu PING do spuštění dalšího pokusu. Čas se nastavuje v milisekundách.
  - **Timeout** - Pomocí tohoto pole můžete nastavit celkové dovolené časové rozpětí pro vykonání testu této rutiny. Pokud bude čas překročen, rutina se vyhodnotí jako chyba. Čas se nastavuje v milisekundách.
- **UPS**
  - **WarningBracket** - Pokud baterie UPS baterie klesne pod nastavenou hranici, bude test vyhodnocen statusem WARNING (oranžová). Hodnota reprezentuje procentuální hodnotu (0-100).
  - **DangerBracket** - Pokud baterie UPS baterie klesne pod nastavenou hranici, bude test vyhodnocen statusem ERROR (červená). Hodnota reprezentuje procentuální hodnotu (0-100).

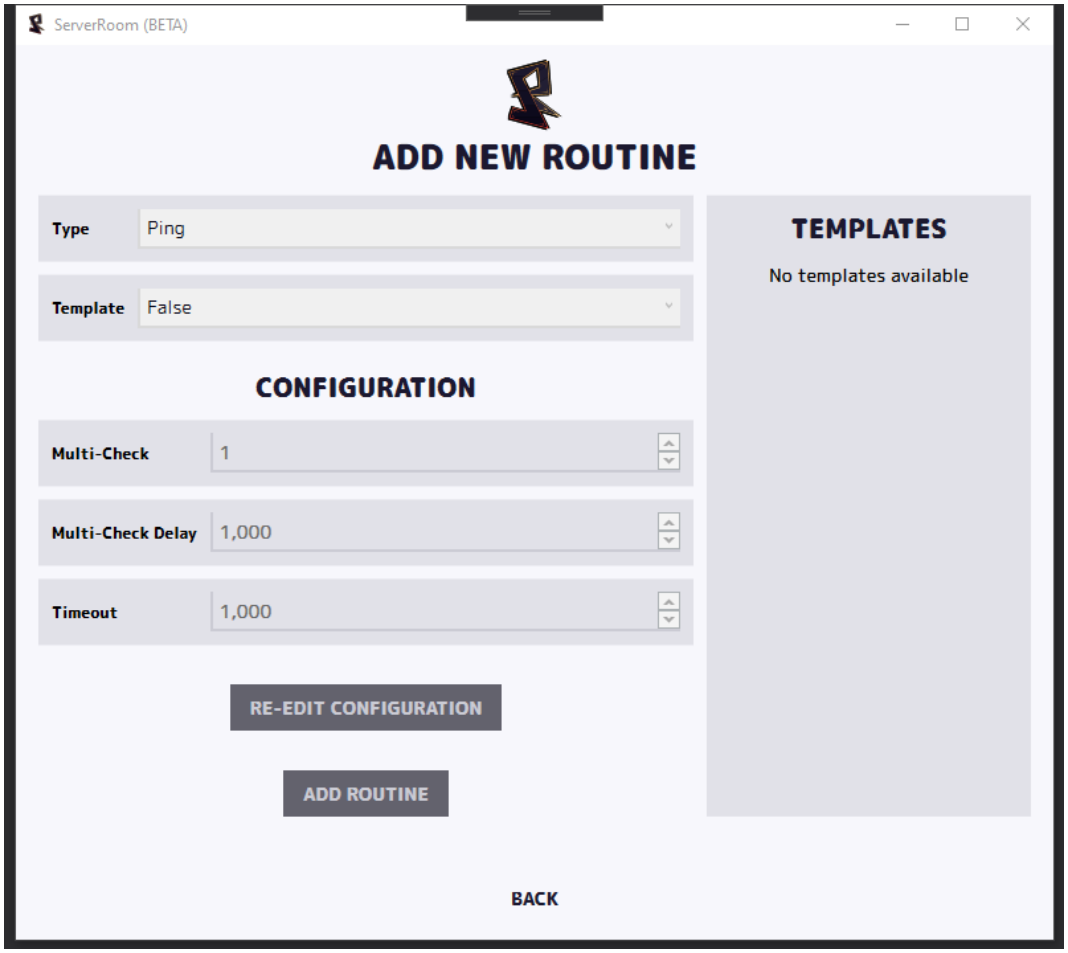
– **WarningBracketInterval** - Specifikuje interval, ve kterém bude uživatel znovu upozorněn (datový log). Pokud se kapacita baterie již nachází v rozmezí **WarningBracket**, tak bude každých X procent uživatel znovu upozorněn.

– **DangerBracketInterval** - Specifikuje interval, ve kterém bude uživatel znovu upozorněn (datový log). Pokud se kapacita baterie již nachází v rozmezí **DangerBracket**, tak bude každých X procent uživatel znovu upozorněn.

– **DataConnector - SNMP** - UPS rutina vyžaduje konekci pomocí „data connectoru"(nabízí možnost SNMP[73]).



Obrázek 10.13: Formulář pro přidání nové rutiny do zařízení



Obrázek 10.14: Formulář pro přidání nové rutiny do zařízení (připraveno pro odeslání požadavku)

C Obsah přiloženého CD

Popis adresářové struktury a obsahu přiloženého CD:

- **Server** - Funkční verze serveru připraveného k nasazení.
- **Client** - Funkční verze klientské aplikace připravené k nasazení.
- **Source** - Adresář obsahující podadresáře jednotlivých projektů obsahující zdrojové kódy celého systému.
- **Doc** - Obsahuje text práce