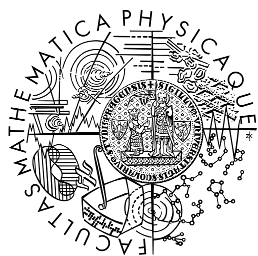


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Vojtěch Kočandrle

Vyhodnocování výkonnosti investičních portfolií

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.
Studijní program: Informatika
Studijní obor: Informatika se specializací Programování a vývoj software

Praha 2023

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

Děkuji panu RNDr. Filipu Zavoralovi, Ph.D za vedení této práce a cenné rady a připomínky, které mi během jejího psaní poskytl. Dále bych rád poděkoval své rodině za podporu během celého studia.

Název práce: Vyhodnocování výkonnosti investičních portfolií

Autor: Vojtěch Kočandrle

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D., Katedra softwarového inženýrství

Abstrakt: Mnoho lidí investuje své peníze do nejrůznějších aktiv a snaží se dosáhnout těmito aktivitami zisku. Potřebují proto sledovat, jak se jim investování daří, k čemuž slouží aplikace pro vyhodnocování výkonnosti investičních portfolií. Většina stávajících řešení obsahuje jen limitované množství funkcí obzvláště v oblasti zobrazování grafů a nedávají uživateli možnost rozhodovat, kde se nachází data nebo odkud se data čerpají. V práci se proto zaměříme na vytvoření konkurenční aplikace, která bude rozšiřitelná pomocí zásuvných modulů a zároveň jednoduchá na používání s možností vytváření grafů a jejich uživatelsky definovaným rozvržením v rámci nástěnky. Jednoduchost aplikace bude také v tom, že i přes její rozšiřitelnost nebude potřeba procházet žádné konfigurační soubory. Všechna nastavení budou proveditelná přímo z grafického uživatelského rozhraní.

Klíčová slova: investice, zásuvné moduly, výkonnost

Title: Portfolio performance evaluation

Author: Vojtěch Kočandrle

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: Many people invest their money in various assets in order to produce gain. Therefore, they need to monitor how their investments are doing. There are specialized applications for this purpose. However, most of the existing solutions contains only a limited amount of features, especially when it comes to displaying graphs. Users also usually can not decide where data is stored and retrieved from. In out work, we will therefore focus on creating a competitive application that will be expandable using plugins and at the same time simple to use with possiblity of creating graphs and their user-definined layout within the dashboard. The simplicity of the application will also mean that, despite its extensibility, there will be no need to edit any configuration files. All settings will be configurable directly from the graphical user interface.

Keywords: investment, plugins, performance

Obsah

1 Úvod **3**
1.1 Cíl práce 3

2 Existující produkty **5**
2.1 Yahoo! Finance 5
2.2 Sharesight 5
2.3 Kubera 6
2.4 Simply Wall St 6
2.5 StockMarketEye 7
2.6 Delta Investment Tracker 8
2.7 Další produkty 8

3 Analýza **9**
3.1 Platforma 9
3.1.1 Web 9
3.1.2 Hybridní aplikace pro web a desktop 9
3.1.3 Desktop 10
3.2 Programovací jazyk a knihovna grafického uživatelského rozhraní . 10
3.2.1 C++ 10
3.2.2 Java 11
3.2.3 C# 11
3.3 Knihovny pro zobrazování grafů 13
3.3.1 LiveCharts 13
3.3.2 OxyPlot 13
3.4 Práce s daty 14
3.4.1 Úložiště dat 14
3.4.2 Reprezentace dat v aplikaci 15
3.4.3 Datový model 17
3.4.4 Reprezentace časových položek 22
3.5 Stahování dat z veřejných API 22
3.5.1 Konfigurace zásuvného modulu 23
3.5.2 Parametry úlohy 23
3.5.3 Proces zpracování úlohy 24
3.5.4 Import instrumentů 26

4 Implementace **27**
4.1 Struktura aplikace 27
4.2 Uživatelské rozhraní 27
4.2.1 Obrazovky obsahující neuložená data 28
4.2.2 Možnost otevření více oken 29
4.2.3 Dialogová okna 29
4.3 Komponenty uživatelského rozhraní 30
4.3.1 Tabulka s vnořenými tabulkami 30
4.3.2 Widgety 30
4.3.3 Deaktivovatelná skupina komponent 32

4.3.4 Horizontální seznam zaškrtávacích políček 32
4.4 Zobrazování dat v tabulkách 33
4.5 Zobrazování dat v grafech 34
4.6 Stahování dat na pozadí 34
4.7 Synchronizace práce s daty ve více vláknech 35
4.8 Aktualizace zobrazovaných dat 36
4.9 Výpočet výkonnosti portfolií a pozic 36
4.10 Obrana proti zamrznutí grafického uživatelského rozhraní při načítání dat 37
4.11 Změna referenční měny 38
4.12 Zásuvné moduly 39
4.12.1 Zásuvný modul pro práci s úložištěm 39
4.12.2 Zásuvný modul pro stahování dat 40

5 Uživatelské rozhraní 42
5.1 První spuštění 42
5.2 Časté společné znaky různých obrazovek 42
5.3 Měny 43
5.4 Trhy 43
5.5 Instrumenty 46
5.6 Portfolia 47
5.7 Skupiny obchodních prázdnin 50
5.8 Nastavení 50
5.9 Automatické stahování dat 52
5.10 Grafy 53
5.11 Nástěnka 55
5.12 Možnost zobrazení více oken 58

6 Otestování výkonu aplikace 59
6.1 Test rychlosti výpočtu IRR výkonnosti 59
6.1.1 Generování testovacích dat 59
6.1.2 Testovací prostředí 59
6.1.3 Analýza výsledků testování pozice 60
6.1.4 Analýza výsledků testování portfolia 60

Závěr 62

Seznam použité literatury 63

Seznam obrázků 64

A Přílohy 65
A.1 Instalace aplikace 65
A.1.1 Požadavky na systém 65
A.1.2 Sestavení 65
A.1.3 Využití již hotového sestavení 65
A.1.4 Přidávání zásuvných modulů a konfigurace 65
A.2 Data získaná při měření rychlosti výpočtu IRR výkonnosti 66
A.3 Adresářová struktura elektronické přílohy 68

1. Úvod

Investování do akcií, komodit, indexů či jiných druhů investic se dnes věnuje mnoho lidí. Dle průzkumu provedeného centrální bankou USA v roce 2019 15,2 % amerických rodin přímo vlastnilo akcie, 9 % rodin vlastnilo podíl ve sdílených investičních fondech.[1] O investice je zájem a i menší investor potřebuje aplikaci, která by mu pomohla sledovat vývoj instrumentů (akcie, komodita, index, kurzovní lístek), které ho zajímají, a umožnila sdružovat je do portfolií. Takovýchto aplikací pochopitelně existuje celá řada, ale mají obvykle řadu nevýhod, které mohou menšího investora odradit od jejich používání.

První z nich je často cena. Menší investor typicky nechce větší část svého zisku využít k zaplacení software, který mu umožní sledovat jeho vývoj. Levnější varianty se pak pohybují v řádech desítek amerických dolarů, ale bývají značně omezené například množstvím instrumentů, které je možné sledovat.

Dalším častým společným znakem je umístění na webové platformě. To přináší mnoho výhod. Za zmínku určitě stojí možnost přístupu z kteréhokoliv zařízení s přístupem na internet prakticky odkudkoliv. Zároveň má tato varianta značnou vadu v tom, že uživatel nemá žádnou kontrolu nad svými daty a to nejen z pohledu bezpečnosti. V případě, že by se poskytovatel služby rozhodl její provozování ukončit, je reálné, že uživatel o svá data přijde nebo si je bude muset včas opsat.

Webové aplikace také obvykle nebývají nijak rozšiřitelné a neumožňují uživateli získávat hodnoty instrumentů z jiných než předdefinovaných zdrojů, protože poskytovatelé těchto služeb typicky nechtějí, aby si uživatel do jejich systému nahrával další rozšíření, která by posléze mohla ovlivnit i ostatní uživatele. Navíc jde o potenciální bezpečnostní riziko. Toto má však za následek, že v případě, že se instrument v seznamu dostupných instrumentů nenachází, uživatel ho nemůže analyzovat.

1.1 Cíl práce

Cílem této bakalářské práce tedy je navrhnout a implementovat aplikaci sloužící pro vyhodnocování výkonnosti investičních portfolií běžící přímo na počítači koncového uživatele, která bude dostatečně modifikovatelná a zároveň jednoduchá na používání s uživatelsky přívětivým grafickým rozhraním.

Aplikace by měla umožnit uživateli mít kontrolu nad svými daty, k čemuž patří i možnost zvolit si, jakým způsobem a kde budou data uložena, a základní analýzu vytvořených portfolií, tedy sledovat aktuální stav zisku a výkonnosti jednotlivých portfolií a jejich historický vývoj prostřednictvím uživatelem definovaných grafů. Zadávání dat do aplikace by mělo být možné provádět automaticky stahováním z veřejně dostupných API (Application Programming Interface), ale i manuálním zadáváním hodnot pro jednotlivé časové body v případě, že se daný instrument nenachází v žádném z podporovaných veřejných API.

Aby bylo pro uživatele snadné přidat si do aplikace možnost stahování dat z nového podporovaného zdroje, mělo by být stahování dat implementované pomocí systému zásuvných modulů. Jelikož je důležité, aby bylo používání těchto zásuvných modulů pro uživatele co možná nejjednodušší, musí být po nahrání zásuvného modulu veškerá jeho konfigurace proveditelná prostřednictvím grafického

rozhraní.

Důležitou součástí aplikace je již zmíněná možnost zobrazení údajů pomocí grafů. Ty by mělo být možné vytvářet pomocí jednoduchého editoru v grafickém rozhraní, který umožní nakonfigurovat vše od typu grafu a jeho parametrů až po jednotlivé datové řady. Zobrazování grafů by mělo podporovat dvě varianty: zobrazení samotného grafu na celou obrazovku a zobrazení několika grafů v uživatelsky definovaném rozvržení.

2. Existující produkty

Produktů zabývajících se sledováním investic je na trhu celá řada. V této kapitole zmíníme několik z nich, které se často používají, a rozebereme jejich přednosti a nevýhody.

2.1 Yahoo! Finance

Yahoo! Finance [2] je jednou z mnoha služeb poskytovaných webovým vyhledávačem a poskytovatelem emailových schránek Yahoo! Služba je nabízena na webu zcela zdarma a poskytuje aktuální a historické hodnoty velkého množství instrumentů ze světových trhů. Výhodou pro české investory může být, že obsahuje i instrumenty obchodované na Pražské burze.

Kromě hodnot jednotlivých instrumentů nabízí Yahoo! Finance také články týkající se jednotlivých instrumentů, informace o společnostech a možnost zobrazit data v několika typech grafů, mezi kterými nechybí spojnicové a svícové grafy. Přestože služba umožňuje zobrazit hodnoty velkého množství instrumentů, ostatní informace a články jsou k dispozici spíše u větších nebo amerických instrumentů. V případě těch českých služba obsahuje typicky článků jen velice málo nebo žádné a všechny metriky nejsou počítány.

Kromě toho Služba Yahoo! Finance nabízí možnost evidence svého portfolia. Pro použití této služby je nutné přihlášení, které lze provést pomocí účtu Yahoo nebo externího přihlášení, mimo jiné, pomocí účtu Google. Po přihlášení je možné začít přidávat do portfolia instrumenty a transakce. Služba umožňuje vytvářet si i další portfolia. Portfolia mohou být také importována propojením s uživatelovým účtem u makléře.

Hlavní zobrazovací metrikou portfolií je jejich aktuální tržní hodnota a její změny. Portfolio umožňuje vytvářet tabulky s uživatelem vybranými sloupci. Mezi podporované metriky patří kromě průměrů předpokládaný zisk z dividend a metrika EPS TTM (Earnings Per Share for Trailing Twelve Months)[1]. Všechny tyto metriky však platí pro instrument obecně bez ohledu na údaje zadané v portfoliu.

Pro portfolio se zobrazuje jediný graf, který by měl zobrazovat roční výkonnost. V době psaní tohoto textu jde však pouze o miniaturu, kdy po kliknutí na ni je uživatel přesměrován na hlavní stránku Yahoo! Finance.

2.2 Sharesight

Sharesight [3] je webovou aplikací, která má variantu zdarma i několik placených variant. Mezi funkce této aplikace patří možnost přidávat si instrumenty do portfolia a sledovat jejich tržní hodnotu. V nabídce jsou instrumenty z více než 40 trhů, mezi které ale nepatří Pražská burza. V případě, že potřebujete zadat instrument, který v nabídce není, můžete si vytvořit vlastní instrument a jeho cenu v čase zadávat manuálně. Data je opět možné importovat z účtu u makléře. Lze také provádět automatický import provedených transakcí pomocí příjmu emailových zpráv od makléře potvrzujících provedení transakce.

[1] Zisk z jednotky podílu za posledních 12 měsíců

Obrazovka instrumentu umožňuje zobrazit graf jeho vývoje a zprávy týkající se instrumentu. Na stránce portfolia se zobrazuje graf s hodnotou instrumentů sdružených podle volitelného parametru nebo konfigurovatelné skupiny. Zobrazit v grafu jednotlivé instrumenty není možné.

Ve variantě zdarma aplikace umožňuje vytvoření pouze jednoho portfolia s maximálně 10 instrumenty a jednu skupinu pro sdružování instrumentů v grafu portfolia. Placené verze pak snižují či odebírají limit počtu instrumentů a přidávají další možnosti analýzy. Maximální počet portfolií nikdy nemůže přesáhnout 10 a skupin lze vytvořit maximálně 5.

2.3 Kubera

Kubera [4] je placená webová aplikace na bázi ročního předplatného se 14denní zkušební dobou umožňující sledování portfolia, která na rozdíl od většiny ostatních umožňuje sledovat nejen instrumenty, ale také dluhy například z kreditních karet a hodnotu dalšího majetku, jako jsou domy a byty, auta nebo domény. V případě bydlení v USA je aplikace schopná zjistit u služby Zillow odhadovanou tržní hodnotu. Podobně je aplikace schopna odhadnout hodnotu u amerických a kanadských aut pomocí identifikačního čísla vozidla i u domén. Kromě už zmíněných položek je ale možné přidat cokoliv s manuálním zadáním tržní hodnoty. Data lze propojit s účtem u makléře či peněženkou na kryptoměny.

Grafické rozhraní je velice čisté a přehledné. Veškeré položky se dají dělit do záložek a v rámci záložky ještě do skupin. I přesto ale začátky v aplikaci nejsou příliš intuitivní, protože chování je v některých ohledech úplně jiné, než u ostatních aplikací. Kubera totiž při přidání instrumentu sice načte jeho aktuální cenu, ale nenačítá jeho historii. Aplikace funguje tak, že uživatel přidá instrument a kolik ho aktuálně vlastní. Z toho se spočítá hodnota jeho investice, která se zapíše s aktuálním datem do historie. Každý den se pak do historie přidá další záznam vypočtený jako počet jednotek, které uživatel vlastní, vynásobený aktuální cenou jedné jednotky. Tímto způsobem se portfolio začne plnit daty. Trvá tedy pár dní, než obsahuje dostatek dat na to, aby se začaly zobrazovat grafy. Starší data ale mohou být vyplněna manuálně.

Mezi možnosti analýzy portfolia patří graf čistého zisku a rozložení investic mezi jednotlivé položky. Pro aktiva je také možné zobrazit graf vývoje tržní hodnoty vlastněného množství. Pro všechny druhy aktiv[2] může uživatel vyplnit tabulku toku financí. Na jejím základě aplikace počítá celkové množství vložených a vybraných financí a metriku IRR (Internal Rate of Return)[5], která se porovnává s IRR indexů S&P 500, Dow Jones a Nasdaq Composite, kryptoměn Bitcoin a Ether a společnostmi Apple, Alphabet[3] a Tesla. Porovnání s jinými indexy, kryptoměnami či společnostmi není možné.

2.4 Simply Wall St

Simply Wall St [6] je také webová aplikace, která nabízí variantu zdarma i varianty s měsíčním nebo ročním předplatným. Od předchozích webových apli-

[2] Akcie a vše další, co uživatel vlastní.

[3] Společnost vlastnící vyhledávač Google

kací se liší především množstvím informací o jednotlivých instrumentech a tím, jak moc zachází do detailu, a to nejen u amerických instrumentů, ale i u těch českých.

O společnosti se zde uživatel dozví nejen historický vývoj ceny, ale také analýzu toho, jak adekvátní je její ohodnocení ve srovnání se zbytkem trhu, odhad budoucího vývoje, informace o tom, jak se společnosti vedlo za posledních 5 let, o jejím hospodaření a vyplácení dividend. Nechybí ani seznam členů vedení a představenstva společnosti včetně jejich životopisů. Instrumenty je možné také porovnávat, při čemž se tyto analýzy otevřou ve sloupcích vedle sebe. Analýzy instrumentů může uživatel, který využívá službu zdarma, použít pro 5 instrumentů měsíčně. V případě placených variant je to pak 30 instrumentů měsíčně nebo neomezeně.

Kromě podrobných analýz jednotlivých instrumentů umožňuje aplikace vytvářet portfolia a pro každý instrument zadávat provedené transakce. Podobně jako pro jednotlivé instrumenty je i zde k dispozici podrobná analýza. Ta obsahuje obvyklé shrnutí tržní hodnoty, ale také analýzu diverzifikace portfolia, jeho ohodnocení na základě poměrů PE (Price-Earnings), PEG (Price-Earnings to Growth) a PB (Price-to-Book), shrnutí posledních 5 let pomocí metrik ROE (Return on Equity), ROA (Return on Assets) a ROCE (Return on Capital Employed) a informace o dividendách vyplácených instrumenty v portfoliu. Chybí zde ale možnost zobrazit vývoj zmíněných hodnot v grafu, či zobrazit graf s vývojem více instrumentů.

2.5 StockMarketEye

StockMarketEye [7] je aplikace pro počítače s operačními systémy Windows, macOS a Linux. K dispozici jsou také mobilní aplikace pro systémy Android a iOS. Aplikace má bezplatnou zkušební verzi na 30 dní a následně je placená na bázi ročního předplatného. Data jsou čerpána z Yahoo! Finance a MSN Money. Aplikace umožňuje vytvářet neomezené množství portfolií a skupin portfolii. Instrumenty, se kterými uživatel zrovna neobchoduje, si může přidat mezi sledované a mít je tak také zobrazené v tabulkách. Grafické rozhraní je přehledně rozdělené na levou část se stromem portfolií a seznamů sledovaných instrumentů a pravou část s tabulkou v horní části a grafem v dolní části. Zobrazit více grafů je možné pomocí samostatných oken.

Aplikace těží z velkého množství dat především na straně Yahoo! Finance, ale neumožňuje přidat vlastní instrumenty, které by neodpovídaly ani jednomu ze zmíněných zdrojů, a manuálně zadávat jejich hodnotu v čase. Také grafové zobrazení je v aplikaci zjevně inspirováno právě Yahoo! Finance, kdy graf, jeho ovládání a zobrazování popisných údajů vypadá prakticky stejně a má podobné funkce.

Pro portfolia má aplikace zvláštní část pro analýzu, ve které kromě tržní hodnoty portfolia a jeho částí v různých časových okamžicích zobrazuje také metriky IRR (Internal Rate of Return) a TWRR (Time-Weighted Return Rate) a jejich vývoj.

Data aplikace zůstávají uložena pouze v počítači uživatele. V případě, že chce uživatel sdílet data na více zařízeních, může se zaregistrovat u vývojáře aplikace a prostřednictvím tohoto účtu provádět synchronizaci.

2.6 Delta Investment Tracker

Delta Investment Tracker [8] je aplikace pro mobilní telefony se systémy Android a iOS, která je ke stažení zdarma a umožňuje kromě prohlížení aktuálních cen instrumentů, NFT a kurzů měn také vytvářet portfolia. Aby mohl uživatel začít aplikaci používat, není nutná registrace a veškeré informace o portfoliu zůstávají v úložišti telefonu. V případě potřeby lze sdílet data mezi několika zařízeními. Transakce je do portfolia možné importovat propojením s účtem u uživatelova makléře nebo manuálně. Manuální zadávání je ale na mobilním zařízení zdlouhavé a nehodí se pro zadávání většího množství transakcí. Data mohou být importována přímo z účtu u makléře nebo ze souborů ve formátech QIF (Quicken Interchange Format), OFX (Open Financial Exchange) a CSV (Comma-separated values).

V aplikaci jsou k dispozici instrumenty z řady trhů, mezi které ale nepatří Pražská burza. U jednotlivých instrumentů lze sledovat vývoj ceny, číst články, zobrazit provedené transakce s tímto instrumentem a nastavovat upozornění na zásadnější změny ceny instrumentu.

Základní varianta aplikace je zdarma, ale možnost provádět analýzu portfolia má uživatel až v placené „PRO“ verzi. Ta také zvyšuje počet zařízení, přes která je možné data synchronizovat, a počet možných připojení k účtům u makléřů nebo peněženkám. Analýza portfolia v bezplatné verzi je omezena na zobrazení aktuální tržní hodnoty portfolia a její změnu.

2.7 Další produkty

Existuje pochopitelně i řada dalších produktů, které typicky nabízí různé kombinace funkcí výše zmíněných produktů. Některé ale není možné používat celosvětově. Mezi tyto produkty patří webová aplikace Personal Capital [9] s více než 3,3 miliony uživatelů, která je k dispozici zcela zdarma a často hodnocena jako vůbec nejlepší aplikace pro sledování portfolií. Její zásadní vadou však je, že při registraci vyžaduje americké telefonní číslo.

3. Analýza

V této kapitole provedeme analýzu jednotlivých problémů, které je potřeba vyřešit před zahájením implementace samotné aplikace. Postupně zvolíme platformu, programovací jazyk, knihovny a způsob interpretace a ukládání dat.

3.1 Platforma

Během analýzy jsme zvažovali tři možné kandidáty na platformu: web, kombinace webu a desktopu a desktop. Jako možnost se nabízela také mobilní platforma, ale byla považována spíše za podporu některé z předchozích variant kvůli limitované velikosti obrazovky. V následujících podkapitolách představíme zmíněné kandidáty a jejich přednosti i nevýhody.

3.1.1 Web

Velkou výhodou webových aplikací je, že fungují na všech operačních systémech bez toho, aby pro ně vývojář explicitně přidával podporu. To má ovšem také své nevýhody, především pokud jde o mobilní zařízení, která mají obvykle výrazně menší obrazovku. Webová aplikace by tedy vyžadovala, aby se grafické rozhraní dobře používalo na velkých obrazovkách desktopu i na malých obrazovkách mobilních telefonů. U některých aplikací to není velký problém, ale v případě sledování investičních portfolií potřebujete mít na obrazovce větší množství grafů, které by na mobilních telefonech byly příliš malé. Grafické rozhraní by bylo tedy nutné rozdělit na mobilní a desktopovou variantu, což by v případě snahy o zachování všech funkcí i v mobilní verzi výrazně komplikovalo vývoj.

Kromě složitějšího grafického rozhraní může být u webové aplikace problém také s výkonem. Opět zde jde hlavně o grafy, které typicky obsahují velké množství bodů. Výpočet jednotlivých bodů by bylo možné přenechat databázovému serveru a tím snížit nároky na výkon klienta. Stále zde ale zůstává problém s výkonností vykreslování. Tento problém zjevně řešil i web Yahoo Finance [2], kde problém vyřešili tak, že na mobilních zařízeních odebrali možnost zobrazit graf.

Aby bylo pro uživatele jednoduché webovou aplikaci používat, musel by být odcloněn od jejího nasazení, které se skládá hned z několika kroků a zahrnuje konfigurace samotného hostingu a databázového serveru. To ale v zásadě znamená, že by aplikace musela být poskytována jako hotová služba. V takovém případě by nebylo možné, aby si každý uživatel přidával vlastní zásuvné moduly pro stahování dat a nemohl by tedy být naplněn jeden z našich cílů, umožnit uživateli, aby si zvolil zdroj dat.

3.1.2 Hybridní aplikace pro web a desktop

Hybridní aplikace, která by byla schopná běžet na webu i na desktopu se zdá jako hezký kompromis, protože je pak na uživateli, jestli chce používat webovou nebo desktopovou variantu. Tato možnost se z počátku zdála jako nejlepší, ale v rámci důkladnější analýzy se ukázalo, že má množství nevýhod.

Aby tato varianta mohla fungovat, muselo by její jádro být vytvořené v některé z webových technologií. V praxi by to tedy vypadalo tak, že bychom měli jádro i části specifické pro webové a desktopové rozhraní psané v jazyce JavaScript. Vše by se tedy lišilo jen tím, zda se aplikace zobrazuje v standardním webovém prohlížeči nebo v aplikaci vytvořené například pomocí ElectronJS [10].

Jelikož jde stále vlastně o webovou aplikaci, i ve své desktopové variantě si nese všechny problémy webu zmíněné výše v části 3.1.1. Rozdílem je, že zde bychom v rámci desktopové varianty nutně nepotřebovali samostatný databázový server, ale mohli bychom využít lokální databáze.

3.1.3 Desktop

Oproti webu má aplikace pro desktop několik výhod. Předně má i na slabších počítačích k dispozici vyšší výkon než webová aplikace, kterou typicky zpomaluje interpretovaný jazyk. Další výhodou je výrazně snazší zprovoznění, protože není potřeba konfigurovat webový server. V případě použití lokální databáze odpadá také latence způsobená posíláním dat po sítí. Použití lokální databáze navíc zvyšuje bezpečnost, protože spíše dojde k napadení veřejné služby vystavené na internetu, než souboru na lokálním disku. Není zde ani problém s rozdílností grafického rozhraní pro různé typy zařízení.

Pochopitelně jsou zde i nevýhody. Mezi ty patří především přístup omezený pouze na počítače s přístupem k úložišti, kde je uložená databáze. Tento problém se dá ale řešit podporou externích databází například pomocí webového API.

Komplikovanější je situace také ohledně podpory různých operačních systémů. V tomto případě jsme se ale rozhodli zaměřit na operační systém Windows, mimo jiné proto, že jde o systém, který podle statcounter na počítačích celosvětově používá 75,7% uživatelů (údaj platný k březnu 2022).[11]

3.2 Programovací jazyk a knihovna grafického uživatelského rozhraní

Programovacích jazyků existuje velké množství, ale zdaleka ne všechny jsou vhodné pro tento projekt. Při rozhodování jsme zvažovali jazyky C++, Java a C#. Všechny tři tyto jazyky jsou kompilované (byť některé z nich jen do byte code), což jim oproti interpretovaným jazykům typicky dává výhodu vyššího výkonu. V následující části představíme výhody a nevýhody jednotlivých jazyků a knihoven uživatelského rozhraní, které je možné s nimi používat.

3.2.1 C++

C++ je staticky typovaný jazyk kompilovaný přímo na instrukce konkrétního operačního systému a architektury. Jeho hlavní výhodou je především výkon, který není limitovaný interpretem ani tím, že by byl spouštěn na virtuálním stroji. Díky tomu, že existují kompilátory vycházející z centrálně spravovaného standardu, je možné aplikaci napsanou v C++ zkompilovat pro většinu operačních systémů.

Velkou nevýhodou ale je, že C++ nemá žádnou oficiální knihovnu pro uživatelské rozhraní. Existují zde sice značně rozšířené knihovny jako Qt [12], což je ovšem poměrně rozsáhlý framework, jehož použití má tendenci značně ovlivnit strukturu celé aplikace. Qt navíc v době provádění této analýzy přicházelo s novou výrazně upravenou verzí 6.0, která ještě neměla podporu grafů. Vzhledem k tomu, že pro naši aplikaci je právě uživatelské rozhraní velice důležité, rozhodli jsme se touto cestou nevydávat.

3.2.2 Java

Java je také staticky typovaný jazyk, který se ovšem od C++ výrazně liší v tom, že je kompilován do tzv. byte code, který se následně spouští na virtuálním stroji, a má automatickou správu paměti. To způsobuje, že má Java typicky horší výkon než C++, ale stále výrazně lepší než interpretované jazyky, které při vykonávání musí zpracovávat kód v textové reprezentaci. Na rozdíl od C++ je Java multiplatformní bez potřeby program znovu zkompilovat, ale za cenu toho, že na každém počítači musí být k dispozici Java Virtual Machine[1].

Java má k dispozici několik uživatelských rozhraní. Pokud by měl být využit fakt, že Java je multiplatformní, měla by být použita knihovna Swing. V rámci našeho rozhraní budeme potřebovat mimo jiné možnost vytvářet tabulky s vnořenými tabulkami pro jednotlivé řádky. Bylo by tedy potřeba vytvořit si vlastní variantu tabulkové komponenty, což by díky minimálním zkušenostem s touto knihovnou bylo obtížné.

3.2.3 C#

C# je v mnohém velice podobný jazyku Java. Jde také o staticky typovaný jazyk, který se také kompiluje do byte code (jiného než v případě jazyku Java), vyžaduje přítomnost svého virtuálního stroje (Common Language Runtime) a má automatickou správu paměti. Přestože jsou si jazyky podobné, nejsou stejné. Asi nejviditelnější rozdíl je v přístupu k primitivním typům a v tom, že Java nemá struktury, ale pouze objekty uložené na haldě. Pro nás hlavní výhodou a důvodem pro zvolení C# místo jazyku Java je, že s tvorbou aplikací s grafickým rozhraním máme u tohoto jazyku řadu zkušeností, díky čemuž můžeme vytvořit lepší aplikaci.

Pro svůj běh vyžaduje C# nainstalovaný .NET, tedy sadu standardních knihoven spolu s Common Language Runtime. Ten byl původně ve formě .NET Framework dostupný pouze pro operační systém Windows, ačkoliv existovala i multiplatformní varianta Mono od společnosti Xamarin. V roce 2016 Microsoft, který spravuje a financuje vývoj .NET, vydal open source variantu .NET s názvem .NET Core (později přejmenovaný na .NET). Tato nová varianta je multiplatformní a zároveň má pro Windows podporu stávajících uživatelských rozhraní. Jelikož se dá předpokládat, že .NET Framework nebude dále rozvíjen, rozhodli jsme se vývoj rovnou začít v .NET 5. Po vydání nové verze jsme projekt aktualizovali na .NET 6.

Podobně jako Java má i C# několik grafických uživatelských rozhraní, které jsou ale s výjimkou Xamarin Forms spjaté s operačním systémem Windows.

[1] Virtuální stroj, který zpracovává byte code získaný kompilací jazyku Java.

Vzhledem k rozhodnutí vytvořit aplikaci pro operační systém Windows nám toto omezení nevadí.

Windows Forms

Windows Forms (WinForms) je knihovna poskytující aplikacím grafické uživatelské rozhraní, které je k dispozici v operačním systému Windows. Jde v podstatě o nadstavbu nad základními funkcemi operačního systému a zpřístupňuje tak aplikacím komponenty definované v operačním systému. Přestože existuje již od roku 2001, je stále aktivně vyvíjena a vylepšována.[13] Velkou výhodou je, že má již poměrně dobře odladěný návrhář ve Visual Studio a to i ve verzi pro .NET 5 a 6.

Nevýhodou Windows Forms je, že díky své spjatosti s operačním systémem nemůže být nikdy multiplatformní a zároveň je obtížnější provádět úpravy komponent. Poslední zmíněný nedostatek však vyvažuje množství zkušeností, které s touto knihovnou máme, což je také důvodem, proč jsme se ji rozhodli zvolit pro náš projekt, který bude úpravy komponent vyžadovat.

Windows Presentation Foundation

Knihovna uživatelského rozhraní Windows Presentation Foundation, která je více známá pod zkratkou WPF, vznikla později než Windows Forms a v mnoha ohledech se liší. WPF totiž není nadstavbou knihoven operačního systému. Přesto je však závislá na operačním systému Windows. K návrhu uživatelského rozhraní se zde využívá formát XAML[2], který vychází z XML[3].[14]

WFP umožňuje jednodušší úpravy jednotlivých komponent a vytváření stylů, což je funkce, která je v případě Windows Forms teprve zvažována. Stejně jako i Windows Forms v C# nebo Swing v Java také WPF neobsahuje všechny komponenty, které by se v rámci tohoto projektu hodily. Právě proto jsme se rozhodli využít místo WPF knihovnu Windows Forms, u které je díky zkušenostem snazší provést potřebné úpravy.

Xamarin Forms

Xamarin Forms se od předchozích dvou knihoven liší tím, že podporují i jiné operační systémy než Windows. Knihovna je založena na tom, že je možné sdílet stejný kód mezi aplikací pro operační systém Windows a mobilními platformami se systémy Android a iOS. Důraz je zde právě na mobilní platformy, které by pro tento projekt byly spíše hezkým doplňkem.[15] Z multiplatformních knihoven grafického uživatelského rozhraní by pro tento projekt byla zajímavá spíše — v době provádění této analýzy chystaná — knihovna .NET Multi-platform UI (.NET MAUI), která by kromě operačních systémů podporovaných Xamarin Forms měla podporovat i macOS a za pomoci komunity také Linux.

[2] Extensible Application Markup Language
[3] Extensible Markup Language

3.3 Knihovny pro zobrazování grafů

V rámci .NET Framework existovala pro WinForms knihovna DataVisualization.Charting, která umožňovala zobrazovat množství různých typů grafů. Kromě toho, že knihovna obsahuje mnoho funkcí, by také bylo výhodou, že nejde o knihovnu třetí strany. Při přechodu na .NET bylo tvůrci knihovny rozhodnuto, že tato knihovna již nebude nadále podporována a vyvíjena a bude poskytována již pouze jako NuGet balíček kvůli kompatibilitě se starými aplikacemi. Jelikož používáme novější a aktivně vyvíjený .NET, museli bychom využít zveřejněné zdrojové kódy a nadále si knihovnu upravovat a především opravovat sami. Z tohoto důvodu jsme začali hledat jiné open-source knihovny, které by bylo možné použít. V rámci analýzy jsme zvažovali LiveCharts a OxyPlot, které jsou obě nejen zdarma a open-source, ale také licencované jako MIT[4].

3.3.1 LiveCharts

Knihovna LiveCharts je dle jejího tvůrce reakcí na nedostatek knihoven pro zobrazování grafů, ve kterých by grafy dobře vypadaly a zapadaly do moderního designu, a zároveň by knihovna nestála tisíc dolarů.[16] Knihovna podporuje řadu různých typů grafů od spojnicových, přes koláčové, po specializované jako jsou finanční svícové nebo mapové. Navíc umožňuje v reálném čase body za doprovodu animace přidávat. V tomto ohledu tedy obsahuje o dost víc, než funkce, které by se nám mohly hodit.

Design a animace ovšem způsobují, že knihovna není dostatečně rychlá. Při testování na herní sestavě[5], která je výkonnější než pracovní notebook, který by nejspíše menší investor využíval, se ukázaly potíže s odezvou při větším množství bodů. Při testování proběhlo náhodné vygenerování 30 000, 50 000 a 100 000 hodnot. Již při 30 000 hodnotách byla znát pomalejší reakce na ovládání a při 50 000 hodnotách už byl graf naprosto neovladatelný. Jelikož se v aplikaci bude nacházet i několik grafů najednou, které mohou obsahovat data za několik let, není tento výkon dostatečný.

Výkonového nedostatku této knihovny si je vědom i její autor, a proto existuje její optimalizovaná varianta LiveCharts.Geared, která již zvládá stovky tisíc hodnot. Tuto variantu však již není možné zdarma využít pro komerční či studijní účely.

3.3.2 OxyPlot

Grafy vytvořené pomocí knihovny OxyPlot[17] nevypadají tak moderně jako ty z LiveCharts a i přes to, že je knihovna vyvíjena již od roku 2010, dokumentace jednotlivých typů grafů a jejich parametrů prakticky neexistuje. Na druhou stranu součástí repositáře se zdrojovými kódy je i aplikace s mnoha příklady využití jednotlivých typů grafů, os a dalších nastavení. I tak je ale konfigurace grafů složitější, protože je nejprve nutné zjistit, co která možnost dělá a znamená.

Zásadní výhodou OxyPlot je ale výkon. Při testech na stejné sestavě jako v případě LiveCharts a stejných počtech bodů byl Oxyplot schopen reagovat

[4]Licence, která povoluje neomezeně používat, kopírovat, modifikovat a distribuovat software s touto licencí.

[5]CPU: AMD Ryzen 5 3600, GPU: AMD Radeon 5600 XT, RAM: 16 GB

rychleji. Také zde byla zjevná latence, ale v porovnání s LiveCharts byla řádově nižší. Navíc v případě, že byl graf přiblížen a vykresloval tedy menší množství bodů, byla reakce na ovládání zcela plynulá. Jelikož je pro nás výkon zobrazení grafu důležitý, rozhodli jsme se zvolit k zobrazování grafů knihovnu OxyPlot.

3.4 Práce s daty

Naše aplikace bude pracovat s potenciálně velkým množstvím dat, takže je důležité v rámci analýzy rozhodnout, kde budou uložená, jak budou načtená data interpretovaná v paměti a jaká bude jejich struktura. V této kapitole se budeme postupně věnovat všem třem těmto otázkám.

3.4.1 Úložiště dat

Již v úvodu jsme si dali za cíl, že aplikace má být dostatečně modifikovatelná. Dáme tedy uživateli možnost, aby si sám vybral úložiště dat z podporovaných možností. Aby se podpora nových úložišť jednoduše přidávala, oddělíme správu úložiště do samostatných zásuvných modulů, kde každý bude přidávat podporu pro jeden vybraný typ úložiště. Díky tomu bude možné implementovat zásuvný modul pro lokální databázi (například SQLite), externí databázi v rámci databázového serveru (například PostgreSQL), vzdálené webové API nebo třeba souborový systém.

Ohledně toho, co by měl takovýto zásuvný modul obsahovat a implementovat, se nabízí několik možností. Mohli bychom vyžadovat implementaci rozhraní pro Entity Framework, ADO.NET nebo LINQ. Z výše zmíněného rozsahu možných úložišť je zjevné, že mají velice odlišné možnosti a schopnosti. Použití některé ze zmíněných možností by znamenalo snadné přidání podpory pro úložiště podporované těmito metodami. Ovšem v opačném případě by to znamenalo implementovat řadu rozhraní a funkcí, které náš projekt nakonec nevyužije. Z tohoto důvodu jsme se rozhodli jít jinou cestou.

Místo toho, abychom zásuvným modulům nutili implementaci některého z rozsáhlých univerzálních rozhraní, vytvořili jsme nové jednoduché rozhraní specifické pro potřeby naší aplikace. Naše rozhraní funguje tak, že místo toho, abychom se mohli zeptat na cokoliv pomocí stejného dotazu, náš požadavek cíli na konkrétní operaci a entitu. Toto má pochopitelně za následek to, že pro každou entitu ve výsledku existují tři metody zajišťující operace ukládání, načítání a mazání. Velkou předností tohoto přístupu však je, že implementace samotné komunikace s úložištěm je plně v moci zásuvného modulu, který má zároveň v okamžiku volání metody k dispozici veškeré parametry požadavku. To umožňuje implementovat podporu databáze na externím databázovém serveru i úložiště založené na csv souborech při použití stejného jednoduchého rozhraní. Je na zásuvném modulu, aby určil, co mu jeho úložiště umožňuje a co musí provést sám, a vrátil nám data splňující naše požadavky. Může jít o podporu transakcí, převod datových typů, či umožnění třídění.

Tento přístup má ještě jednu důležitou výhodu. Umožňuje vytvořit implementaci pro dočasnou mezipaměť pro úložiště, u kterých nehrozí konflikt způsobený spuštěním další instance aplikace na jiném zařízení, nebo s implementovaným řešením konfliktů, aniž by o tom musela vědět a rozhodovat samotná aplikace.

V rámci naší práce jsme implementovali podporu pro databázi SQLite, kterou jsme vybrali kvůli tomu, že jde o jednu z často používaných lokálních databází a máme s ní pozitivní zkušenosti.

3.4.2 Reprezentace dat v aplikaci

Máme tedy vymyšleno, jak budeme data ukládat a načítat. Nyní se zaměříme na to, v jaké formě s daty budeme pracovat. Každý záznam v databázové tabulce budeme považovat za entitu s unikátním ID a dalšími parametry. Pro reprezentaci těchto entit jsme se rozhodli zvolit objekty, které budou z pohledu veřejného rozhraní neměnné. Právě neměnnost záznamů je důležitá pro striktní oddělení zatím neuloženého upraveného záznamu a varianty záznamu, která je uložená v databázi a může být například v mezipaměti databázového zásuvného modulu. V případě, že by neměnnost zajištěna nebyla, mohlo by se jednoduše stát, že dojde k úpravě mezipaměti neuloženými daty, které se nakonec uživatel navíc může rozhodnout neuložit. Editaci entit zajistíme pomocí odvozených editovatelných objektů, které při ukládání do databáze změny propíší do původního objektu pomocí protected metod. Objekty jsme zvolili hlavně proto, že hodnotová sémantika struktur neodpovídá našemu využití entit s unikátním ID.

Ověření výkonnosti

Jelikož objekty jsou vytvářeny na haldě, přináší jejich používání paměťovou i výkonovou režii. Právě z tohoto důvodu jsme v rámci analýzy ověřovali, že tato režie není natolik velká, aby příliš zpomalovala aplikaci. Provedli jsme tedy měření pomocí knihovny Benchmark.NET[18] ve variantách s 1 000, 10 000, 200 000 entitami ve formě objektů uložených v poli, struktur uložených v poli, jako řádky multidimenzionálního pole a jako prvky DataTable. Entitu tvořil jeden textový řetězec (referenční typ) a jedno číslo typu decimal (hodnotový typ o velikost 16 B). Během testu došlo k vytvoření zmíněného počtu entit a jejich iterace pomocí for cyklu, který sečetl hodnoty obsaženého čísla. Měření probíhalo na stolním počítači s procesorem AMD Ryzen 5 3600 a 16 GB operační paměti. Naměřené hodnoty můžeme najít v tabulce 3.1.

Typ entity	**1 000 entit**		**10 000 entit**		**200 000 entit**	
	Čas	**Paměť**	**Čas**	**Paměť**	**Čas**	**Paměť**
Objekt	100 µs	125 KB	1,16 ms	1,29 MB	59,4 ms	25,9 MB
Struktura	98 µs	102 KB	1,44 ms	1,06 MB	42,0 ms	21,4 MB
Multidim. pole	114 µs	125 KB	2,87 ms	1,29 MB	57,0 ms	25,9 MB
DataTable	759 µs	336 KB	9,52 ms	3,69 MB	270,3 ms	69,7 MB

Tabulka 3.1: Výsledky měření výkonu entit různých typů obsahujících položku typu decimal a položku typu string při generování daného počtu entit a následné iteraci a součtu číselných hodnot v entitách.

Z výsledků je na první pohled zjevné, že jednoznačně nejhůře vychází DataTable. Kromě výrazně delší doby potřebné k provedení testu bylo také alokováno výrazně vyšší množství paměti, než u ostatních typů entity. Třída DataTable poskytuje velké množství funkcí, které v podstatě simulují databázi, jako jsou kontroly integrity dat, cizí klíče apod.

Lépe je na tom multidimenzionální pole, které se po stránce výkonu i paměťové náročnosti obvykle drží blízko objektových entit uložených v jednodimenzionálním poli. I v tomto případě totiž dochází k vytváření objektů na haldě. Je to způsobené tím, že aby bylo možné v silně typovaném jazyce, jakým je C#, mít v poli uložené různé datové typy, musí být pole takového typu, který je předkem všech ostatních datových typů. V případě C# je jím typ object, což je referenční typ. Použitý decimal je ale hodnotový typ a to znamená, že při uložení do multidimenzionálního pole typu object je potřeba provést tzv. boxování, kdy je na haldě vytvořen objekt obsahující hodnotu daného čísla. To přidává výraznou režii.

Při malém množství entit, které bude v našem případě nejčastější, vychází objekty a struktury velice podobně a až při opravdu velkých počtech entit se ukazuje režie vytváření objektů na haldě, která je však vzhledem k počtu entit zanedbatelná. Rozhodnutí využít objekty pro reprezentaci entit tedy neovlivní významným způsobem výkon naší aplikace.

Editace entit

Objekty reprezentující entity nebudou veřejně editovatelné. Potřebujeme mít ale možnost entity v rámci obrazovky editovat a zobrazovat jejich upravenou variantu mezi ostatními, dokud nedojde k jejich uložení. Vytvoříme proto pro každý typ entity její editovatelnou variantu. Ta bude potomkem svého needitovatelného předobrazu, aby ji bylo možné začlenit mezi ostatní entity a také aby měla přístup k protected metodám a mohla tedy provádět uložení změn.

Tato editovatelná entita dostane při svém vzniku needitovatelnou entitu, která bude určovat její identitu a ze které si zkopíruje počáteční data. Následně bude poskytovat data prostřednictvím stejného rozhraní jako needitovatelná entita, které ale bude umožňovat také zápis. Navíc bude možné prostřednictvím veřejné metody provést uložení do databáze, které bude probíhat už v rámci jinak needitovatelné entity a která si při úspěšném uložení svá data zaktualizuje. Při editaci nové entity dojde během tomto procesu také k zapsání přiřazeného ID do obou variant entity.

Díky tomu, že editovatelná entita nebude stejnou instancí, jako needitovatelná entita, nemůže její editací dojít k přepsání dat v jiných částech aplikace, dokud neproběhne uložení upravených dat.

Speciální situací je načítání dat entity z databáze, kdy je potřeba také provést editaci entity. Pro tuto situaci bude mít zásuvný modul pro komunikaci s databází k dispozici ve svém rozhraní neveřejnou statickou metodu, která mu umožní nahrát do entity nová data přímo bez použití editovatelné varianty entity.

Porovnávání na rovnost

Rovnost objektů je ve výchozím chování v C# určena referenčním porovnáním, tedy jestli obě porovnávané proměnné obsahují referenci na stejný objekt. Při opakovaném načítání entity z databáze by díky tomu docházelo k situaci,

kdy pro dvě entity stejného typu se stejným ID, které jsou z pohledu databáze instancemi jedné entity, nebude platit referenční rovnost, protože se jedná o dva různé objekty se dvěma různými referencemi. Řešením by mohlo být přetížení porovnávání tak, aby se za stejné entity považovaly objekty se stejným ID. Tato varianta ovšem nebude fungovat pro nové entity, které ještě nemají v databázi přiřazené ID a jejich ID je tedy nastavené na -1.

Místo změny způsobu porovnávání změníme způsob vytváření entit tak, aby vždy existoval jen jeden objekt, který reprezentuje danou entitu. Toho dosáhneme pomocí statické metody specifické pro daný typ entity, která přijímá jako argument ID, které má požadovaná entita mít. V případě, že už objekt pro toto ID někdy vznikl a je tedy uložený ve statické datové struktuře pro tento typ entity, vrátí metoda tuto instanci objektu. V opačném případě vytvoří nový objekt a uloží ho do datové struktury. Speciální případ jsou nové entity s ID -1, pro které se vždy bude vytvářet nový objekt, který se nevloží do datové struktury, dokud nedojde k uložení entity a tedy i přidělení ID.

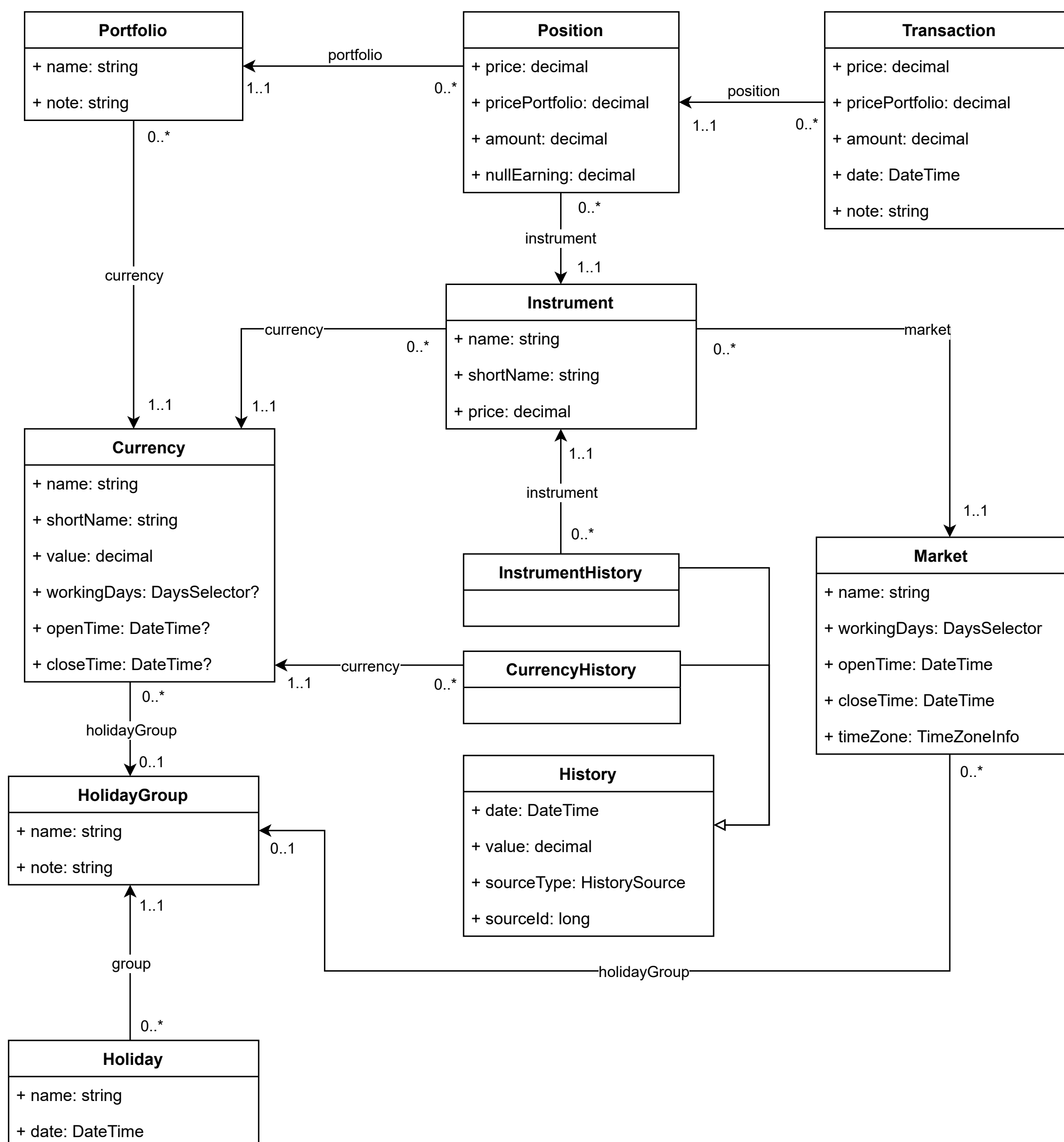
Problém s tímto přístupem budou mít editovatelné varianty entit. V případě, že budeme chtít například ze seznamu určitou entitu vymazat, mělo by dojít k jejímu vymazání bez ohledu na to, jestli jde o editovatelnou nebo needitovatelnou variantu. Podle referenčního porovnání ovšem nejde o stejné entity. Řešením bude v tomto případě přetížení porovnávání u editovatelných entit, které tak ke svému porovnávání budou využívat místo sebe objekty entit, které editují. Díky tomu můžeme dělat porovnání i mezi editovatelnými a needitovatelnými entitami, protože každá editovatelná entita se může identifikovat nějakou needitovatelnou entitou, se kterou se pak daná needitovatelná entita může referenčně porovnat. Aby toto fungovalo ve všech situacích, musí mít také needitovatelná varianta entit přepsané porovnávání a to tak, že v případě, kdy se porovnává s entitou, která není needitovatelnou entitou, nechá druhou entitu provést porovnání za ní. Díky tomu v případě porovnávání editovatelných a needitovatelných entit o výsledku vždy rozhoduje editovatelná entita, která má k dispozici všechny informace potřebné k učinění rozhodnutí.

3.4.3 Datový model

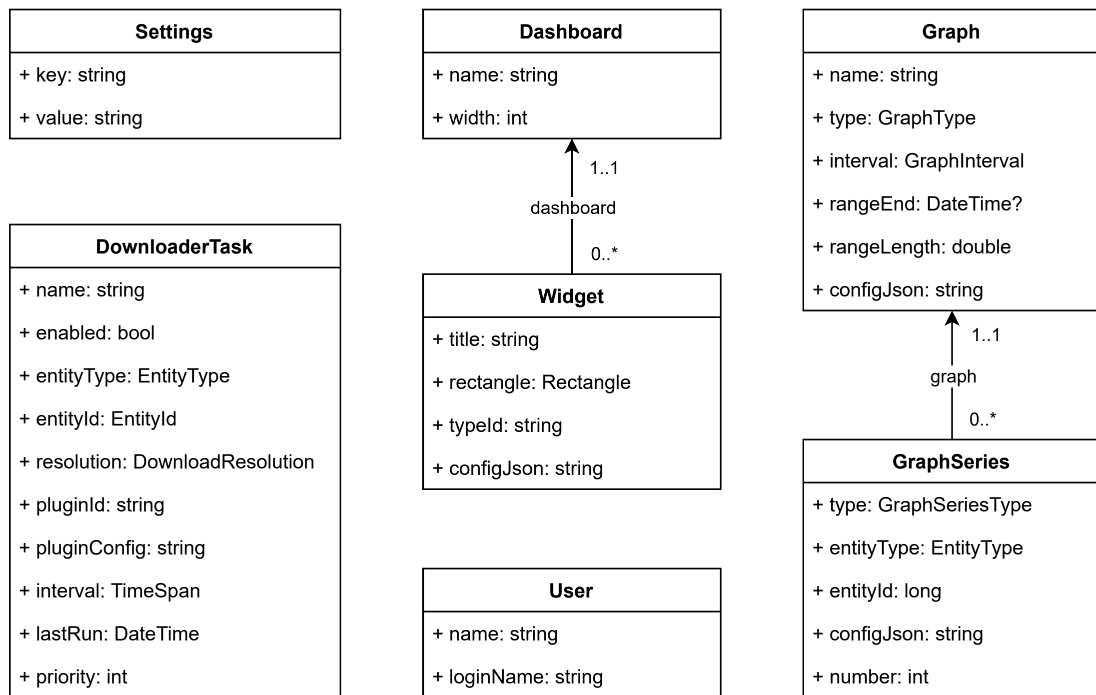
Aby bylo pro vývojáře zásuvných modulů pro práci s úložišti a návrháře struktury těchto úložišť možné vytvářet efektivní návrh využívající všech předností zvoleného úložiště, nemůžeme přesně nadiktovat, jakým způsobem mají být data uložena. Pouze určíme, že datový model bude relační. Dále budeme navrhovat rozhraní, pomocí kterého budeme zásuvným modulům data předávat a také od nich data získávat, a vazby mezi těmito rozhraními. V rámci vybrané entity tedy například určíme, že s položkou „cena“ chceme pracovat jako s datovým typem decimal. To, že SQLite nemá dostatečně přesný číselný datový typ a položka tedy bude v databázi reprezentována jako textový řetězec, je implementační detail zásuvného modulu, který pro nás v této části není důležitý.

Zaměřme se tedy nyní na jednotlivé třídy entit, které můžeme najít v UML diagramech 3.1 a 3.2.

Třída *Portfolio* představuje jedno konkrétní investiční portfolio s jeho názvem, uživatelskou poznámkou a měnou, na kterou se budou převádět hodnoty instrumentů v případě metrik agregovaných přes všechny instrumenty v portfoliu.



Obrázek 3.1: UML diagram části datového modelu s přímými vazbami na entitu portfolia.



Obrázek 3.2: UML diagram části datového modelu bez přímých vazeb na entitu portfolia. Tyto entity stojí buď samostatně nebo jsou na zbytek datového modelu vázány nepřímo pomocí položky s typem entity a položky s identifikátorem entity, ke které se váží.

S portfoliem úzce souvisí třída *Position*, které určuje přítomnost jednoho konkrétního instrumentu v portfoliu. Pozice vzniká přidáním první transakce s daným instrumentem a váže se pouze k jednomu portfoliu. Datové položky pozice obsahují součty ceny a množství přes všechny transakce. V případě ceny pak ve dvou variantách, ve měně instrumentu a ve měně portfolia. Tyto informace slouží k urychlení načítání, protože je možné je měnit inkrementálně a není potřeba je při každém načítání znovu počítat. Položka „nullEarning“ označuje cenu instrumentu, při které bude zisk této pozice nulový.

Pozice vzniká přidáním transakce, která je reprezentována třídou *Transaction* a obsahuje informace o provedené transakci instrumentu v rámci určité pozice v daný čas. Vazba na instrument je realizována skrze pozici. Kromě data provedené transakce obsahuje třída také informace o množství, ceně a uživatelskou poznámku k transakci. Podobně jako v případě pozice je zde cena dvojí, ve měně instrumentu a ve měně portfolia. Transakce označující nákup instrumentu má množství kladné. Oproti tomu transakce, která označuje prodej instrumentu, má záporné množství.

Informace o instrumentu udržují entity třídy *Instrument*, která kromě názvu a zkratky instrumentu obsahuje také vazby na měnu instrumentu a trh, na kterém je instrument obchodován. Existovat může více instrumentů se stejným názvem, které se liší trhem, na kterém jsou obchodovány. Součástí třídy je i položka obsahující aktuální cenu instrumentu. Díky této položce není nutné při zobrazení seznamu instrumentů hledat v historii informace o aktuální ceně každého instrumentu.

Každý instrument je obchodován na nějakém trhu. Informace o trhu se nachází

v třídě *Market*, která obsahuje název trhu a další informace důležité pro stahování dat o instrumentech tohoto trhu. Jde o seznam pracovních dní, tedy dní, během kterých se na trhu obchoduje, čas otevření a zavření trhu a jeho časovou zónu. Jelikož během roku mohou být dny, kdy se na trhu neobchoduje, přestože jde o pracovní den, má každý trh nepovinnou vazbu na skupinu obchodovacích prázdnin.

Instrumenty i portfolia mají vždy určenou svou měnu, která je reprezentována třídou *Currency*. Ta obsahuje jméno, zkratku a aktuální hodnotu měny vůči referenční měně. Na rozdíl od instrumentů měna typicky nemá vazbu na určitý trh a její hodnota se často mění po celý den. Zároveň je však časté, že API pro získávání dat poskytují hodnotu měny pro začátek a konec obchodovací doby dne. Proto třída obsahuje některá pole, které pro instrumenty poskytuje trh. Jde o seznam obchodovacích dnů, vazbu na skupinu obchodovacích prázdnin a čas začátku a konce obchodování. Zmíněné časy slouží k tomu, aby si uživatel mohl zvolit, jaký čas má být přiřazen hodnotám ze začátku a konce obchodovací doby dne. Všechny položky související s tím, jak se s měnou obchoduje, jsou však nepovinné. V případě, že je uživatel nezadá, jsou využity hodnoty z globálního nastavení aplikace.

V případě instrumentů a měn chceme mít k dispozici kromě aktuální ceny také historické záznamy. K tomu nám slouží třídy *InstrumentHistory* pro instrumenty a *CurrencyHistory* pro měny. Obě tyto třídy mají stejné datové položky zděděné z třídy *History* a liší se vazbou na instrument, resp. měnu. Třída určuje hodnotu v daném čase. Dále obsahuje položku „sourceType“, která slouží k určení toho, jak záznam vznikl, např. jestli jde o stažený záznam, či manuálně zadaný. Některé zdroje jsou dále identifikovatelné pomocí identifikátoru v položce „sourceId“. Například stažený záznam byl vytvořen některou z existujících úloh, a tedy v této položce najdeme její identifikátor.

Ve speciální dny v roce, jako jsou Nový rok nebo některé státní svátky, trhy neobchodují. Pro tyto dny se tedy nemá smysl snažit stáhnout data, protože nebudou dostupná. Z tohoto důvodu mají trhy i měny nepovinnou vazbu na skupinu obchodních prázdnin reprezentovanou třídou *HolidayGroup*. Tato třída slouží k pojmenování a sdružení skupiny dní, během kterých se na některém trhu nebo s některou měnou neobchoduje. Pokud se tyto dny shodují pro více různých trhů, například sídlících ve stejném státě, může pro ně být využita stejná skupina. Jednotlivé dny ve skupině jsou reprezentované třídou *Holiday* obsahující datum a uživatelský popisek daného dne.

Úvodní obrazovkou aplikace je uživatelsky konfigurovatelná nástěnka obsahující widgety. Její konfiguraci a sdružení jejích widgetů zajišťuje třída *Dashboard*, která obsahuje název a šířku. Šířka zde znamená počet sloupců čtverečkované sítě, na kterou bude dostupná zobrazovací plocha rozdělena. Aplikace umožňuje existenci pouze jedné nástěnky. Tato třída zde však existuje pro možné budoucí rozšíření. Jednotlivé widgety patřící na danou nástěnku jsou zaznamenané pomocí třídy *Widget*. Mezi její pole patří zobrazovaný titulek widgetu a obdélník určující pozici a velikost widgetu v rámci nástěnky. Pozici osy x a šířku widgetu označují číslo sloupce, resp. počet sloupců. Analogicky to samé platí pro pozici v rámci osy y a výšku widgetu ve vztahu k řádkům nástěnky. Důležitou součástí třídy Widget jsou pole „typeId“ a „configJson“. První z nich obsahuje textový identifikátor typu widgetu, který má být zobrazen (např. widget obsahující graf).

Druhé ze zmíněných polí pak obsahuje konfiguraci specifickou pro daný typ grafu ve formátu JSON. Využití tohoto přístupu nám umožňuje uchovávat konfiguraci různých typů widgetů s často velice odlišnými parametry a počtem parametrů.

V aplikaci je možné nejen zobrazovat uživatelsky definované grafy, ale také je ukládat pro pozdější zobrazení. K tomu slouží třída *Graph* obsahující veškerá metadata uloženého grafu a sdružující jeho datové řady. Kromě informace o typu grafu je v této třídě uložená také konfigurace specifická pro tento typ. Podobně jako v případě widgetů jde o konfiguraci ve formátu JSON. Aby bylo možné sledovat jak pevný časový rozsah (od určitého data do určitého data), tak i dynamický časový rozsah (např. posledních 30 dní), určuje se pouze datum a čas konce rozsahu. Hodnota pole „rangeLength“ pak určuje kolik dní před tímto datem má rozsah začínat. V případě, že je koncové datum nezadané, využívá se aktuální datum a čas. Třída také obsahuje interval, který svou hodnotou určuje minimální interval mezi dvěma datovými body.

Každý graf může obsahovat neomezené množství datových řad, které jsou reprezentované třídou *GraphSeries*. Podobně jako v případě samotného grafu existuje několik typů datových řad (např. spojnicová) a konfigurace parametrů specifických pro daný typ je opět uložena ve formátu JSON. Dále se zde nachází nepřímá vazba na entitu pomocí jejího typu a identifikátoru. Důvodem pro zvolení odlišného přístupu než v případě historie instrumentů a měn je, že zde budeme chtít v typickém případě získávat všechny řady pro daný graf. Rozdělení do zvláštních tříd a tedy i tabulek by zde bylo pouze komplikací pro vývojáře zásuvných modulů pro přístup k úložišti, protože by museli následně výsledky z několika tabulek skládat dohromady. Poslední nezmíněnou položkou je číslo datové řady, které určuje její pořadí v rámci grafu a je využívané k uchování pořadí zadaného uživatelem.

K nastavení úloh pro automatické stahování dat slouží třída *DownloaderTask*. Obsahuje základní informace o úloze, jako je její název, zda je aktivní, minimální časový interval mezi jejími provedeními a datum a čas posledního provedení. Dále pak také informace nutné k provedení samotného stahování dat. Podobně jako u datových řad grafu a ze stejných důvodů se využívá nepřímé vazby na entitu, pro kterou se mají data stahovat. Jelikož je obvyklé, že API s daty mají dva režimy, s hodnotami v pravidelném intervalu a se souhrnnými hodnotami za zvolené období (např. jeden den), je možné pomocí pole „resolution“ rozlišit, které údaje má tato úloha stahovat. S tím souvisí také pole s prioritou úlohy určující, který záznam zůstane uložen v případě, že ke stahování dat jedné entity je k dispozici více různých úloh. Aby bylo možné samotné stažení dat provést, je potřeba mít k dispozici také identifikátor zásuvného modulu, který má zpracování této úlohy na starosti, a konfiguraci pro tuto úlohu specifickou pro vybraný zásuvný modul. Ta je stejně jako v podobných případech u jiných tříd uložená ve formátu JSON.

Pro vstup do aplikace je potřeba se přihlásit pomocí uživatelského jména a hesla. Uživatele reprezentuje třída *User*, která však obsahuje pouze přihlašovací jméno a jméno k zobrazení. Zbylá pole potřebná pro přihlášení nejsou v rámci dat této třídy pro aplikaci dostupná a jsou implementačním detailem zásuvného modulu pro práci s úložištěm. Uživatel může být aktuálně pouze jeden, ale podobně jako v případě nástěnky jde o přípravu pro možné budoucí rozšíření, které by umožnilo mít více uživatelských účtů, aby nebylo nutné sdílet jedno heslo. Případně by bylo možné také zcela oddělit data jednotlivých uživatelů a umožnit

tak, aby více různých uživatelů využívalo jednu společnou databázi. Tato varianta by byla užitečná především v případě vzdáleného přístupu k databázi například přes webové API.

Globální nastavení aplikace jsou uchovávána pomocí třídy *Settings*, která reprezentuje jednu dvojici klíč-hodnota. Klíčem je textový řetězec a hodnoty jsou serializované také do textového řetězce.

3.4.4 Reprezentace časových položek

Jednotlivé trhy obchodují v různých časových zónách a stejně tak se stažená data váží k určité časové zóně. V případě, že by se navíc uživatel i s databází přestěhoval do jiné časové zóny, nastal by problém s tím, že by stará data zůstala v původní časové zóně a nová data by byla přidávána v nové časové zóně.

První zvažované řešení bylo přidat do nastavení novou položku, ve které by uživatel zvolil, kterou časovou zónu chce používat. Tato varianta by ale po změně uživatelovi časové zóny způsobila značnou nepřehlednost. Změna nastavení by navíc znamenala potřebu aktualizovat veškerá data aplikace.

Zvolené řešení místo toho, aby uživatel vybíral časovou zónu, vždy používá lokální časovou zónu a je zodpovědností zásuvného modulu pro přístup k úložišti, aby datum při ukládání a načítání převáděl mezi lokální časovou zónou a pevnou časovou zónou zvolenou autorem zásuvného modulu. Díky tomu uživatel vždy vidí veškerá data a časy ve své časové zóně a její změna nezpůsobí žádný problém. Pokud zdroj dat poskytuje hodnoty v jiné než lokální časové zóně, je zodpovědností zásuvného modulu pro stahování dat provést při požadavku převod.

3.5 Stahování dat z veřejných API

V rámci naší aplikace chceme uživateli umožnit, aby si zvolil, ze kterého zdroje budou automaticky stahovány hodnoty instrumentů a měn, které má v aplikaci zadané. Rozsah takovýchto zdrojů může být velice široký od souborů ve formátu csv na síťovém disku po webové API. Z toho plyne, že získávání a zpracování surových dat bude pro jednotlivé zdroje vyžadovat různé postupy. Aby tedy bylo jednoduché zdroje přidávat, budou implementovány jako samostatné zásuvné moduly, podobně jako tomu je v případě úložišť.

Díky rozdílnosti různých typů zdrojů, ale i odlišnostem mezi jednotlivými webovými API nebo formáty, nemůžeme dělat žádné předpoklady o tom, jak používaný zdroj dat funguje nebo jaké má možnosti. Naše rozhraní, pomocí kterého budeme zásuvné moduly žádat o data, tedy musí být tak jednoduché, jak to jen jde. Případné optimalizace, jako je uložení všech načtených záznamů do mezipaměti, musí dělat až sám zásuvný modul podle chování svého zdroje. Naše aplikace tedy bude postupovat od nejstaršího záznamu, který je zapotřebí stáhnout, k nejnovějšímu a zásuvného modulu se vždy dotáže pouze na jednu konkrétní hodnotu. Pokud je ze zdroje dat schopen získávat jen jednu hodnotu v jednu chvíli, získá a vrátí nám právě tu. Ale v případě, že musí stahovat více hodnot najednou, může se spolehnout na to, že žádné starší hodnoty již potřebovat nebudeme. Novější hodnoty získané v rámci jednoho požadavku si může uložit v mezipaměti, ze které následně obslouží naše další dotazy.

3.5.1 Konfigurace zásuvného modulu

Pro správnou funkci bude zásuvný modul potřebovat některé informace. Ty může získat ze dvou zdrojů, konfigurace úlohy a globální konfigurace zásuvného modulu.

Globální konfigurace zásuvného modulu je společná pro všechny úlohy obstarávané tímto zásuvným modulem a typicky slouží k uložení např. bezpečnostního klíče pro přístup k webovému API. V případě, že je zásuvný modul schopen stahovat také údaje o měnách, bude typicky potřebovat identifikátor referenční měny tak, jak je v jeho zdroji dat definovaný. Globální konfigurace se ukládá ve formátu JSON pomocí třídy Settings.

Konfigurace specifická pro danou úlohu je uložená přímo v entitě úlohy ve formátu JSON, což umožňuje zásuvnému modulu uložit si jakákoliv nastavení, která bude pro stažení hodnot potřebovat. V typickém případě půjde o identifikátor instrumentu nebo měny, pro kterou se mají hodnoty stahovat. V rámci různých zdrojů dat totiž může mít entita různé identifikátory a není proto možné použít například její zkratku.

Provedení všech zmíněných konfigurací musí být možné prostřednictvím grafického rozhraní aplikace. Za tímto účelem bude zásuvný modul aplikaci poskytovat třídu grafického rozhraní Winforms typu UserControl, kterou je možné vložit do konfiguračního okna globálního nastavení zásuvných modulů nebo úlohy. Nevýhodou tohoto přístupu je, že v rámci zásuvného modulu je potřeba vytvářet část grafického rozhraní. Tento přístup má však pro nás i pro tvůrce zásuvných modulů také značné výhody. Především je pro tvorbu grafického rozhraní možné využít návrhář a vývojář není nijak omezen v tom, co bude konfigurační okno jeho zásuvného modulu obsahovat. Díky tomu může například k poli pro zadání identifikátoru instrumentu přidat tlačítko pro ověření zadaného vstupu nebo dynamicky upravovat obsah na základně vstupu uživatele. Nám toto řešení přináší jednoduchý způsob, jak poskytovat i velice pokročilé konfigurátory bez nutnosti vytvářet složité rozhraní, které by překládalo požadavky zásuvného modulu na příkazy pro tvorbu grafického rozhraní ve Winforms. Toto obslužné rozhraní by navíc i tak neobsahovalo všechny možnosti, protože to by v podstatě znamenalo vytvořit vlastní „klon“ rozhraní Winforms.

3.5.2 Parametry úlohy

Už víme, že zdroje dat se mohou značným způsobem lišit a že jednotlivé zásuvné moduly je potřeba konfigurovat. Toto zároveň znamená, že nemůžeme vytvořit úlohu zcela automaticky. Nevíme totiž, jaký je pro daný zdroj vhodný interval, jaký je název instrumentu, resp. měny, v rámci tohoto zdroje, ani jak je spolehlivý. V tomto směru nám nezbývá nic jiného, než se obrátit na uživatele. Na druhou stranu by samozřejmě bylo nežádoucí, aby konfigurace obsahovala příliš velké množství parametrů a zabrala tedy uživateli dlouho. Snažili jsme se proto navrhnout řešení, které bude vyvažovat tyto dva aspekty. Ve výsledku tedy uživatel musí nastavit pro nás potřebné pouze 3 obecné parametry a typicky jednotkové množství nastavení zásuvného modulu (např. pouze identifikátor instrumentu v rámci zdroje). Do tohoto počtu nezapočítáváme název úlohy, a zda je úloha aktivní.

Základním parametrem je časový interval mezi provedeními dané úlohy. Tímto

parametrem uživatel zvolí, za jak dlouho se úloha znovu provede od jejího posledního provedení. Jelikož může být aplikace v té době zrovna vypnutá, nejde o striktní, ale o minimální interval. Úloha se tedy provede nejdříve za daný interval.

Je nutné počítat s tím, že zdroj dat může být někdy nedostupný z důvodu chyby nebo i nastaveného omezení přístupu jen na určitou denní dobu. Řešením tohoto problému je umožnit uživateli vytvoření více úloh pro jeden instrument nebo měnu s různými nastaveními. Rozdílem mezi nimi může být volba jiného zásuvného modulu a tedy i jiného zdroje dat. Různé zdroje dat mohou mít ale různé hodnoty a uživatel může preferovat hodnoty z jednoho zdroje dat před hodnotami z ostatních. K tomuto účelu má každá úloha číselnou prioritu. V případě, že je dostupná hodnota z úlohy s vyšší prioritou, je dříve uložená hodnota z úlohy s nižší prioritou smazána a nahrazena.

Podobně se také mohou lišit pro čas označující začátek či konec obchodování intradenní hodnoty a hodnoty otevření a zavření trhu ze souhrnných údajů dne. Webové API navíc typicky tyto údaje poskytují pomocí různých dotazů. Prostřednictvím parametru rozlišení se proto odlišují úlohy stahující intradenní hodnoty a úlohy stahující souhrnné hodnoty. Tento parametr má navíc vliv i na zmíněný parametr priority. Typicky se dá předpokládat, že souhrnné informace zveřejněné až po úplném uzavření trhu budou přesnější než průběžné informace. Z tohoto důvodu je priorita úloh se souhrnným rozlišením vždy vyšší, než všech úloh s intradenním rozlišením.

3.5.3 Proces zpracování úlohy

Když už známe parametry úloh a máme možnost provést jejich konfiguraci, můžeme se přesunout k samotnému procesu zpracování úlohy.

Příprava potřebných informací

Abychom se mohli dotazovat zásuvného modulu, který má na starosti získávání dat pro tuto úlohu, musíme vědět, jaká data potřebujeme získat. V případě intradenního rozlišení se snažíme získat údaj za každých pět minut. U souhrnného denního rozlišení nám jde pouze o hodnoty z otevření a zavření trhu pro každý den. To ovšem znamená, že potřebujeme vědět, v jakém časovém rozmezí dne má smysl hodnoty zkoušet získat, resp. kdy je otevření a zavření trhu. K tomu nám v případě instrumentu slouží vazba na entitu trhu, která tyto informace obsahuje.

Složitější je situace v případě měn, u kterých není vždy jasné, v jakém časovém rozmezí se s nimi obchoduje, a často se jejich obchodování nevztahuje k jednomu trhu. Tento problém jsme se rozhodli vyřešit tak, že necháme uživatele, aby si určil, na jaký čas chce mapovat hodnoty otevření a zavření trhu s měnou a tím i časové rozmezí, pro které se data budou stahovat.

Zatím jsme se věnovali pouze času v rámci jednoho dne. Nemá však smysl stahovat data za posledních několik let, pokud investor obchoduje teprve rok. Začátek tohoto rozmezí si tedy může podle svých potřeb uživatel zvolit v nastavení aplikace. Koncovým okamžikem rozmezí je vždy aktuální datum a čas.

Z rozmezí nesmíme zapomenout vyloučit výjimečné dny a dny v týdnu, během kterých se neobchoduje. Kdyby tyto dny nebyly vyjmuty, mohla by se aplikace opakovaně snažit stahovat hodnoty, které nikdy nebudou dostupné, a zbytečně

tak plýtvat někdy omezený počet přístupů ke zdroji dat. Obě tyto informace nám poskytne entita trhu, resp. obdobné položky měny, a související vazba na skupinu obchodních prázdnin. Tuto operaci komplikují časové zóny, které v tomto případě musí být zohledněny. Den musí být vztažen k časové zóně trhu, protože v době, kdy v Japonsku je již sobota a zdejší trh tedy nemusí tento den vůbec obchodovat, je v České republice stále pátek. Pokud by tedy čas zároveň spadal do doby, kdy se v Japonsku ve všední dny obchoduje, aplikace by se snažila pro tuto dobu data stáhnout a pochopitelně neuspěla. Kvůli této situaci obsahuje trh také informaci o tom, kterou časovou zónou se řídí. To umožní lépe kontrolovat, zda se mají pro daný moment data stahovat.

Kontrola záznamů a získávání chybějících záznamů

Nyní si načteme z úložiště všechny *stažené* záznamy z historie naší entity ve zvoleném rozsahu a budeme jedním průchodem zjišťovat, jestli máme záznam pro každý okamžik našeho rozmezí. Pokud objevíme datum a čas, pro který neexistuje záznam nebo jde o záznam z úlohy s nižší prioritou, požádáme zásuvný modul o stažení hodnoty pro tento datum a čas. Pro získanou hodnotu vytvoříme záznam do historie a pokračujeme v průchodu. Po jeho dokončení uložíme získané záznamy.

Při získávání hodnoty může dojít k chybě způsobené problémem na straně zdroje dat nebo hodnota nemusí být pro daný datum a čas dostupná. V takových případech zásuvný modul vyhazuje výjimku. V případě, že jde pouze o nedostupnost hodnoty a je stále šance, že naše další dotazy bude možné úspěšně vyřídit, využívá se výjimka ValueNotAvailableException. Zareagujeme na ni tak, že záznam vynecháme a budeme pokračovat, jako by záznam již existoval. Pokud zásuvný modul dopředu ví, že tento ani další dotazy již nebude možné vyřídit (např. novější data ještě nejsou dostupná) může nás o tom parametrem zmíněné výjimky informovat. Druhá varianta je, že není možné data ze zdroje získat kvůli nedostupnosti služby nebo špatnému nastavení. V takovém případě se využívá výjimka ValueSourceNotAvailableException, která rovnou říká, že nemáme pokračovat ve snaze získat další hodnoty.

Mazání starších záznamů

Kdybychom záznamy pouze přidávali mělo by to za následek značný růst potřebné velikosti úložiště. Jemné informace o vývoji jsou přitom nejzajímavější pouze pro posledních několik dní. Poté již uživatele bude zajímat spíše dlouhodobý vývoj než momentální výkyvy. Proto se při průchodu kontroluje také to, jestli nejsou nějaké záznamy příliš staré a neměly by být smazány. Jako smysluplnou frekvenci historie jsme zvolili, že pro posledních 24 hodin budeme uchovávat záznam každých 5 minut, pro poslední 2 až 5 dní každou celou hodinu a pro starší data budeme uchovávat pouze hodnoty otevření a zavření trhu.

U některých úložišť, mezi které patří i databáze SQLite, se při mazání umístění záznamu pouze označí jako prázdné, ale k faktickému mazání nedochází. To způsobuje, že nikdy nedochází ke zmenšení velikosti databáze. Kvůli této situaci po provedené mazání starších záznamů požádáme zásuvný modul pro práci s úložištěm, aby úložiště uklidil.

3.5.4 Import instrumentů

Provádět zcela automatické vytváření úloh pro stahování dat nemůžeme, ale i tak můžeme uživateli jejich vytváření usnadnit. Místo vytvoření instrumentu pomocí jeho editačního okna, umožníme uživateli instrument vyhledat pomocí jednoho ze zásuvných modulů, který touto funkcí disponuje (implementuje rozhraní IInstrumentImporterProvider). Díky tomu dojde k ověření, že instrument ve zdroji dat existuje a získáme také jeho identifikátor specifický pro daný zdroj dat. To nám následně dává možnost vytvořit nejen entity instrumentu, ale rovnou také úlohy pro stahování z tohoto zdroje dat. Jako interval využijeme hodnotu, kterou uživatel bude moct upravit v globálním nastavení aplikace a nastavení úlohy specifické pro vybraný zdroj dat necháme vygenerovat zásuvný modul, který nám ho už pouze předá ve formátu JSON.

4. Implementace

V předchozí kapitole jsme zvolili platformu a programovací jazyk, které budeme využívat, a navrhli datovou strukturu a řešení hlavních funkcí naší aplikace. Nyní se zaměříme na samotnou implementaci těchto funkci a řešení důležitých detailů, které jsme v rámci analýzy neřešili.

4.1 Struktura aplikace

Zaměřme se nejprve na samotnou strukturu aplikace (viz. obrázek 4.1). Hlavní a propojovací částí celé aplikace je třída implementující rozhraní *IPortEvalApp*. Po spuštění aplikace dojde voláním statické metody *IPortEvalApp.InitializeApp*, která jako parametr přijímá instanci implementující toto rozhraní, k naplnění statické položky *Instance*. Tuto statickou položku následně všechny části aplikace využívají k přístupu k dalším částem aplikace.

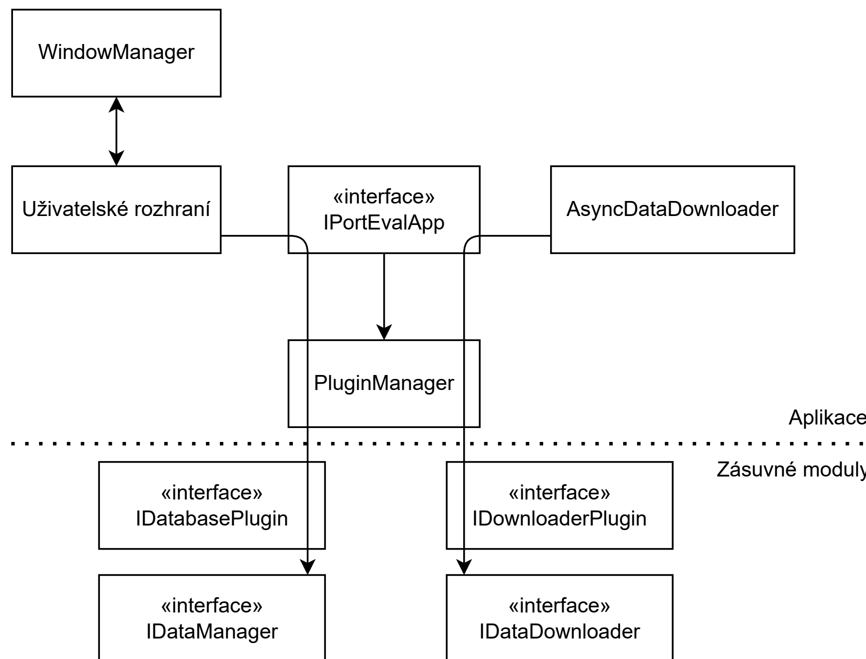
Již jsme řekli, že přístup k úložišti budeme řešit pomocí zásuvných modulů a stejně tak i stahování hodnoty měn a instrumentů. Zásuvné moduly pro přístup k úložišti reprezentované rozhraním *IDatabasePlugin* nám tedy budou poskytovat třídu implementující rozhraní *IEditableDataManager*, prostřednictvím které budeme s daty manipulovat. Podobně zásuvné moduly pro stahování dat reprezentované rozhraním *IDownloaderPlugin* nám poskytnou třídu implementující rozhraní *IDataDownloader*, které se budeme dotazovat na hodnoty instrumentu nebo měny. Abychom od zásuvných modulů mohli tyto třídy získávat, musíme být schopni je načíst a mít možnost získat jejich instanci podle jejich identifikátoru. K tomu nám bude sloužit třída *PluginManager*. Zásuvným modulům se budeme podrobněji věnovat později v části 4.12.

Uživatelské rozhraní inicializuje a spravuje *WindowManager*, který je implementován dle návrhového vzoru „singleton“. Jeho funkci se budeme více věnovat v části 4.2.2 věnované možnosti mít otevřených více oken najednou. Abychom mohli v uživatelském rozhraní zobrazovat a ukládat data, má uživatelské rozhraní prostřednictvím instance IPortEvalApp přístup k IEditableDataManager ve formě IDataManager bez editačních metod. Rozhraní IDataManager je zde použito, aby se zabránilo úpravě entity jinou cestou, než skrze samotnou entitu. Vzhledem k tomu, že aplikace fakticky vždy pracuje instancí implementující IEditableDataManager budeme pro zkrácení dále v tomto textu používat název IDataManager i pro editace, pokud nebude nutné rozdíl zdůraznit.

Automatické stahování dat probíhá na pozadí pomocí třídy *AsyncDataDownloader*, která k získávání hodnot využívá instanci IDataDownloader získanou ze zásuvného modulu, jehož instanci poskytne PluginManager. Fungování této třídy se budeme dále věnovat v části 4.6.

4.2 Uživatelské rozhraní

V naší aplikaci bude uživatel přepínat mezi hned několika obrazovkami. Pokud bychom tyto obrazovky implementovali jako jednotlivá formulářová okna, znamenalo by přepínání obrazovek vytváření nových oken a zavírání těch starých. To by



Obrázek 4.1: Diagram zobrazující hlavní části aplikace a jejich propojení.

bylo jednak pomalé, ale také nehezké. Z tohoto důvodu budeme využívat pouze jedno obecné formulářové okno, do kterého budeme vkládat jednotlivé obrazovky. Za tímto účelem jsme vytvořili rozhraní *IFormBase* a *IScreen*.

Rozhraní IFormBase implementuje třída formulářového okna, které je rozděleno na tři části: boční panel, titulek obrazovky a obsah obrazovky. Rozhraní IFormBase je důležité zejména kvůli tomu, aby jednotlivé obrazovky implementující rozhraní *IScreen* a boční panel mohly okno, do kterého jsou vloženy, ovládat. Obrazovky i boční panel totiž potřebují mít možnost oknu sdělit, že má dojít ke změně obrazovky. K tomu slouží metoda *IFormBase.SetScreen(IScreen)*. Jednotlivé obrazovky pak mohou také skrýt boční panel nebo nastavit titulek obrazovky.

Každá obrazovka je tvořená UserControl a implementuje rozhraní IScreen, pomocí kterého je možné získat její instanci právě jako UserControl nebo inicializovat načítání obrazovky pomocí metody *LoadDataAsync*.

4.2.1 Obrazovky obsahující neuložená data

V aplikaci jsou dva hlavní typy obrazovek, k zobrazování seznamu entit určitého druhu a k zobrazování detailu a editaci vybrané entity. První zmíněný typ obrazovky můžeme klidně kdykoliv zavřít a nedojde tím ke ztrátě žádných dat. To samé se ale nedá říct o druhém typu obrazovek. Zde může mít uživatel neuložené změny, o které nesmí přijít kvůli například překliknutí se v bočním panelu. Tyto obrazovky proto kromě rozhraní IScreen implementují ještě rozhraní *ICloseableScreen*, ve kterém najdeme dvě metody: *PrepareCloseAsync* a *CloseScreen*.

Metoda PrepareCloseAsync uvědomí obrazovku o tom, že existuje požadavek na její zavření a měla by si tedy uzavřít všechny rozpracované záležitosti. Právě v rámci volání této metody by se tedy obrazovka měla uživatele zeptat, zda chce neuložená data zahodit nebo uložit. Jak je v podobných případech zvykem, uživatel má také třetí možnost a to sice celou operaci zrušit a na obrazovce zůstat.

Právě proto metoda PrepareClose také vrací booleovskou hodnotu určující, zda je možné obrazovku zavřít. To se hodí také v situaci, kdy při ukládání dojde k chybě, protože v takovém případě opět nelze obrazovku uzavřít.

Metoda CloseScreen bývá typicky o poznání jednodušší. Zde už totiž nic nekontrolujeme a zkrátka obrazovku zavřeme. Metoda se však volá pouze v případě, že dochází opravdu k zavření obrazovky a nikoli k přechodu na jinou obrazovku. Metoda se proto nevolá při výběru jiné obrazovky z bočního panelu. Ve skutečnosti tak zavření obrazovky může znamenat otevření jiné obrazovky. Tato situace nastává proto, že v typickém případě se chceme při opouštění obrazovky detailu entity vrátit na obrazovku se seznamem všech entit.

4.2.2 Možnost otevření více oken

Abychom se vyhnuli konfliktům při současném využívání lokálních dat více instancemi aplikace, rozhodli jsme se pomocí třídy Mutex zakázat spouštění více než jedné instance. Zároveň ale chceme umožnit, aby si uživatel mohl dát dvě okna vedle sebe a porovnávat údaje. Jelikož nechceme, aby se aplikace ukončila při zavření prvního vytvořeného okna, vytvořili jsme si vlastní třídu odvozenou od ApplicationContext nazvanou *WindowManager*. Pomocí této třídy můžeme sledovat počet otevřených oken a aplikaci ukončit až ve chvíli, kdy budou všechna zavřená.

Souběhu při ukládání dat předcházíme tak, že nevytváříme pro každé okno zvláštní vlákno. To nám sice brání pracovat s jedním oknem, zatímco v jiném probíhá načítání dat, ale velkou výhodou je, že uživatel nemůže ukládat stejnou entitu ve dvou oknech najednou. Přesto musíme řešit otázku aktualizace zobrazovaných dat. Vzhledem k tomu, že k tomuto problému dochází nejen kvůli možnosti mít otevřených více oken najednou, budeme se mu věnovat později v části 4.8.

4.2.3 Dialogová okna

Nastávají situace, kdy bychom potřebovali zobrazit obrazovku modálním způsobem jako dialog. Dialog se od obrazovky liší tím, že je modální a při uzavření se zavírá celé jeho okno. Rozhodli jsme se tedy využít infrastruktury, kterou máme připravenou pro obrazovky, i pro dialogy. Dialog tak není nic jiného než konvence, že vybraná obrazovka se bude využívat jako dialog. Obrazovka si je tohoto faktu vědoma a tak se v případě, že implementuje rozhraní ICloseableScreen, při volání metody CloseScreen místo uzavření obrazovky zavírá celé okno.

Z pohledu třídy implementující IFormBase je dialog jen obyčejná obrazovka a své chování proto nijak nemění. My ale potřebujeme tuto třídu nastavit tak, aby se chovala a vypadala jako dialog. Tato nastavení se liší podle typu dialogu. Jde o skrytí postranního panelu a často mezi ně může patřit změna velikosti okna, omezení její změny uživatelem a skrytí obslužných tlačítek v záhlaví okna.

Zbývá tedy provést inicializaci okna implementujícího IFromBase a vložit do něj dialogovou obrazovku, která už si zbytek nastaví. O to se stará již zmiňovaný WindowManager, který po zavolání metody *ShowAsDialog* inicializuje nové okno, nastaví zobrazovanou obrazovku na tu, kterou metoda dostala jako parametr, a pomocí metody *Form.ShowDialog()* zobrazí okno jako dialog.

4.3 Komponenty uživatelského rozhraní

Již při výběru knihovny grafického uživatelského rozhraní bylo zřejmé, že pro naše potřeby bude nutné si některé komponenty upravit nebo vytvořit. Konkrétně jde o datovou tabulku s možností zobrazit k vybranému záznamu vnořenou tabulku s detailem, systém widgetů, skupinu komponent s možností deaktivace pomocí zaškrtávacího políčka v záhlaví a horizontální seznam zaškrtávacích políček.

4.3.1 Tabulka s vnořenými tabulkami

Při tvorbě grafického rozhraní se nám hodila tabulka, která po kliknutí na záznam byla schopná zobrazit pod tímto záznamem další tabulku obsahující seznam položek souvisejících s vybranou položkou původní tabulky. To nám umožňuje zobrazit detail několika entit v rámci jedné obrazovky. Užitečné je to v situaci, kdy máme hierarchickou strukturu, ve které jedna entita sdružuje seznam dalších souvisejících entit. Takovýchto případů máme v aplikaci hned několik. Jako příklad můžeme uvést skupinu obchodních prázdnin a seznam jednotlivých dní této skupiny.

Tabulky ve Winforms (DataGridView) tuto funkci nemají, ale i přesto jsme je mohli za tímto účelem využít. Vytvořili jsme uživatelsky definovanou komponentu (UserControl) s názvem *MultiLevelDataGridView*, jejíž celá plocha je vyplněná komponentou DataGridView. Při kliknutí na řádek dojde k přidání nového řádku pod vybraným řádkem. Následně vytvoříme novou instanci MultiLevelDataGridView a nastavíme její pozici a velikost tak, aby vyplňovala tento řádek. Tím získáváme zanořenou tabulku. Jelikož jsme jako vnořenou tabulku využili opět MultiLevelDataGridView, je možné vnoření neomezeně opakovat.

Pochopitelně musíme reagovat na události vzniklé interakcemi uživatele, aby vnořená tabulka zůstala na svém místě v řádku. Nastane-li tedy například změna velikosti sloupce nebo řádku, je potřeba všechny vnořené tabulky projít a jejich pozici a velikost aktualizovat.

4.3.2 Widgety

Abychom měli možnost zobrazovat více grafů nebo tabulek najednou v rámci nástěnky, hodil se nám systém widgetů. Navíc chceme, aby uživatel měl možnost widgety konfigurovat a přesouvat. Za tímto účelem jsme implementovali UserControl komponenty *WidgetHolder* a *WidgetPlace*.

WidgetHolder slouží jako kontejner, do kterého jsou widgety vkládány. Vyplňuje ho TableLayoutPanel[1], který se stará o samotné zobrazování widgetů v rámci čtverečkované sítě.

Všechny widgety implementují rozhraní *IWidget* a při umístění do WidgetHolder jsou zabalení do WidgetPlace. To umožňuje zobrazit grafu hlavičku s titulkem a dalšími možnostmi. Pomocí událostí pak WidgetPlace umožňuje provádět komponentě WidgetHolder další operace navazující na uživatelské interakce.

[1] Kontejner umožňující tabulkové rozvržení.

Velikost widgetů a jejich umístění

Máme dvě možnosti, jak určovat velikost a umístění jednotlivých widgetů. Buď přímo v pixelech, nebo v diskrétních jednotkách. Využití pixelů jako jednotky by způsobovalo mnohé problémy. Bylo by velice obtížné zarovnat dva widgety na stejnou úroveň a v případě, že dojde ke změně velikosti obrazovky, by muselo proběhnout přepočítání a přeškálování všech hodnot. Rozhodli jsme se tedy zvolit diskrétní jednotky, se kterými bude pro uživatele jednodušší pracovat. Ty následně využíváme jak pro velikost widgetů, která určuje, kolik buněk mřížky widget zabírá, tak i pro jejich umístění do mřížky.

Zarovnávání widgetů do mřížky provádíme pomocí TableLayoutPanel. Nyní se zaměříme na její vlastnosti. Konkrétně na to, jak vypadají jednotlivé buňky této mřížky a jaká je jejich velikost. Rozhodli jsme se, že půjde o čtverečkovanou síť s pevným konfigurovatelným počtem sloupců a proměnlivým počtem řádků. Velikost jednotlivých buněk je pak dána podílem šířky obrazovky a počtem sloupců. V případě, že se při šířkou určené velikosti buněk na obrazovku nevejdou všechny řádky, je zobrazen posuvník.

Druhou zvažovanou variantou bylo umožnit nastavení počtu řádků i sloupců s tím, že by byla celá obrazovka rozdělena na přesně tyto počty místo proměnného počtu řádků. Buňky by tak byly obdélníkové místo čtvercových. Výhodou tohoto přístupu by bylo to, že by uživatel vždy viděl všechny widgety i při změně velikosti okna a bez nutnosti posuvníků. V takovém případě by ovšem docházelo ke změně poměru stran jednotlivých widgetů, což ve zvoleném řešení nenastává. Navíc by k této situaci došlo i v případě změny počtu řádků či sloupců. To by v praxi znamenalo nutnost projít následně všechny widgety a upravit jejich velikosti a pozice.

Vytváření a přesouvání widgetů

Hlavním důvodem, proč tento systém widgetů vytváříme, je, aby byl uživatel schopný vytvořit si své vlastní rozvržení informací, které ho zajímají. Je důležité, aby bylo pro uživatele co nejjednodušší toto rozvržení vytvořit. Proto jsme se rozhodli, že pozice widgetu nebude uživatelem zadávána číselně, ale pomocí principu drag-n-drop. Uživatel tedy může měnit pozici widgetu tak, že uchopí jeho hlavičku s titulkem a tahem myši ho přesune do nového volného umístění.

Podobně jednoduché by pro uživatele mělo být také přidání widgetu. Implementovali jsme tedy možnost si widget „nakreslit“. Ve vytvářecím režimu uživatel stiskne levé tlačítko myši na místě, kde bude levý horní roh widgetu, a tahem myši se posune na místo, kde bude pravý dolní roh. Tam tlačítko pustí a WidgetHolder vytvoří nový prázdný widget. Ten následně může uživatel konfigurovat. Tímto jsme také vyřešili zadání prvotní velikosti widgetu, protože ta je určena zvolenými dvěma rohovými body. Pro případ, že by chtěl uživatel mít velikost přesně stejnou jako u jiných existujících widgetů, může po vytvoření velikost změnit v nastavení widgetu nebo opět pomocí principu drag-n-drop tahem pravého dolního rohu widgetu.

Režim jednoho widgetu

Může nastat situace, kdy se nám hodí zobrazit některý z widgetů tak, aby byl na celou obrazovku. Z tohoto důvodu se v komponentě WidgetHolder aktivuje speciální režim v případě, že má pouze jeden sloupec. V tomto režimu přestávají platit některé věci, které jsme zmiňovali v předchozích částech. Především jde o to, že místo několika čtvercových buněk ve sloupci se zobrazování změní na jednu buňku, která vyplní celý prostor. To umožňuje celoobrazovkové zobrazení jednoho widgetu. Jelikož jde o speciální situaci, ve které se nepředpokládá, že by měl uživatel možnost si widget nějak upravovat a jeho přesouvání nedává smysl, je také skryto záhlaví widgetu.

4.3.3 Deaktivovatelná skupina komponent

V rámci analýzy datového modelu (viz. část 3.4.3) jsme u měny zmiňovali, že její položky otevření a zavření trhu, obchodních prázdnin a seznamu dnů v týdnu, během kterých se obchoduje, jsou nepovinné. Pro zdůraznění této informace by se nám hodila jasná a jednoduchá komponenta, díky které bude zřejmé, že vybrané položky patří do jedné skupiny a je možné je deaktivovat. O zdůraznění příslušnosti ke skupině se může postarat kontejnerová komponenta GroupBox, která zobrazí kolem obsažených komponent rámeček s titulkem. GroupBox ale neumožňuje místo titulku zobrazovat zaškrtávací políčko, které by všechny komponenty deaktivovalo.

Vytvořili jsme tedy komponentu CheckedGroupBox, která tentokrát dědí od třídy GroupBox a překrývá text v hlavičce zaškrtávacím polem, který obsahuje nastavený text a tvoří tedy také původní hlavičku. Při změně stavu zaškrtávacího políčka se pak projdou veškeré komponenty obsažené v GroupBox a aktivují se nebo deaktivují podle aktuálního stavu zaškrtávacího políčka.

Důvodem pro využití dědičnosti místo kompozice je v tomto případě fakt, že v rámci návrháře rozhraní Winforms v .NET 5[2] nejde zajistit, aby bylo možné do GroupBox obsaženého v UserControl přidávat komponenty. V .NET Framework 4.8 toto možné bylo a dá se tedy předpokládat, že jde o chybu návrháře v nové verzi .NET.

4.3.4 Horizontální seznam zaškrtávacích políček

V rámci některých obrazovek jsme potřebovali seznam zaškrtávacích políček pro výběr dní v týdnu. K tomu je možné využít komponentu CheckedListBox, která dělá přesně to, co potřebujeme. Její zásadní nevýhodou je, že zaškrtávací políčka jsou vždy pod sebou. To však způsobuje v našem případě s krátkými popiskami velké množství nevyužitého místa vedle této komponenty. Navíc nám seznam zabere poměrně hodně místa na výšku. Hodilo by se nám tedy místo ve sloupci mít zaškrtávací políčka v řádku. To však CheckedListBox neumožňuje a vzhledem k napojení na operační systém není jednoduché provést úpravu jeho rozvržení.

Rozhodli jsme se tedy vyvinout naši UserControl komponentu Horizontal-CheckedListBox zcela bez využití CheckedListBox. Použili jsme kontejner Flow-

[2] V době vývoje této komponenty šlo o nejnovější verzi .NET.

LayoutPanel, který nám umožní skládat jednotlivá zaškrtávací políčka vedle sebe s určenými mezerami a v případě nedostatku místa automaticky provede zalomení. Následně při přidání nové hodnoty vytvoříme zaškrtávací políčko s danou popiskou. Aby bylo rozhraní této komponenty alespoň částečně podobné CheckedListBox, využíváme pro přidávání hodnot ObservableCollection, která nám pomocí událostí umožňuje reagovat na přidávání a změny položek.

4.4 Zobrazování dat v tabulkách

K zobrazování dat v tabulce se v aplikaci vždy používá MultiLevelDataGridView (viz. část 4.3.1), tedy i v případech, kdy není potřeba vytvářet zanořené tabulky. Důvodem je, že jejich použití pomáhá vyřešit dva poměrně podstatné problémy:

1. Tabulka musí být dostatečně výkonná i při velkém množství řádků
2. Je potřeba umožnit jednoduchým způsobem předat data vnořeným tabulkám

Oba tyto problémy řeší třídy implementující rozhraní *IDataGridFiller* ve spolupráci právě s MultiLevelDataGridView. Pro každou entitu, která se v tabulkách zobrazuje, existuje jedna implementace tohoto rozhraní, která si z IDataManager načítá všechna potřebná data (může jít i o složení více různých entit - např. pozice potřebuje zobrazit název instrumentu) a následně data poskytuje MultiLevelDataGridView, které je tato instance IDataGridFiller předána.

To umožňuje řešit problém výkonnosti vykreslování, protože DataGridView používaná v MultiLevelDataGridView může mít nastavený tzv. virtuální mód, kdy tabulka data neobsahuje přímo, ale když je potřeba zobrazit další řádek, vyvolá se událost *CellValueNeeded*, která hodnotu pro určenou buňku tabulky dodá. MultiLevelDataGridView si v rámci této události voláním metody *GetCellValue* požádá IDataGridFiller, který má k dispozici, o potřebná data jako naformátovaný textový řetězec. Díky tomu není nutné mít všechna data v buňkách a provádět jejich vykreslení, pokud je není potřeba zobrazovat, což zrychluje odezvu na uživatelskou interakci. Vzhledem k tomu, že má DataGridFiller data již načtená, trvá přístup k datům daného řádku konstantní čas.

Typicky nedává smysl pro jeden typ entity ve vnořené tabulce zobrazovat více než jeden typ souvisejících entit. Díky tomu IDataGridFiller implementovaný pro danou entitu ví, kterou entitu má smysl zobrazovat ve vnořené tabulce. Řešením problému předávání dat vnořeným tabulkám je tedy požádat o ně IDataGridFiller, který máme k dispozici. Voláním metody *GetSubDataGridFillerAsync* tedy IDataGridFiller požádáme o vnořený IDataGridFiller ke zvolené entitě. Pokud ho dostaneme, můžeme zobrazit vnořenou tabulku a získaný IDataGridFiller jí předat. Může se také stát, že nám metoda nic nevrátí, což znamená, že pro tuto entitu nedává smysl zobrazovat vnořené tabulky nebo je to v tomto specifickém případě programátorem zakázáno (např. aby nedocházelo k moc hlubokému zanoření).

To, že získáváme data pro vnořené tabulky tímto způsobem, má navíc ještě další pozitivní efekt na výkon aplikace. Pokud by uživatel pořád dokola oteví-

ral a zavíral vnořenou tabulku pro jednu entitu, museli bychom pokaždé z úložiště získat data pro vnořenou tabulku. IDataGridFiller nám zde ale vytváří další vrstvu, v rámci které může být implementována mezipaměť. Místo toho, aby se pokaždé pro vnořenou tabulku této entity vytvářel nový IDataGridFiller a znovu se načítala jeho data, vrátí se již jednou vytvořený IDataGridFiller s načtenými daty.

4.5 Zobrazování dat v grafech

Při implementaci grafů a zobrazování dat v nich bylo potřeba vyřešit, jak umožnit různé kombinace typů zobrazení, typů datových řad a zdrojů. Jako přímočará implementace by se nabízelo zkrátka vytvářet pro každou možnou kombinaci zvláštní třídu, která by vše připravila a řekla grafu, co má zobrazit. Tento přístup by mohl být použitelný pro jednotky kombinací, ale už při malém množství jednotlivých typů a zdrojů by znamenalo velké množství kombinací a především kopírování kódu. Nemluvě o komplikovanosti přidání nového typu zobrazení nebo zdroje.

Rozhodli jsme se tedy jít jinou cestou a rozdělili jsme celý problém na tři části podle toho, co budeme potřebovat s čím kombinovat. Místo toho, aby jedna třída obsahovala celý proces od získávání dat, přes jejich zpracování po jejich zobrazení, budeme mít vždy alespoň tři třídy, kdy každá bude mít na starosti svou část procesu. Část pro zobrazení tak při své inicializaci získá třídu, která jí poskytne zpracovaná data původně získaná ze třídy, která ví, kde a jak je získat.

Tímto nám vznikla trojice rozhraní, které budou tyto třídy implementovat a budeme je tak moct libovolně kombinovat. Proces tedy začíná u třídy implementující generické rozhraní *IGraphDataSource*, která načte požadovaná data a poskytne je jako seznam struktur obsahujících dvojici datum a hodnotu stejného typu, jako je generický parametr rozhraní. Tento seznam si převezme třída implementující rozhraní *IGraphValueSource*, která data zpracuje a připraví konkrétní datové body. Tyto datové body umožňují také reprezentaci intervalu a obsahují kromě data i hodnoty počáteční, koncové, minimální a maximální. Nakonec si datové body převezme třída implementující rozhraní *IGraphSeriesProvider*, která připraví konkrétní datové řady knihovny OxyPlot s těmito datovými body. O tom, jaké zobrazení, jaké metriky a který zdroj se použije, nakonec rozhoduje to, které třídy implementující tato rozhraní nakonec vybereme.

Tato implementace nám navíc umožňuje v některých krocích zdroje slučovat či na sebe dále navazovat. Můžeme tedy například pro intervalové zobrazení využít IGraphValueSource, který bude data brát z jiného IGraphValueSource a upravovat je na intervalová. Podobně pokud chceme v grafu zobrazovat také značky transakcí, můžeme využít IGraphValueSource pro získání číselných hodnot a IGraphDataSource pro získání entit transakcí.

4.6 Stahování dat na pozadí

Stahování dat je důležitou součástí naší aplikace. Je prováděno pomocí úloh, které jsou implementovány tak, jak jsme si to naplánovali v rámci naší analýzy (viz. část 3.5). Pokud by ale zpracovávání probíhalo ve stejném vlákně jako zby-

tek aplikace, způsobovalo by to velké problémy s odezvou grafického rozhraní. O zpracovávání úloh se tedy stará třída *AsyncDataDownloader*, která s úlohami pracuje ve zvláštním vlákně.

Důležitou metodou třídy AsyncDataDownloader je metoda *Start*, která vytváří nové vlákno nebo probouzí již vytvořené vlákno. V rámci tohoto vlákna pak dochází k načtení úlohy, která má proběhnout jako první. Datum a čas příštího spuštění této úlohy (poslední spuštění navýšené o interval úlohy) je nejnižší ze všech aktivních úloh. Pokud již uběhl interval této úlohy, je provedena. V opačném případě je vlákno uspáno do doby, kdy interval úlohy uběhne. Může se také stát, že zmíněný požadavek žádnou úlohu nevrátí, protože žádné úlohy neexistují nebo jsou všechny deaktivované. V takovém případě se vlákno ukončí, protože je zbytečné, aby zabíralo systémové zdroje, když se automatické stahování dat nevyužívá.

Pokud by uživatel vytvořil nebo upravil některou z úloh, zavolá se opět metoda Start, a tedy se probudí nebo opět vytvoří vlákno pro zpracovávání úloh.

4.7 Synchronizace práce s daty ve více vláknech

Díky automatickému stahování dat na pozadí (viz. část 4.6) se nám může stát, že se stejnými daty budou pracovat v jednu chvíli dvě vlákna. AsyncDataDownloader je na toto připravený a tak si pro všechny entity, které se mu nesmí během provádění úlohy změnit, vytvoří editovatelnou kopii, která se při změně entity, ze které vznikla, neaktualizuje. Problém ale nastává, když má dojít k uložení dat. V tu chvíli by mohlo dojít k tomu, že zatímco jedno vlákno bude data ukládat, druhé je bude načítat a mohlo by tedy načíst polovinu nových a polovinu starých dat. Tuto situaci by mohl zkusit řešit přímo IDataManager, ale jelikož ho jakožto zásuvný modul může implementovat i někdo jiný než my, nemůžeme se spoléhat na to, že ochranu poskytne. Navíc nemusí jít vždy pouze o jeden požadavek, takže by IDataManager ani nebyl schopen problémům zabránit.

Budeme tedy tuto situaci řešit přímo v aplikaci. Zařídíme, že nikdy nebude probíhat více zápisů najednou a nebudou dvě části aplikace zároveň jedna zapisovat a druhá číst. Jelikož samotné čtení nevytváří žádné konflikty, nevadí nám, když bude z úložiště zároveň číst více částí aplikace. K tomu se nám hodí zámek typu *ReaderWriterLockSlim*, který právě umožňuje více vláknům zároveň číst, ale vždy pouze jednomu vláknu zapisovat a to jen ve chvíli, kdy žádné jiné vlákno nečte.

Aby zámek plnil svou funkci a v celé aplikaci bránil konfliktům, musí mít každá část aplikace k dispozici tu stejnou instanci. Z tohoto důvodu je zámek statickou vlastností rozhraní IPortEvalApp, ke kterému mají přístup všechny části aplikace. Výjimkou jsou zásuvné moduly, které používat tento zámek nepotřebují.

V souladu s oficiální dokumentací jsme se rozhodli využít doporučenou nerekurzivní variantu ReaderWriterLockSlim.[19] Nastávají však situace, kdy například načítání IDataGridFiller musí obsahovat zámek pro čtení, aby v případě, že by volající metoda zamknutí neprovedla, nedošlo k porušení nastavených pravidel. Pokud ale volající metoda zamknutí provedla, dostáváme se do situace, kdy provádíme rekurzivní zamčení. Řešením této situace a zároveň usnadněním používání zámků v takovýchto volaných metodách je statická metoda *Utils.ExecuteInLock*, která jako parametry dostává akci, která se má v zámku provést, a typ zámku,

ve kterém se má akce provést. Tato metoda ověří, jestli vlákno vlastní daný typ zámku nebo lepší (pokud požadujeme zámek pro čtení, zámek pro zápis nám také vyhovuje). Je-li to nutné, provede zamčení a po dokončení akce také odemknutí. V situaci, kdy dochází k pokusu o rekurzivní získání zámku, se zamykání neprovádí a je pouze spuštěna akce, která je parametrem metody.

4.8 Aktualizace zobrazovaných dat

Vzhledem k tomu, že se nám zobrazená data mohou měnit z jiných vláken a také z jiných oken, musíme být schopni na tuto situaci reagovat a data aktualizovat. Nabízí se nám několik možností, jak tohoto docílit, které se liší tím, jak proběhne oznámení toho, že došlo ke změně.

Mohli bychom vytvořit nový IDataManager, který bude pouze předávat požadavky opravdovému IDataManager ze zásuvného modulu a po provedení operace informuje všechna okna pomocí události. Tímto způsobem bychom měli jistotu, že k oznámení dojde opravdu vždy, ale je zde zásadní nevýhoda. Oznámení by totiž bylo vždy pouze pro jedinou entitu. Navíc by nebylo možné poznat, zda přijde ještě další oznámení, nebo bylo toto poslední. Obrazovky by tak musely zpracovávat každé oznámení samostatně, což by mohlo vést k značnému snížení výkonu. Problém by nastával především s obrazovkami obsahujícími seznam entit, u kterých je po přidání nebo odebrání entity potřeba provést přetřídění nebo výpočet.

Abychom tento problém s výkonem vyřešili, posuneme oznamování o úroveň výše. Jinými slovy, oznámení o změně bude odesílat ten, kdo změnu provádí a využívá IDataManager. Právě na této úrovni totiž víme, co všechno se ukládá a můžeme tedy pomocí události oznámit více entit najednou.

K oznámení změn entity využíváme událost *DataChanged*, která je součástí třídy PortEvalApp. Jelikož jde o využití globální události, která přežívá všechny obrazovky, je důležité, aby při zavírání obrazovky proběhlo odhlášení z této události. Jinak by docházelo k tomu, že by paměť obrazovky nemohla být nikdy uvolněna.

V případě zpracovávání oznámených změn je důležité, aby se změny zpracovaly ve stejném pořadí, v jakém byly provedeny. Vyvolání události se proto provádí ještě v rámci zámku jako poslední příkaz před odemčením a zpracovávání všech událostí následně probíhá v jednotlivých obrazovkách prostřednictvím DataUpdateQueue, která vykonání událostí naplánuje do vlákna grafického rozhraní a až po jejím dokončení spustí další.

4.9 Výpočet výkonnosti portfolií a pozic

Při investování je důležité být schopen zjistit, jak se jednotlivým portfoliím a pozicím v nich daří, a být schopen je porovnávat. Na orientační odhad by mohlo stačit podívat se na zisk, ale ten při srovnávání není vypovídající. Využijeme tedy pro měření výkonnosti metriku zvanou Internal Rate of Return (IRR). Jde o úrokovou sazbu investice, při které se investovaná hodnota rovná získané hodnotě. K jejímu výpočtu použijeme transakce provedené během sledovaného

období. Chceme tedy, aby se součet vkladů a zisků vynásobených touto úrokovou mírou při zohlednění času rovnal nule.[5]

Definice 1. *[5] Nechť A_i je hodnota vkladu i-té transakce a B_i je hodnota zisku i-té transakce. Mějme posloupnost $C_0, C_1, \dots C_n$, kde $C_i = A_i - B_i$ v časech $t_0, t_1, \dots t_n$. Pak IRR je úroková sazba $r = v - 1$ pro reálné v z rovnice $\sum_{i=0}^{n} C_i \cdot v^{t_i} = 0$.*

Pro nás bude užitečnější alternativní vzorec $\sum_{i=0}^{n} C_i \cdot v^{t_n - t_i} = 0$ [5]. Ten si dále upravíme, aby lépe odpovídal našim datovým strukturám. Dejme $C_i = T_i \cdot P_i$, kde T_i je velikost transakce i (množství nakupovaných nebo prodávaných jednotek) a P_i je cena jednotky transakce i. V našem případě typicky budeme počítat IRR pro neuzavřené investice, tedy budou ještě existovat neprodané jednotky. Poslední prvek sumy bude proto simulovat prodej zbylých jednotek za aktuální cenu v čase t_n. Provedeme jeho vyčlenění ze sumy a upravíme na $-C_a \sum_{i=0}^{n-1} T_i$, kde C_a je aktuální cena za jednotku instrumentu. Získáváme náš finální vzorec výkonnosti za jednotkové období

$$\sum_{i=0}^{n-1} C_i \cdot v^{t_n - t_i} = C_a \sum_{i=0}^{n-1} T_i.$$

Zajímat nás ale bude výkonnost za celé období a proto bude výsledná výkonnost $V = v^{t_n - t_0}$. Uživateli budeme chtít zobrazovat procentuální hodnotu, kterou získáme jako $(V - 1) \cdot 100$.

Zaměřme se nyní na samotnou realizaci výpočtu. Pravou stranu rovnice si můžeme spočítat jednou, protože není závislá na hodnotě výkonnosti. Levá strana rovnice obsahuje polynom vyššího stupně, takže k řešení využijeme aproximační algoritmus půlení intervalu s přesností na 4 desetinná místa tak, abychom získali výslednou procentní výkonnost s přesností na 2 desetinná místa.

4.10 Obrana proti zamrznutí grafického uživatelského rozhraní při načítání dat

Při práci s lokální databází jako je SQLite, kdy se data nemusí přenášet přes internet nebo alespoň místní síť, probíhá načítání dat rychle. Nemůžeme ale předpokládat, že to tak bude vždy, protože pomocí zásuvných modulů umožňujeme, aby byl zdroj v podstatě jakýkoliv. Pokud tedy tímto zdrojem bude například webové API, zvýší se latence všech dotazů minimálně na desítky až stovky milisekund. V případě většího vytížení sítě nebo větších požadavků bychom se pak mohli dostat až na sekundy. Nemluvě o tom, že výpočet výkonnosti může na slabších strojích také nějakou dobu trvat.

Pokud bychom tedy tyto operace prováděli ve vlákně, ve kterém je uživatelské rozhraní, docházelo by při delších operacích k tomu, že by rozhraní tzv. zamrzlo, tedy přestalo reagovat na interakce uživatele. Když nastane takováto situace, má uživatel pocit, že je s aplikací něco v nepořádku a měl by ji ukončit. Nám ale přitom jen o trochu déle trvalo získat nebo zpracovat data.

Řešení je tedy zřejmé. Načítání dat musí probíhat v samostatném vlákně. To však nestačí. Uživatel by si totiž i tak mohl myslet, že aplikace nefunguje, protože by před sebou měl sice reagující, ale prázdnou obrazovku.

Rozhodli jsme se tedy řešení těchto dvou problémů spojit a vytvořili *LoadingDialog*, který uživatele informuje o tom, že aplikace pracuje, a zároveň provádí na pozadí operaci, která mu byla předána při volání asynchronní statické metody *ShowLoading*. Po dokončení této operace se dialog zavře a zpátky ve vlákně uživatelského rozhraní může proběhnout naplnění komponent.

Jelikož operace načítání může být také velice krátká, bylo by neefektivní pro ni pokaždé vytvářet nové vlákno a nové dialogové okno. LoadingDialog proto vytváříme pouze při provádění prvního načítání. Ten následně ve zvláštním vlákně zpracovává frontu úloh plněnou statickou metodou ShowLoading. Metoda ShowLoading vrací *Task* a umožňuje tak vláknu grafického rozhraní počkat na výsledek načítání neblokujícím způsobem pomocí techniky async/await. Díky tomu nedojde k zablokování aplikačního cyklu a nemůže nastat situace, kdy by aplikace nereagovala na interakce.

4.11 Změna referenční měny

Pro přepočty hodnot v různých měnách používáme referenční měnu, ke které jsou hodnoty všech ostatních měn vztažené. Díky tomu nemusíme mít uložené hodnoty pro všechny možné převody, ale pro každou měnu pouze jeden. Zároveň to ale znamená, že referenční měnu musí určit uživatel a je na ní potenciálně navázané značné množství historických dat ostatních měn. To komplikuje situaci, kdy se uživatel rozhodne referenční měnu změnit.

Změna referenční měny znamená přepočítat kompletní historii všech měn. V případě měny, která se má stát novou referenční měnou, to není problém, protože zde pouze provedeme převrácení hodnoty všech záznamů.

Složitější je převod všech zbývajících měn. Zde totiž nemusíme mít dostupné hodnoty převáděné a nové referenční měny ve stejný datum a čas. Máme dvě možnosti, jak tuto situaci řešit. Ptát se vždy uživatele na odpovídající hodnotu nebo hodnoty aproximovat. Ani jedna z těchto variant není dokonalá. První varianta by v případě tisíců historických záznamů byla prakticky nerealizovatelná a druhá varianta by zase mohla způsobit výrazné zkreslení výsledků obzvláště v případě, kdy nejbližší hodnota nové referenční měny bude týdny či měsíce vzdálená.

Obě varianty mají své problémy, když se využijí každá zvlášť. Rozhodli jsme se je proto zkombinovat. Pokud máme záznamy dostatečně blízké, provedeme aproximaci. V opačném případě požádáme uživatele o asistenci. Výchozí povolené rozmezí mezi záznamy, do jejichž intervalu převáděná hodnota spadá, pro aproximaci je jeden den. Jelikož to může být pro uživatele příliš malý interval nebo by to mohlo znamenat zadávání příliš mnoha záznamů, může si uživatel při dotázání na převod nastavit počet povolených dní intervalu. Aby byl pro uživatele převod jednodušší, je dotazován vždy pouze na hodnotu staré referenční měny v nové referenční měně.

Pro provádění aproximace hodnoty jsme se rozhodli využít lineární aproximaci kvůli její jednoduchosti.

4.12 Zásuvné moduly

Zásuvné moduly jsou důležitou součástí aplikace, protože umožňují přidávání podpory různých druhů datových úložišť a možnosti stahování dat instrumentů a měn z různých zdrojů. Jejich velkou výhodou je, že díky nim mají vývojáři možnost poskytnout uživatelům další možnosti bez nutnosti zásahu do jádra aplikace.

Realizovány jsou zásuvné moduly pomocí dll knihoven, které se odkazují na naší knihovnu PortEval.API obsahující všechna potřebná rozhraní a třídy pro vývoj zásuvných modulů. Aplikace dll knihovny vyhledává v podadresářích pojmenovaných podle jejich typu a hledá v nich třídu implementující rozhraní *IPlugin*, resp. jeho variantu pro daný typ zásuvného modulu. Tato třída pak slouží jako vstupní bod do zásuvného modulu a k jeho funkcím. Kromě toho umožňuje aplikaci také získávání základních informací o zásuvném modulu jako je jeho identifikátor a jméno. Identifikátor musí být mezi zásuvnými moduly daného typu vždy unikátní. Jméno je používáno v uživatelském rozhraní, kdykoliv je potřeba uživateli zobrazit seznam zásuvných modulů.

V rámci adresáře pro zásuvné moduly vybraného typu má každý zásuvný modul svůj adresář pojmenovaný stejně jako je jeho název sestavení (assembly). V tomto adresáři se pomocí informací v souboru se jménem složeným z názvu sestavení a přípony „.deps.json“ načte dll zásuvného modulu a případné další knihovny, na kterých zásuvný modul závisí.

Je-li potřeba zásuvnému modulu předat jeho konfiguraci, činí se tak prostřednictvím rozhraní *IConfig*, které umožňuje zásuvnému modulu získávat a ukládat své nastavení ve formátu JSON. Destinace, se kterou se bude při načítání a ukládání pracovat, záleží na předané implementaci rozhraní.

4.12.1 Zásuvný modul pro práci s úložištěm

Zásuvné moduly pro práci s datovými úložišti se nachází v podadresáři lib/db/ a musí obsahovat třídu implementující rozhraní *IDatabasePlugin*. Pomocí tohoto rozhraní si aplikace může vyžádat UserControl implementující rozhraní *IPluginSettingsControl* sloužící ke konfiguraci zásuvného modulu pomocí uživatelského rozhraní. Typicky jde o nastavení cesty k datovému úložišti a údaje potřebné k získání přístupu. Především si ale aplikace pomocí IDatabasePlugin může vyžádat instanci třídy implementující IEditableDataManager, kterou následně bude využívat pro práci s datovým úložištěm.

IEditableDataManager obsahuje 5 typů metod: obslužné, pro přihlášení nebo změnu přihlašovacích údajů uživatele, načítací, ukládací a mazací.

Obslužné metody umožňují aplikaci volat některé funkce datového úložiště. Jde o metody *ExecuteInTransaction* a *CleanUpDatabase*. První ze zmíněných metod, ExecuteInTransaction, zajistí, že veškerá volání tohoto IEditableDataManager v parametrem předaném delegátu se budou chovat jako jedna databázová transakce, tedy selhání provádění jednoho příkazu znamená neprovedení žádného příkazu této transakce. Metoda CleanUpDatabase má za úkol úložiště pročistit a zoptimalizovat (v případě SQLite voláním funkce „Vacuum“). Potřeba této metody plyne z možného vzniku fragmentace úložiště při automatickém stahování dat (viz. část 3.5.3).

Metody pro přihlášení a změnu přihlašovacích údajů uživatele jsou *LoginUser*

a *ChangeCredentials*. Jejich úkolem je využít předané přihlašovací údaje a pokusit se s nimi uživatele přihlásit, resp. uložit jeho nové přihlašovací údaje. V případě, že se přihlášení nepovede, protože přihlašovací údaje nejsou správné, vyhodí metody výjimku typu *IncorrectCredentialsException* a aplikace zobrazí informaci uživateli. Z bezpečnostních důvodů by v případě, že to úložiště umožňuje, do aplikace nikdy nemělo být heslo načteno. Ověření správnosti hesla by mělo probíhat na úrovní databáze. Míru zabezpečení hesla určuje zásuvný modul.

Metody sloužící k načítání dat vždy vrací seznam entit splňujících požadavky a jsou specifické pro daný typ entity. Při načítání entity je nutné aplikovat veškeré filtry definované třídou *Filters* dané entity. Aby mohla být entita do výsledku zařazena, musí být splněny všechny vyplněné podmínky. Následně je potřeba seřadit entity podle požadovaného kritéria určeného výčtovým typem *OrderBy* dané entity. Počet entit může být omezený parametrem. V takovém případě metody vrací pouze prvních n entit.

Ukládací metody přijímají dva parametry. Prvním je entita, která se má uložit, a druhým jsou její data, která se mají uložit. Tyto metody slouží k ukládání nových i existujících entit. V případě, že se identifikátor entity rovná hodnotě *Entity.NEW_ENTITY_ID*, jde o novou doposud neexistující entitu. Po jejím vytvoření metoda vrací přidělený identifikátor. Pokud jde o již existující entitu, dochází pouze k aktualizaci dat v úložišti a metoda vrací identifikátor entity z parametru volání metody.

Nakonec, mazací metody smažou entitu podle identifikátoru z úložiště. Aplikace sama provádí nejprve smazání všech závislých entit a nikdy by tedy nemělo dojít k porušení vazeb. Zároveň aplikace předpokládá, že opravdu dochází k smazání entity a nesnaží se tedy nikdy obnovovat smazané entity s tím, že by předtím byly pouze skryty.

V případě, že dojde při komunikaci s úložištěm k nějakému problému nebo operace skončí chybou, měl by IEditableDataManager vyvolat výjimku libovolného typu. Její zpráva bude zobrazena uživateli. Pro zjednodušení implementace zásuvného modulu není potřeba výjimky ve všech metodách odchytávat a zabalovat do určitého typu výjimky.

4.12.2 Zásuvný modul pro stahování dat

Zásuvné moduly pro stahování dat se načítají z podadresáře lib/downloader/. Aby byla knihovna správně rozpoznána a mohla fungovat jako zásuvný modul pro stahování dat z externího zdroje, musí obsahovat třídu implementující rozhraní *IDownloaderPlugin*. Kromě informací společných pro všechny zásuvné moduly musí IDownloaderPlugin poskytovat také pole *SupportedEntityTypes* obsahující všechny typy entit, pro které je modul schopen stahovat data.

Podobně jako v případě zásuvných modulů pro práci s úložištěm probíhá konfigurace prostřednictvím UserControl implementující rozhraní IPluginSettingsControl. Tentokrát jsou zde ovšem dvě varianty konfigurace, globální konfigurace zásuvného modulu a konfigurace specifická pro danou úlohu. V globální konfiguraci najdeme typicky klíč pro přístup k webovému API. Při získávání komponenty pro konfiguraci konkrétní úlohy se zásuvnému modulu předává také globální konfigurace a informace o tom, který typ entity se bude pomocí konfigurované úlohy stahovat. To zásuvnému modulu umožňuje zobrazit jinou konfiguraci pro úlohy

stahující kurzy měn a pro úlohy stahující cenu instrumentů.

Samotné stahování dat provádí třída implementující rozhraní IDataDownloader. Toto rozhraní obsahuje jedinou metodu *DownloadValue*, která vrací získanou hodnotu, nebo popřípadě vyhazuje výjimku (viz. část 3.5). Instance třídy implementující IDataDownloader se získává voláním metody *GetDataDownloader* zásuvného modulu, které se kromě globální konfigurace zásuvného modulu předává také typ entity, pro kterou se budou data pomocí této instance stahovat. Tím je zásuvnému modulu umožněno rozdělit implementaci IDataDownloader tak, aby byla specifická pro jednotlivé typy entit a bylo tak možné provést lepší dekompozici.

Aplikace žádá zásuvný modul o třídu implementující IDataDownloader na začátku zpracovávání každé úlohy znovu, což umožňuje zásuvnému modulu rozpoznat, že se zahajuje zpracování nové úlohy. Pokud externí zdroj dat není schopen poskytovat údaje pro jeden specifický datum a čas a je tedy nezbytné stahovat více hodnot najednou, měla by implementace IDataDownloader ukládat všechny stažené údaje do mezipaměti, aby se stahování jednoho údaje neprovádělo vícekrát. V takovém případě je také vhodné, aby se při každém volání metody GetDataDownloader vracela nová instance (popř. bylo provedeno smazání mezipaměti) a nedocházelo tak k zaplnění operační paměti hodnotami, které již nebudou nikdy potřeba.

Pokud zdroj dat umožňuje vyhledávání instrumentů, může třída implementující rozhraní IDownloaderPlugin implementovat také rozhraní *IInstrumentImporterProvider*. Tím dává aplikaci vědět, že je možné zásuvný modul použít pro import instrumentů. Rozhraní následně umožňuje aplikaci získat třídu implementující rozhraní *IInstrumentImporter*, která prostřednictvím metody *SearchImportableInstrumentsAsync* poskytuje seznam dostupných instrumentů na základě vyhledávacího dotazu a pomocí metody *GetInstrumentTaskSettings* generuje nastavení úlohy pro stahování dat vybraného instrumentu specifické pro daný zásuvný modul.

5. Uživatelské rozhraní

Uživatelské rozhraní je rozdělené na dvě části, boční panel v levé části a zvolenou obrazovku ve zbytku okna. Boční panel slouží k přepínání mezi jednotlivými obrazovkami aplikace.

5.1 První spuštění

Po instalaci aplikace (viz. příloha A.1) můžeme aplikaci spustit pomocí souboru *PortEval.exe*. Abychom mohli aplikaci začít plně využívat, musíme provést počáteční konfiguraci. Jako první nám aplikace nabídne obrazovku s přihlášením. V pravém horním rohu obrazovky najdeme tlačítko *Settings*, po jehož stisknutí se nám otevře dialog pro konfiguraci úložiště. Konfigurace se liší podle použitého úložiště, ale typicky jde o cestu a případně přihlašovací údaje.

Po nakonfigurování úložiště se můžeme přihlásit pomocí výchozího uživatelského jména *admin* a hesla *admin*. Přihlašovací údaje je možné později změnit v nastavení (viz. část 5.8).

Jelikož na sebe jednotlivé části aplikace navazují, musíme první záznamy vytvořit v určitém pořadí. Začneme vytvořením měny (viz. část 5.3), kterou budeme chtít využívat jako referenční (hodnoty ostatních měn budou v této měně). Po jejím vytvoření ji nastavíme jako referenční měnu v nastavení (viz. část 5.8). Nyní už máme vše potřebné pro vytváření dalších měn a zakládání portfolií (viz. část 5.6). Zatím ale nemáme čím portfolia naplnit. Začneme tedy zadávat informace o trzích (viz. část 5.4) a následně instrumentech (viz. část 5.5), které se na nich obchodují.

Tímto jsme si připravili vše, co potřebujeme, abychom mohli začít aplikaci plnohodnotně využívat.

5.2 Časté společné znaky různých obrazovek

Většina obrazovek v aplikaci se dá zařadit mezi dva druhy, seznamy a editace.

Seznamové obrazovky

Seznamové obrazovky zobrazují typicky tabulku všech vytvořených záznamů daného typu a umožňují přechod na editační obrazovky, které slouží k zakládání nových záznamů (přechod pomocí tlačítka „New“) nebo zobrazení detailů záznamu a provádění jeho úprav (typicky se zobrazí po stisknutí tlačítka „Detail“ nebo „Edit“). Dvojité kliknutí na záznam v tabulce většinou funguje stejně jako stisknutí tlačítka „Detail“. Posledním tlačítkem, které se obvykle na těchto obrazovkách nachází, je tlačítko s popiskou „Delete“ umožňující permanentní smazání vybraného záznamu. Při použití pravého tlačítka myši na řádek tabulky se zobrazí kontextová nabídka, která v některých případech umožňuje provádět se záznamem také další akce, které není možné provést pomocí tlačítek. Kliknutím na záhlaví sloupce tabulky je možné provést přetřídění dat podle tohoto sloupce.

Editační obrazovky

Editační obrazovky se na rozdíl od těch seznamových zaměřují na jeden vybraný záznam. V horní části těchto obrazovek jsou editovatelné údaje daného záznamu. Ve zbylé části obrazovky v některých případech bývá tabulka s editovatelným seznamem souvisejících záznamů. V pravé dolní části obrazovky se pak nachází typicky dvě tlačítka, „Save“ pro uložení provedených úprav a „Cancel“ pro opuštění obrazovky bez ukládání změn. Stisknutí klávesy „Escape“ funguje u těchto obrazovek stejně jako použití tlačítka „Cancel“.

5.3 Měny

Obrazovka na obrázku 5.1 zobrazující seznam měn se zobrazí po stisku tlačítka „Currencies“ v levém bočním panelu. Obrazovka obsahuje tabulku se seznamem všech již založených měn a jejich aktuální kurz vůči referenční měně (nastavení referenční měny viz. část 5.8), která má vždy hodnotu 1,0. Tlačítko „Show graph“ umožňuje zobrazit graf vybrané měny v novém okně. Chování zbylých obslužných tlačítek se neliší od podobných obrazovek (viz. část 5.2).

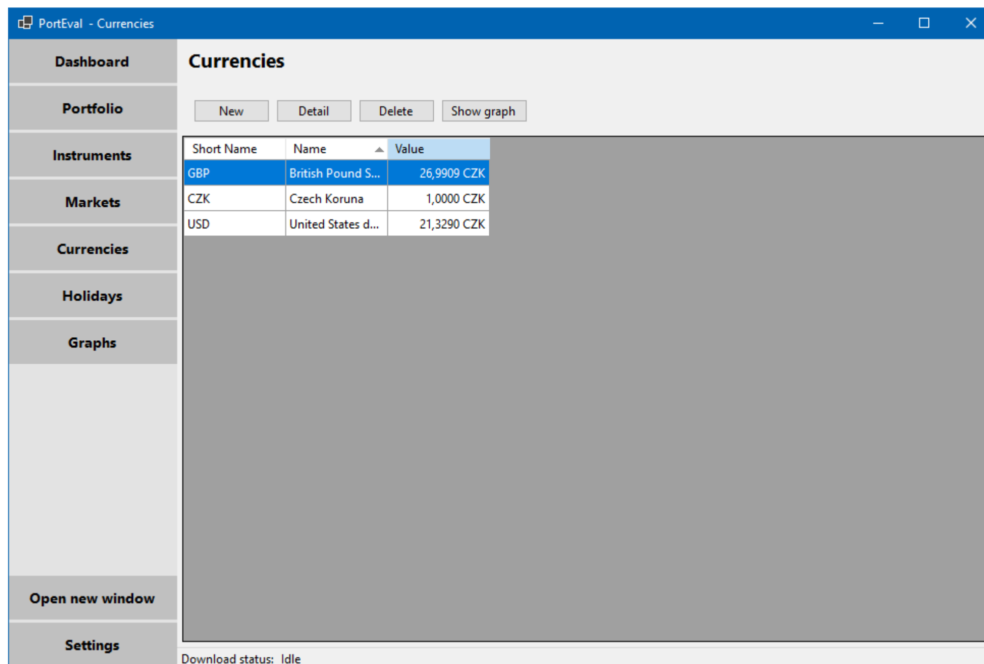
Editační obrazovka měny na obrázku 5.2 poskytuje možnost upravit základní informace o měně, jako je její jméno či zkratka. Lze také nastavit pro tuto měnu jiné než výchozí časové rozmezí a dny, kdy se s měnou obchoduje (údaje se využívají při automatickém stahování dat). Pod těmito informacemi je zobrazena tabulka s historií kurzu zobrazené měny.

Tabulka s historií kurzu obsahuje veškeré záznamy o vývoji kurzu měny seřazené od nejnovějšího po nejstarší. Záznamy v tabulce jsou dvojího typu (zobrazený ve sloupci „Type“), manuálně zadané (označené „M“), nebo automaticky stažené (označené „D“). Upravovat je možné pouze manuálně zadané záznamy. Vybráním určitého řádku v tabulce se vyplní pole „Date and time“ a „Value“. Údaje je možné modifikovat a přepsat jimi původně uložený záznam stiskem tlačítka „Modify“. Tlačítko „Add new“ umožňuje místo přepsání existujícího záznamu využít tyto hodnoty pro založení nového záznamu. Vybraný záznam v tabulce je možné také vymazat pomocí tlačítka„Delete“ bez ohledu na jeho typ.

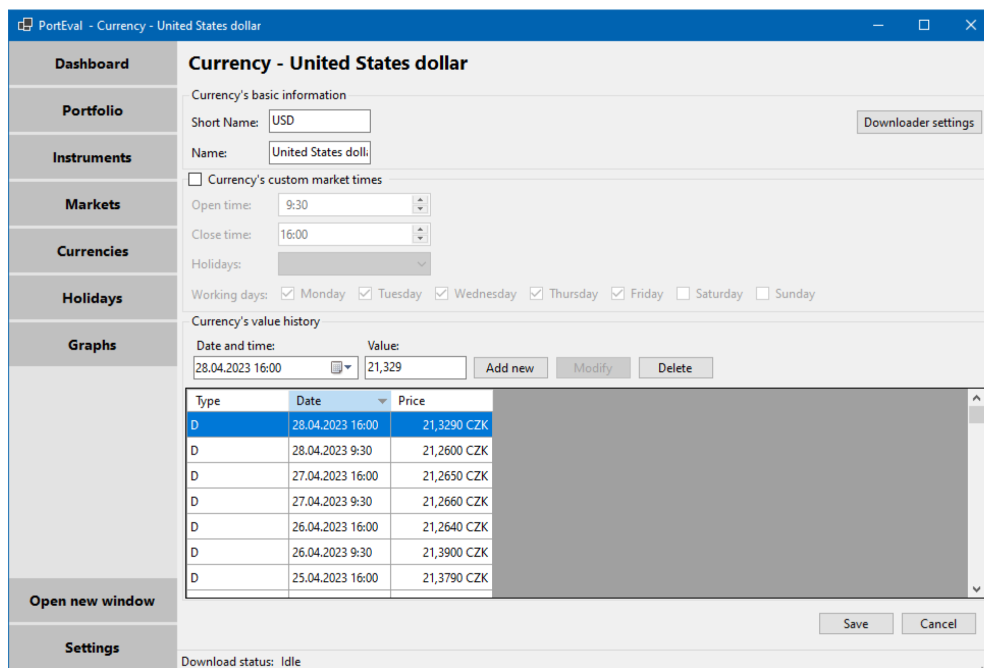
5.4 Trhy

Veškeré instrumenty jsou v aplikaci vždy vázané k nějakému trhu, na kterém se s nimi obchoduje. Zadané informace o trhu se následně využívají při automatickém stahování dat. Seznam trhů evidovaných v aplikaci je možné zobrazit v rámci obrazovky na obrázku 5.3 stisknutím tlačítka „Markets“ v levém bočním panelu. Obslužná tlačítka zde fungují stejně jako u většiny ostatních seznamových obrazovek (viz. část 5.2).

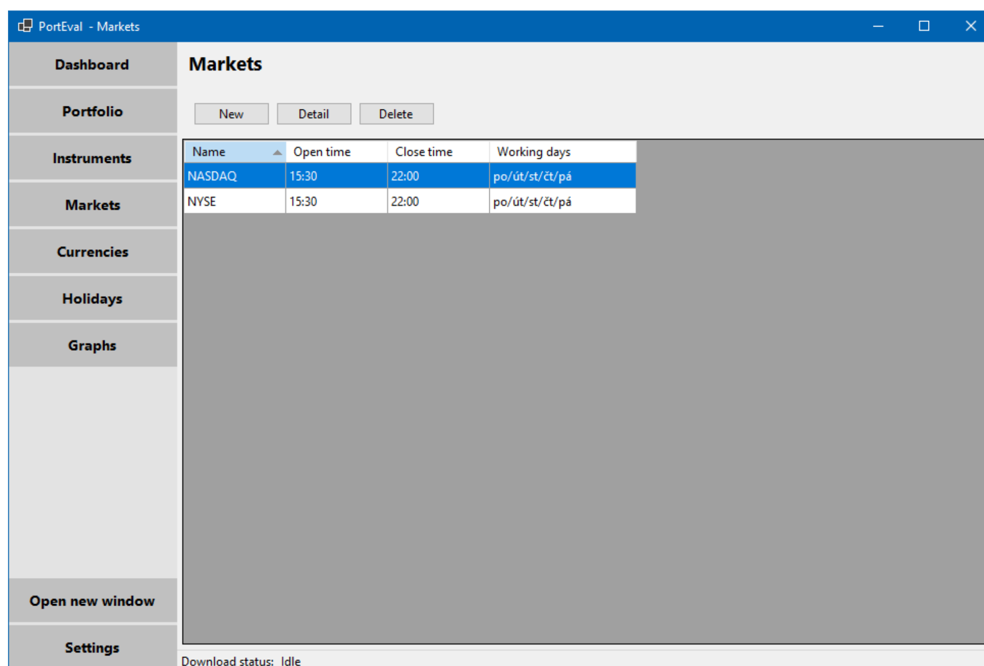
Editační obrazovka z obrázku 5.4 umožňuje změnit název trhu, jeho časovou zónu a s tím související čas začátku a konce obchodování daného dne ve zvolené časové zóně. Jelikož se v aplikaci používá lokální časová zóna, či případně obě časové zóny, je vedle polí pro zadávání času ve zvolené časové zóně také přepočet na čas v lokální časové zóně. Dále je možné určit, které dny v týdnu vůči zvolené časové zóně se na trhu obchoduje. Protože trhy typicky některé dny v roce neobchodují, je možné zvolit seznam těchto výjimečných dnů (viz. část 5.7).



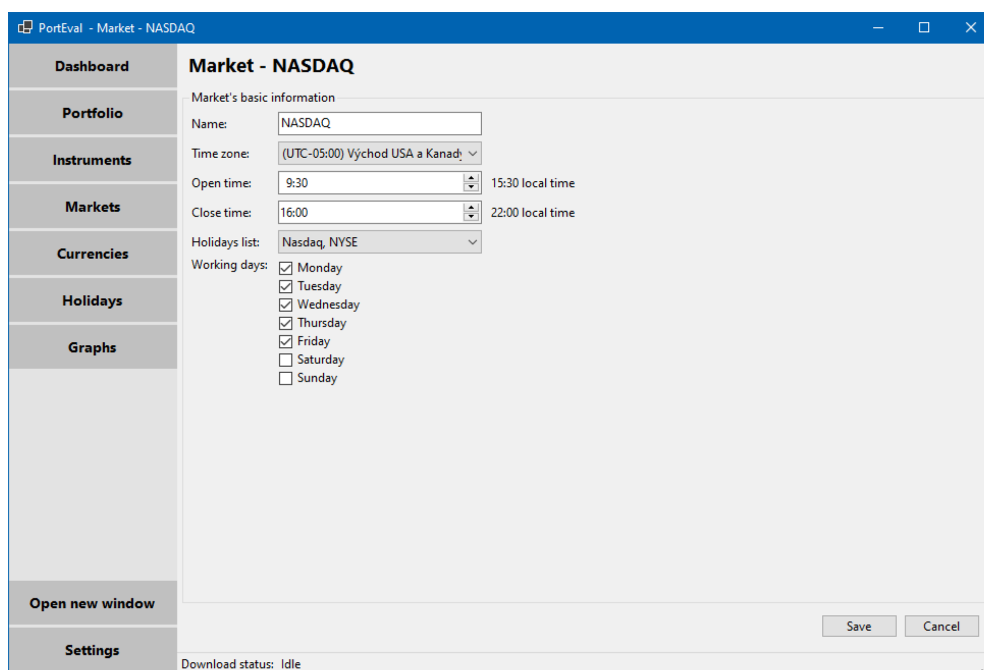
Obrázek 5.1: Obrazovka zobrazující seznam vytvořených měn.



Obrázek 5.2: Obrazovka zobrazující detail měny a jejího kurzu s možností editace.



Obrázek 5.3: Obrazovka zobrazující seznam evidovaných trhů.



Obrázek 5.4: Editační obrazovka trhu. Čas otevření a zavření trhu se zadává v časové zóně trhu.

Obrázek 5.5: Obrazovka se seznamem sledovaných instrumentů

Pro správné fungování automatického stahování dat je důležité správně vyplnit veškeré informace o trhu a v případě, že trh neobchoduje každý den v roce, nastavit také seznam výjimečných dnů. V opačném případě by se aplikace mohla pokoušet stahovat údaje, které nebude nikdy možné získat a plýtvala by tak zbytečně typicky omezeným počtem přístupů k externímu zdroji, což by mohlo mít za následek, že nakonec nebude možné údaje stáhnout.

5.5 Instrumenty

Pokud existuje alespoň jedna měna a jeden trh, je možné vytvářet v aplikaci instrumenty. Seznam všech instrumentů tak, jak je vidět na obrázku 5.5, se zobrazí po stisknutí tlačítka „Instruments“ v levém bočním panelu aplikace. Obrazovka obsahuje tabulku se seznamem všech instrumentů seřazených podle názvu. Kromě názvu, měny instrumentu a trhu, obsahuje tabulka také poslední známou hodnotu tohoto instrumentu. Chování obrazovky a jejích tlačítek je stejné jako u většiny seznamových obrazovek (viz. část 5.2) až na tlačítka „Show graph“, které po stisknutí v novém okně zobrazí graf vybraného instrumentu, a „Import“.

Tlačítko Import otevírá dialogové okno na obrázku 5.6, ve kterém je možné vyhledat nový instrument pomocí jednoho ze zdrojů dat. Během vyhledávání se zadávají také měna a trh, které zásuvný modul může podle svých možností využít k filtrování výsledků. Následné přidání tohoto instrumentu vytvoří nejen samotný instrument s vazbami na měnu a trh, ale podle zvolených zaškrtávacích polí se také vygenerují úlohy pro stahování dat pomocí vybraného zdroje dat.

Obrazovka pro editaci instrumentu (viz. obrázek 5.7) umožňuje změnit základní údaje a prohlížet si a upravovat historii hodnoty instrumentu v tabulce v druhé části obrazovky. Většinu základních informací o instrumentu je možné kdykoliv změnit. Výjimkou je měna instrumentu, kterou lze změnit pouze v pří-

padě, že v historii hodnoty instrumentu není žádný záznam.

Tabulka s historií hodnoty instrumentu se podobá tabulce s historií kurzu měny z části 5.3. Každý záznam v historii hodnoty instrumentu má jeden ze tří typů podle toho, jak tento záznam vznikl. Tato informace je zobrazena ve sloupci „Type“. Může jít o vznik manuálním zadáním (označen „M“), na základě ceny instrumentu při transakci (označen „T“) nebo pomocí automatického stahování dat (označen „D“). Vytvářet a upravovat je možné pouze manuálně přidané záznamy. Mazat je možné také stažené záznamy. Vybráním řádku v tabulce historie hodnoty instrumentu se vyplní pole „Date and time“ a „Price“ hodnotami vybrané položky historie. Stiskem tlačítka „Modify“ se hodnoty vybraného řádku přepíší hodnotami v těchto polích. Stisknutí tlačítka „Add new“ přidá hodnoty z polí jako nový záznam bez modifikace vybraného. Tlačítko „Delete“ umožňuje smazat vybraný záznam z historie.

5.6 Portfolia

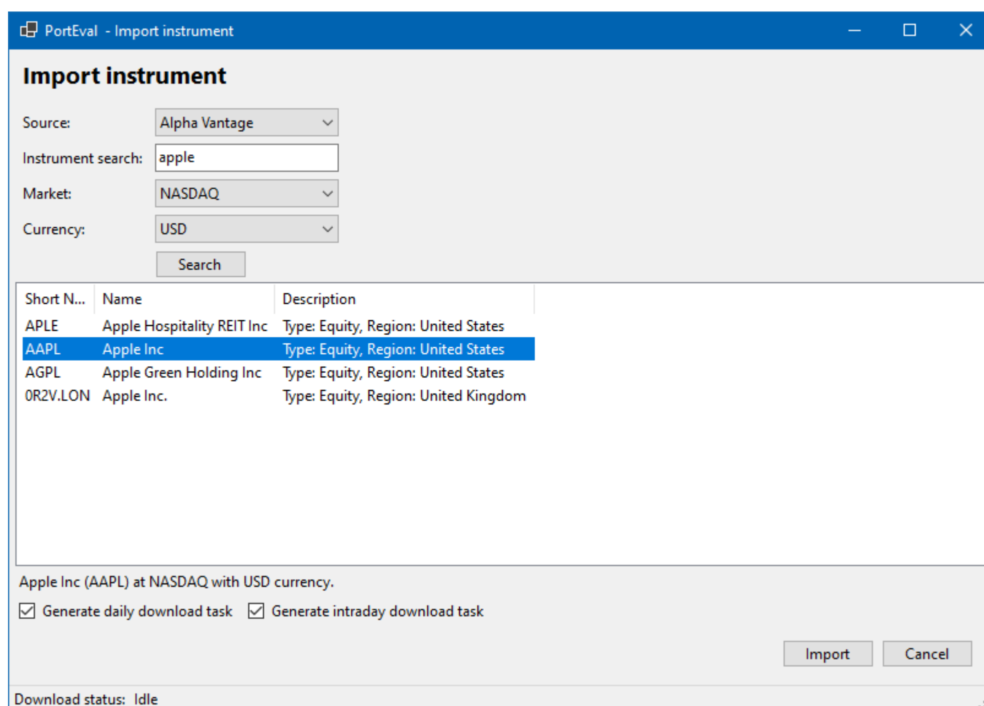
Hlavní částí aplikace jsou investiční portfolia. Jejich seznam se otevře stisknutím tlačítka „Portfolio“ v levém panelu. Obrazovka (viz. obrázek 5.8) obsahuje tabulku se všemi investičními portfolii, jejich celkovým ziskem, změnou zisku za určitá období, celkovou výkonností a změnou výkonnosti za určitá období. Aby bylo možné jednoduchým způsobem do portfolia nahlédnout, zobrazí se po kliknutí na řádek s portfoliem vnořená tabulka se seznamem všech pozic instrumentů obchodovaných v tomto portfoliu. Každý řádek pozice pak obsahuje informace o zisku a výkonnosti, aktuální množství jednotek v pozici, cenu instrumentu při nulovém zisku této pozice a aktuální cenu instrumentu. Chování tlačítek nad tabulkou se neliší od většiny seznamových obrazovek (viz. část 5.2).

Editační obrazovka (viz. obrázek 5.9) obsahuje název portfolia, volbu měny a také možnost přidání víceřádkového komentáře. Provést změnu měny portfolia je možné, pokud není do portfolia přidána žádná transakce. Ve zbylé části obrazovky se pak nachází tabulka s pozicemi a transakcemi zobrazeného portfolia.

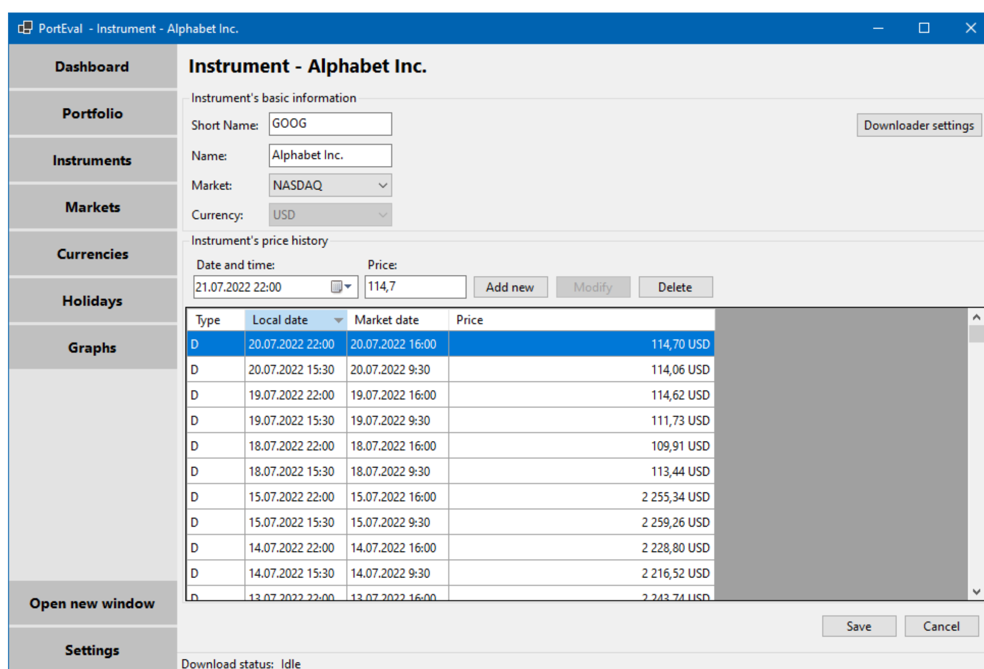
Veškeré transakce jsou seskupené podle pozice pro daný instrument. Stejně jako v případě tabulky portfolií je možné kliknutím na řádek pozice zobrazit vnořenou tabulku všech transakcí této pozice. Modifikace transakce se provádí podobným způsobem jako v případě historie hodnoty instrumentů nebo kurzu měny. Tedy vybráním transakce ve vnořené tabulce, úpravou hodnot v polích pro úpravu transakce („Instrument“, „Date“, „Amount“, „Price“ a „Note“) a stiskem tlačítka „Modify“. Při editaci je možné změnit instrument transakce. V takovém případě dojde k jejímu přenosu z jedné pozice do druhé. Stiskem tlačítka „Delete“ je možné vybranou transakci smazat.

To, jestli se pozice v tabulce zobrazí, záleží na tom, zda pro ni v portfoliu existuje transakce. Pro přidání nové pozice či transakce k existující pozici tedy stačí vyplnit pole pro úpravu transakce a stisknout tlačítko „Add new“. V závislosti na volbě instrumentu se následně vytvoří nová pozice, nebo se transakce přidá k již existující pozici pro daný instrument.

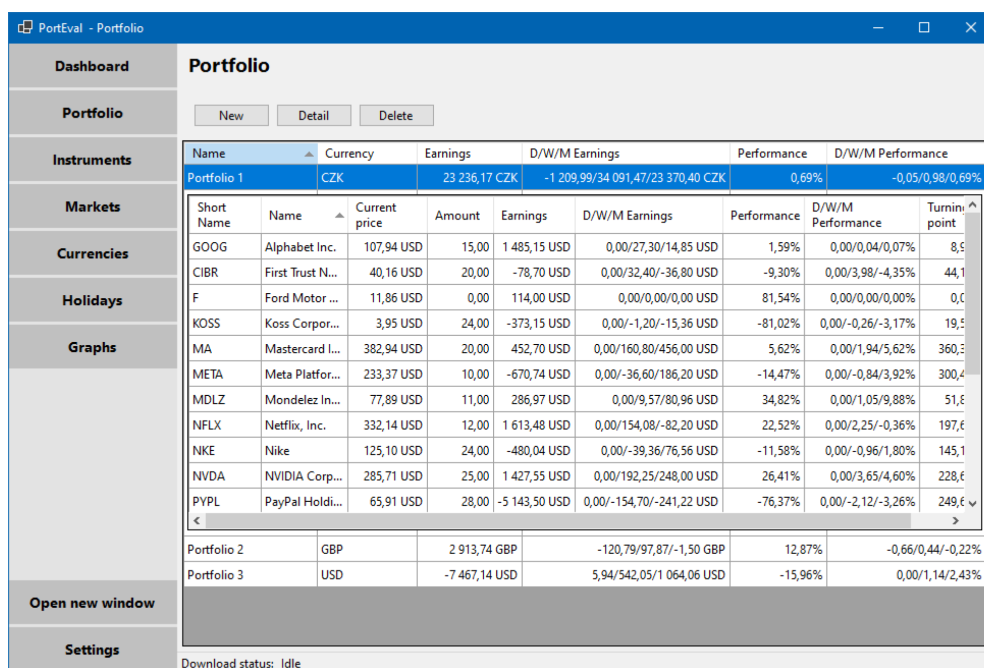
Každé přidání, úprava a smazání transakce způsobí okamžité přepočítání všech hodnot pro ovlivněnou pozici. Aplikace kontroluje, že pozice nemá v žádném časovém okamžiku záporné množství, a při změně instrumentu nebo data transakce načítá nejnovější dostupnou hodnotu před tímto datem. Transakce obsahují



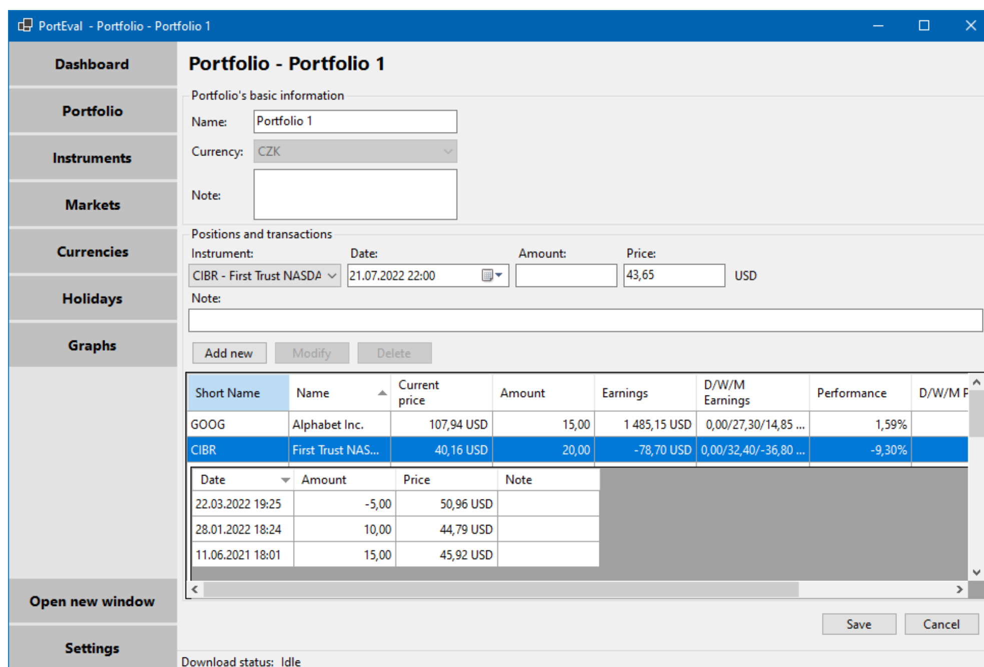
Obrázek 5.6: Obrazovka pro import instrumentů



Obrázek 5.7: Editační obrazovka instrumentu a historie vývoje jeho hodnoty. Měnu instrumentu je možné měnit pouze, pokud se v historii hodnoty nenachází žádný záznam.



Obrázek 5.8: Obrazovka se seznamem portfolií a rozbalitelnými seznamy pozic v rámci jednotlivých portfolií.



Obrázek 5.9: Editační obrazovka portfolia, která umožňuje přidávat, upravovat a mazat transakce provedené v rámci tohoto portfolia.

pole pro doplnění poznámky. To může být vhodné obzvláště pro poznamenání okolností nákupu nebo prodeje pro pozdější prohlížení.

5.7 Skupiny obchodních prázdnin

Téměř na každém trhu jsou během roku dny, kdy je trh zavřený a neobchoduje se. Pokoušet se stahovat v tyto dny z externích zdrojů údaje by bylo pouze plýtváním případného limitu počtu přístupů k tomuto zdroji, protože by žádné údaje stejně nebylo možné stáhnout. Z tohoto důvodu umožňuje aplikace vytvořit skupiny dní, během kterých se neobchoduje. Tyto skupiny je následně možné vybrat u trhů či měn.

Seznam všech skupin se zobrazí stiskem tlačítka „Holidays“ v levém bočním panelu aplikace. Tabulka (viz. obrázek 5.10) zobrazuje o všech skupinách jméno, uživatelskou poznámku a počet dní v této skupině. Po kliknutí na řádek tabulky se zobrazí seznam dní ve vybrané skupině. Chování tlačítek je stejné jako u většiny ostatních seznamových obrazovek (viz. část 5.2).

Obrazovka pro úpravu skupiny (viz. obrázek 5.11) umožňuje editovat základní údaje o skupině a obsahuje tabulku se všemi výjimečnými dny, kdy se neobchoduje. Tak, jako u podobných obrazovek s instrumenty, měnami nebo transakcemi, se kliknutím na řádek tabulky vybrané údaje vyplní do polí „Name“ a „Date“, kde je možné je upravit. Potvrzení úpravy a přepsání původního záznamu se provádí pomocí tlačítka „Modify“. Stisknutí tlačítka „Add new“ přidá zadané údaje jako nový záznam. Vybraný řádek v tabulce je možné také smazat tlačítkem „Delete“.

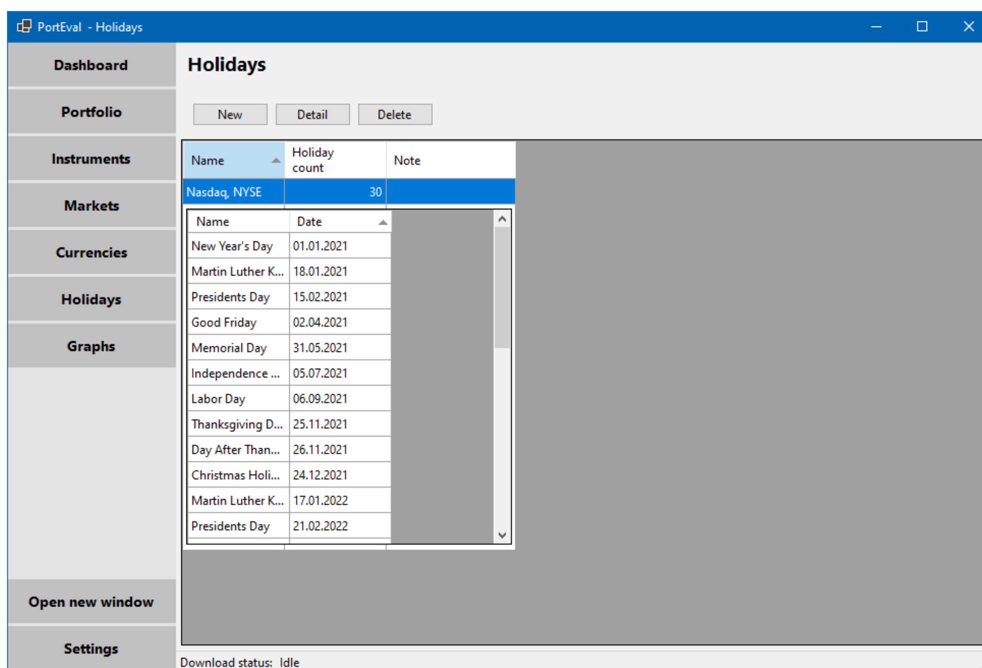
Data výjimečných dnů, během kterých se na trhu neobchoduje, bývají typicky každý rok stejná nebo téměř stejná. Z tohoto důvodu je možné pomocí tlačítka *Clone previous year* zkopírovat všechny záznamy z minulého roku do aktuálního roku a následně pouze provést úpravy několika málo změn pro aktuální rok.

5.8 Nastavení

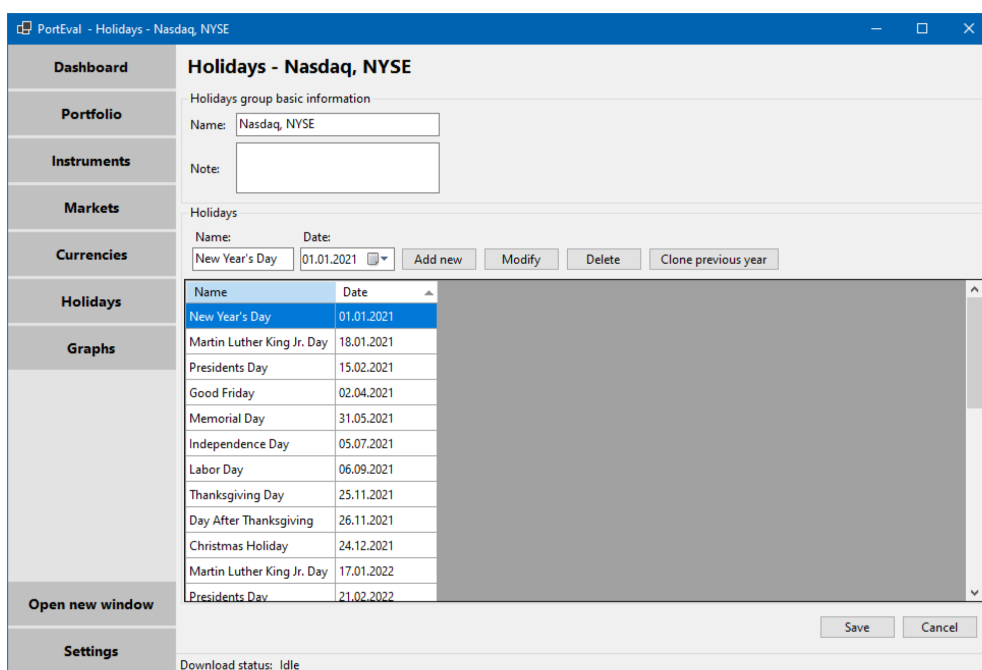
Globální nastavení aplikace lze otevřít stisknutím tlačítka *Settings* v dolní části levého bočního panelu. Tato obrazovka (viz. obrázek 5.12) slouží ke konfiguraci těch parametrů, které ovlivňují větší část aplikace. Nastavuje se zde referenční měna, která se nadále používá, kdykoliv je potřeba provést převod ceny z jedné měny na druhou. Volbu referenční měny je důležité dobře zvážit, protože její změna může být obzvláště v případě existence většího množství měn nebo bez automatického stahování kurzu měn, komplikovanou operací.

Položka „Download history only after“ určuje první datum, pro které se pomocí automatického stahování dat aplikace bude pokoušet stáhnout data. Automatické stahování tedy probíhá od tohoto data včetně do současnosti. Historie instrumentů a měn před tímto datem je algoritmem pro automatické stahování dat zcela ignorována.

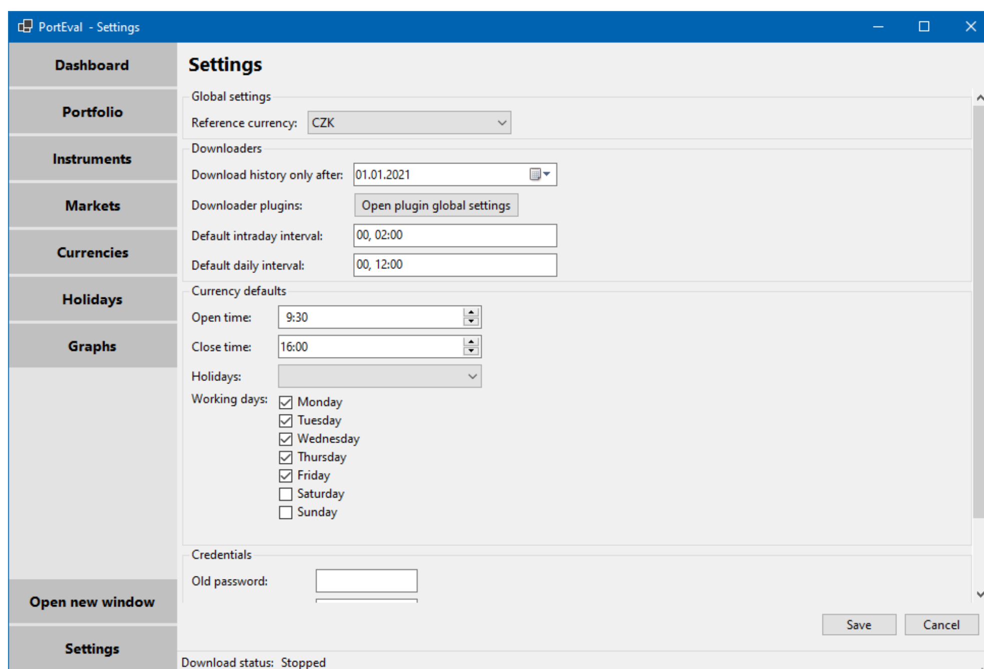
Tlačítko „Open plugin global settings“ otevírá nové dialogové okno, ve kterém je možné konfigurovat jednotlivé zásuvné moduly pro automatické stahování dat na úrovni celé aplikace. Typickými nastaveními jsou například API klíč nebo identifikátor referenční měny pro tento externí zdroj dat. Dialog se skládá z levého



Obrázek 5.10: Obrazovka se seznamem skupin dní, během kterých se neobchoduje, s rozbalovacími seznamy jednotlivých dní v rámci skupin.



Obrázek 5.11: Editační obrazovka skupiny obchodních prázdnin s možností přidávat, upravovat a mazat jednotlivé volné dny, případně zkopírovat všechny z předchozího roku.



Obrázek 5.12: Obrazovka s globálními nastaveními aplikace jako jsou výchozí údaje o měnách, volba referenční měny, konfigurace zásuvných modulů a změna přihlašovacích údajů.

bočního panelu se seznamem všech zásuvných modulů a zbytku obrazovky, ve které se nachází konfigurace vybraného modulu.

V části „Downloaders“ se nacházejí ještě dvě další položky, „Default intraday interval“ a „Default daily interval“. Jejich hodnoty se používají při generování úloh pro stahování dat během importu instrumentu a jde o intervaly provádění těchto úloh.

Skupina nastavení „Currency defaults“ umožňuje nastavit výchozí parametry obchodování s měnami. Podobně jako v případě trhů (viz. část 5.4) tyto údaje slouží k omezení požadavků při automatickém stahování dat, které nemají šanci na úspěch a tedy pouze plýtvají možný limit počtu přístupů k externímu zdroji.

V části *Credentials* je možné změnit uživatelské jméno a heslo používané pro přihlášení. Každá změna přihlašovacích údajů vyžaduje zadání doposud používaného hesla. Je možné změnit pouze přihlašovací jméno, pouze heslo, či obojí najednou. Aplikace neklade žádné požadavky na to, jak moc musí být heslo bezpečné, ale tato kritéria může stanovit použitý zásuvný modul pro přístup k úložišti.

5.9 Automatické stahování dat

Automatické stahování dat probíhá na principu periodického provádění nakonfigurovaných úloh. Tyto úlohy se konfigurují separátně pro každý instrument a měnu. Úloh může pro jeden instrument nebo měnu existovat více s různými nastaveními. Nastavení úloh automatického stahování pro daný instrument nebo měnu se otevře stiskem tlačítka „Downloader settings“ v pravé horní části editační obrazovky instrumentu nebo měny. Tlačítko je aktivní pouze, pokud jde

o již vytvořený a uložený instrument nebo vytvořenou a uloženou měnu.

Po stisknutí tohoto tlačítka se v novém dialogovém okně objeví tabulka se seznamem všech vytvořených úloh pro tento instrument nebo měnu (viz. obrázek 5.13) včetně základních informacích o nich. Mezi ty patří jejich název, zda jsou aktivované (sloupec „Enabled“, hodnota „T“ značí ano, „F“ znamená ne), priorita úlohy, rozlišení úlohy, datum a čas posledního zpracování úlohy a jaký zásuvný modul se při jejím zpracování používá ke stažení dat. Úlohu je možné spustit před uběhnutím jejího intervalu tlačítkem „Run now“, po jehož stisknutí vybraná úloha proběhne hned nebo po dokončení aktuálně probíhající úlohy. I v tomto případě ale platí, že úloha se provede pouze v případě, že je označená jako aktivní. Chování ostatních tlačítek je stejné jako u dalších seznamových obrazovek (viz. část 5.2).

V rámci editační obrazovky máme možnost všechny zmíněné položky upravovat. V dolní části obrazovky je pak navíc nastavení konkrétního zvoleného zásuvného modulu, který se má o stažení dat postarat. Tato nastavení typicky zahrnují označení, které externí zdroj používá jako název daného instrumentu nebo měny.

Rozlišení stahování určuje, zda bude úloha stahovat souhrnné informace při otevření a zavření trhu (denní rozlišení, hodnota „End-of-day“), nebo informace k určitému konkrétnímu časovému údaji (časové rozlišení, hodnota „Intraday“). V historii hodnoty nebo kurzu pak budou hodnoty z obou variant označené datem a časem. Čas v případě hodnot pro otevření a zavření trhu odpovídá informacím dostupným pro měnu (případně výchozí hodnoty společné pro všechny měny) nebo trh instrumentu.

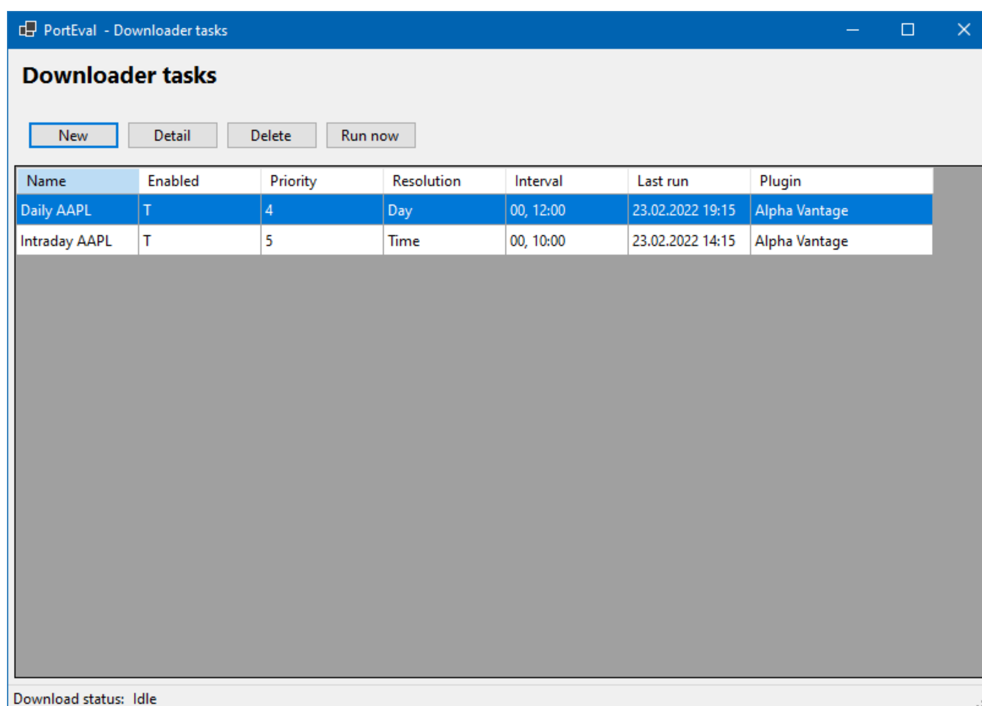
Priorita úlohy je důležitá především v případě, že se využívá více úloh se stejným rozlišením. V případě, že je pro daný datum a čas již v historii instrumentu nebo měny dostupný stažený údaj, který ale pochází z úlohy s nižší prioritou, je při zpracovávání úlohy s vyšší prioritou přepsán. Vyšší hodnota znamená vyšší prioritu. Úlohy s denním rozlišením mají vždy vyšší prioritu než úlohy s časovým rozlišením.

Interval určuje, jak často má docházet k provádění úlohy. Při provádění úloh se nejprve vybírají ty, kterým interval od posledního provedení uběhl nejdříve. Nejmenší možný interval je 1 minuta, největší je 99 dní, 23 hodin a 59 minut. Interval je potřeba nastavit na základě možností a limitů externího zdroje. Při každém provádění úlohy se provádí kontrola, zda není zapotřebí provést opravu dostupných údajů nebo smazat již zbytečně detailní údaje, a časté provádění úloh může mít za následek vyšší a — pokud data nejsou k dispozici — zbytečnou zátěž.

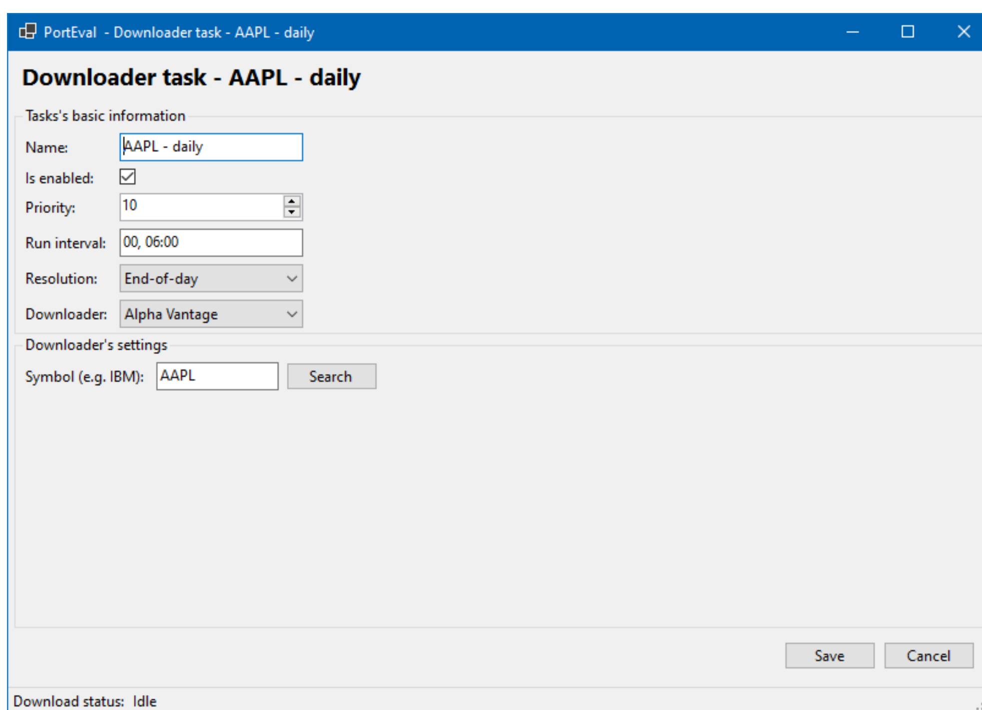
5.10 Grafy

Pro visuální analýzu jsou důležité grafy, které je možné nejen zobrazovat, ale i ukládat pro pozdější zobrazení. Obrazovka se seznamem uložených grafů, ze které je možné také přejít do editoru grafů, se zobrazí po stisknutí tlačítka „Graphs“ v levém bočním panelu.

Otevřená seznamová obrazovka (viz. obrázek 5.15) obsahuje tabulku se všemi uloženými grafy a jejich základními údaji, názvem, typem a intervalem zobrazených bodů. Oproti ostatním seznamovým obrazovkám se zde liší chování některých ovládacích prvků. Je zde navíc tlačítko „Show“, které otevírá celoobrazovkové zobrazení vybraného grafu. Dvojité kliknutí na řádek tabulky se pak chová



Obrázek 5.13: Obrazovka se seznamem úloh automatického stahování jednoho instrumentu nebo měny.



Obrázek 5.14: Editační obrazovka vybrané úlohy automatického stahování jednoho instrumentu nebo jedné měny.

právě jako toto tlačítko, protože se předpokládá, že zobrazení grafu bude častější operací, než jeho editace. Z tohoto důvodu je obvyklé tlačítko „Detail“ přejmenováno na „Edit“. Kromě těchto odlišností je chování ostatních tlačítek stejné jako u ostatních seznamových obrazovek (viz. část 5.2).

Editační obrazovka (viz. obrázek 5.16) obsahuje více informací než tabulka seznamové obrazovky. V horní části obrazovky se nachází volby typu grafu, konfigurace specifická pro tento typ, interval zobrazovaných bodů a časový rozsah, pro který se data v grafu zobrazují. Zbytek obrazovky zabírá tabulka s editovatelným seznamem datových řad zobrazených v grafu.

V aplikaci existují tři typy grafů, hodnotový, ziskový a výkonnostní. Hodnotový graf umožňuje v datových řadách sledovat hodnotu vybraného instrumentu nebo měny. Ziskové a výkonnostní grafy se zaměřují na sledování zisku, resp. výkonnosti, portfolia a pozic v portfoliích.

Zobrazovat všechny dostupné body může být příliš jemné a způsobovat nepřehlednost. Tento problém řeší přenastavení intervalu grafu z minut, kdy se zobrazuje každý dostupný bod, na některý z vyšších intervalů. To způsobí, že se několik bodů spadajících do stejného intervalu sdruží do jednoho bodu, zakresleného podle hodnoty posledního ze záznamů. Při ukázání kurzorem na bod se zobrazí počáteční, minimální, maximální a koncová hodnota bodů v intervalu.

Rozsah určuje, pro jaké období se budou data zobrazovat. Lze ho zadat dvěma způsoby, pomocí pevného počátečního a koncového data, nebo pomocí počtu dnů od aktuálního data. Zvolení relativního rozsahu v podobě určitého počtu dní zároveň aktivuje zobrazení tlačítek při zobrazení grafu, které umožňují dočasně rozsah změnit na některou z předdefinovaných hodnot.

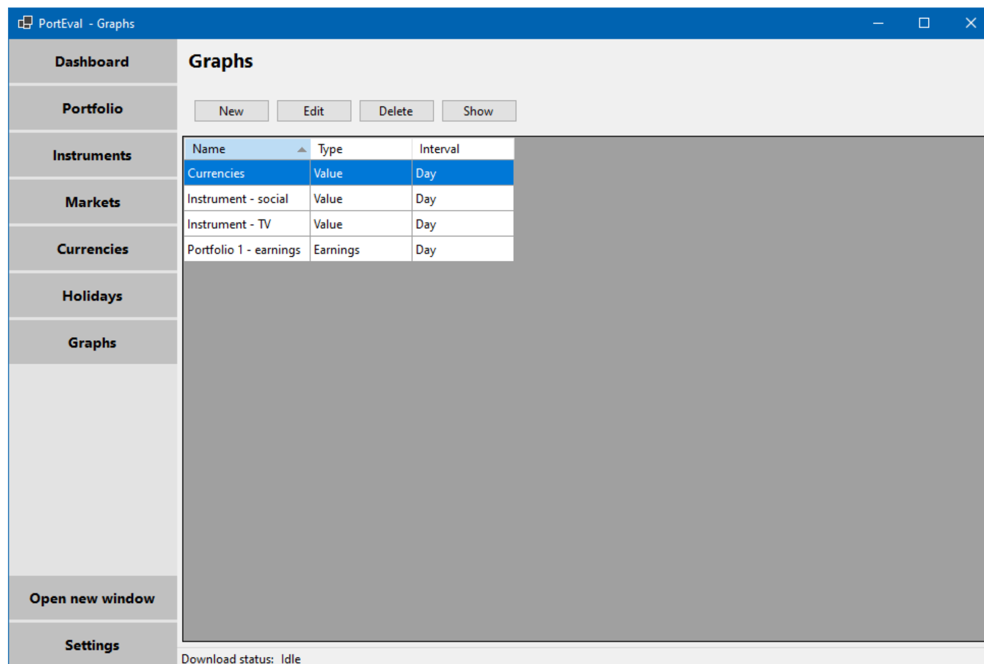
Seznam datových řad grafu se dá měnit pomocí tlačítek nad tabulkou datových řad. Nejčastější operace, přidávání datových řad, je navržena tak, že nabízí možnost přidat několik datových řad najednou. Po stisknutí tlačítka „Add“ se zobrazí dialog s volbou instrumentů a měn, resp. portfolií a pozic, v závislosti na typu grafu.

Datové řady mohou být aktuálně jedním ze dvou typů, spojnicová a spojnicová se značkami. Varianta se značkami je dostupná pouze pro ziskové a výkonnostní grafy, protože zobrazuje značky na místech, kde došlo k transakci. Volba typu datové řady a další související parametry jako barva nebo typ čáry se nastavují v dialogu, který se zobrazí po stisknutí tlačítka „Edit“. Datové řady mohou být také odebrány pomocí tlačítka „Delete“ nebo může být změněno jejich pořadí díky tlačítkům s šipkami nahoru a dolů.

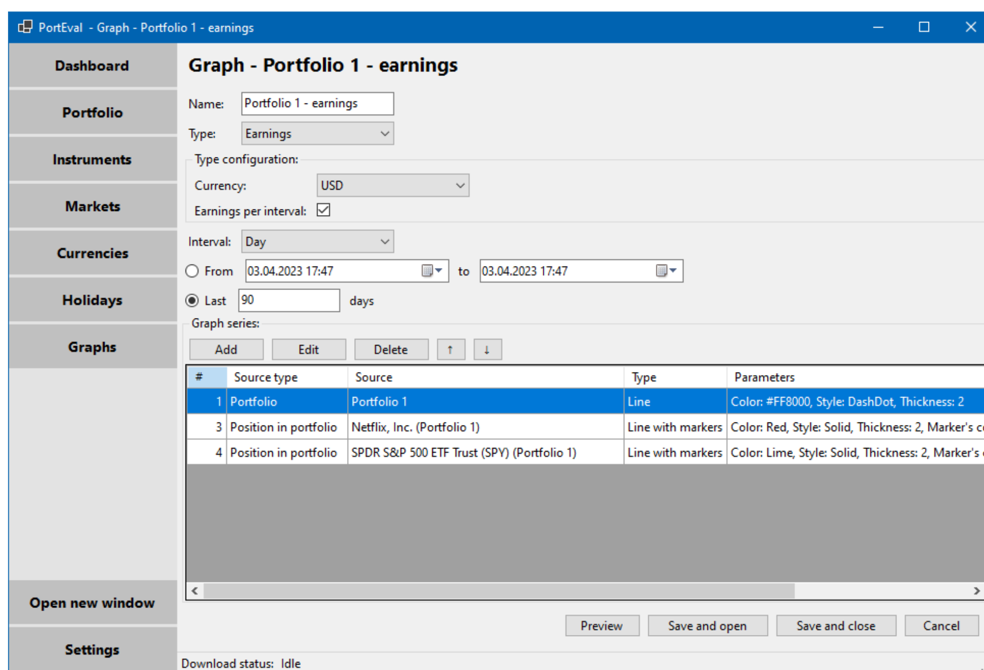
V dolní části obrazovky se kromě obvyklého tlačítka „Cancel“, které umožňuje odejít z obrazovky bez uložení grafu, nachází ještě další tři tlačítka. Jde o tlačítka „Save and close“, které provede uložení grafu a zobrazí zpátky obrazovku se seznamem grafů, „Save and open“, které graf také uloží, ale následně přepne obrazovku na celoobrazovkové zobrazení uloženého grafu, a tlačítko „Preview“. To otevře dialogové okno zobrazující graf vytvořený podle aktuálně vyplněných informací v editační obrazovce bez toho, aby byl graf uložen.

5.11 Nástěnka

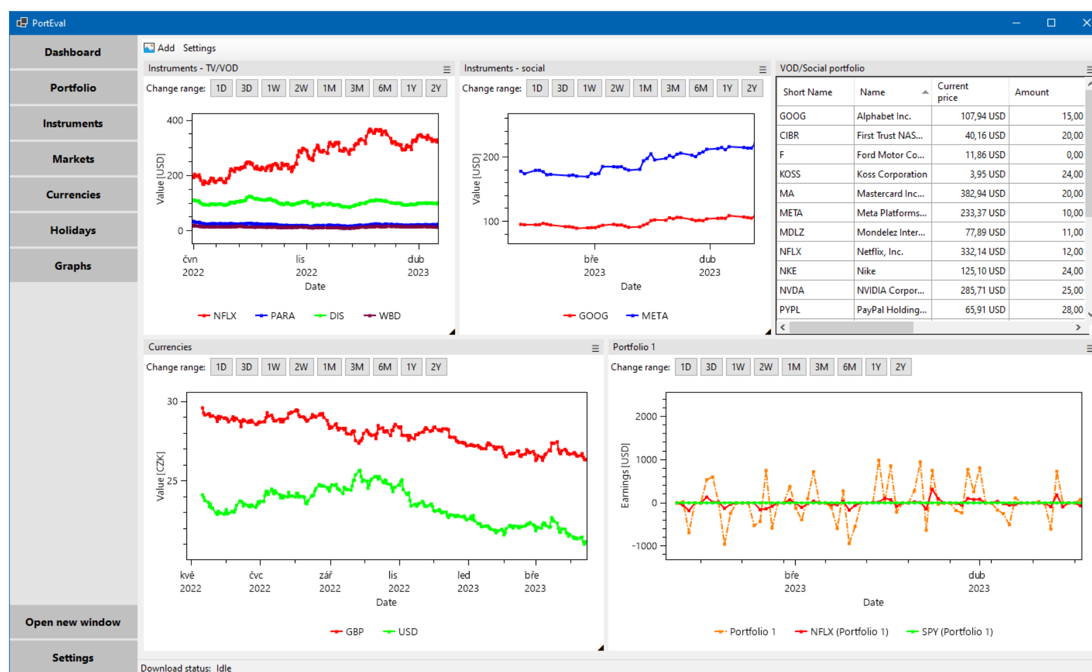
Po přihlášení do aplikace nebo stisknutí tlačítka „Dashboard“ v levém bočním panelu se zobrazí nástěnka (viz. obrázek 5.17). Jde o konfigurovatelnou obrazovku



Obrázek 5.15: Obrazovka se seznamem uložených uživatelsky definovaných grafů.



Obrázek 5.16: Editor konfigurace grafu.



Obrázek 5.17: Nástěnka umožňuje uživateli definovat a rozmístit po obrazovce soustavu widgetů s tabulkami nebo grafy.

schopnou zobrazovat plovoucí okna (widgety) s konfigurovatelným rozmístěním a velikostmi. Widget může obsahovat tabulku nebo jeden z uložených grafů. Nástěnku tvoří čtverečkovaná síť, která vyplňuje celou šířku obrazovky. Množství sloupců sítě může být změněno v nastavení nástěnky (tlačítko „Settings“ v horní liště obrazovky), což umožňuje upravit jemnost sítě. Výška sítě není pevně daná a je možné ji neomezeně rozšiřovat díky zobrazení posuvníku.

Pro přidání nového widgetu je potřeba se nejprve přepnout do vytvářecího režimu stisknutím tlačítka „Add“. K opuštění vytvářecího režimu bez vytvoření widgetu se využívá klávesa „Escape“. Ve vytvářecím režimu je možné držením levého tlačítka myši a tažením myši nakreslit widget. Tah myši při kreslení widgetu musí směřovat z levého horního rohu budoucího widgetu do pravého dolního rohu. Puštěním tlačítka myši dojde k vytvoření prázdného widgetu.

V pravé části hlavičky widgetu se nachází tlačítko (tři vodorovné čárky), které zobrazí kontextovou nabídku. Ta obsahuje možnost widget smazat nebo konfigurovat. Konfigurace umožňuje nastavit titulek widgetu, jeho typ, případné další nastavení specifické pro tento typ a číselně upravit velikost widgetu ve čtverečcích sítě nástěnky.

Přesun widgetů se podobně jako jejich vytváření provádí tahem myši. Stiskem a držením levého tlačíka myši na hlavičce widgetu se zahájí proces přesunu. Widget je vyjmut ze čtverečkované sítě a je možné s ním volně hýbat. Po puštění tlačítka myši je widget umístěn na místo, do kterého byl přetažen. Pokud by dané místo bylo obsazené, je vrácen na své původní umístění.

Podobně je také tahem myši možné změnit velikost widgetu. Operace se zahájí stiskem a držením levého tlačítka myši na trojúhelníkové značce v pravém dolním rohu widgetu. Poté lze tahem měnit velikost widgetu a puštěním tlačítka myši tuto velikost aplikovat a uložit. V případě, že by mělo dojít k překrytí dvou widgetů, vrátí se velikost widgetu na původní hodnotu.

Vytváření, přesouvání, mazání, konfigurace a změna velikosti jsou ihned po provedení operace ukládány do úložiště.

5.12 Možnost zobrazení více oken

Aplikace neumožňuje, aby byla spuštěna vícekrát. Místo toho lze ale vytvořit více oken pomocí tlačítka „Open new window" v dolní části levého bočního panelu. To dává možnost zobrazit si více obrazovek vedle sebe, přepínat mezi nimi a porovnávat jejich data. Aby nebylo nutné manuálně obnovovat informace po jejich úpravě v jednom z oken, předávají si okna mezi sebou informaci o změně dat a automaticky se aktualizují. V případě, že je např. jeden instrument otevřen a upraven ve více oknech, zobrazí se při uložení jednoho z nich upozornění s možností data v druhém okně obnovit. Pokud hrozí narušení integrity dat a není možné modifikace z obou oken bezpečně sloučit, jsou data po zobrazení upozornění obnovena vždy.

6. Otestování výkonu aplikace

Pro naši aplikaci je kromě funkčních požadavků na rozšiřitelnost a zobrazení údajů ve formě vhodné pro analýzu, tedy například ve formě grafů, stěžejní výkonnost načítání a výpočtů, aby uživatel mohl v aplikaci pracovat co možná nejrychleji. Zásadní je, aby uživatel nemusel příliš dlouho čekat, než se mu data zobrazí.

Potenciálně problémovým místem by mohl být výkon vykreslování grafů a výpočet IRR výkonnosti pozic a portfolií, kde je kvůli polynomiální rovnici využit aproximační algoritmus. Vykreslování grafů jsme testovali již při výběru knihovny pro jejich zobrazování (viz. část 3.3). Zaměříme se tedy nyní na výpočet IRR výkonnosti pozic a portfolií.

6.1 Test rychlosti výpočtu IRR výkonnosti

Hodnota IRR výkonnosti pro pozice a portfolia se v aplikaci počítá hned na několika místech a může jít o výpočet pro desítky až stovky transakcí na pozici a množství pozic a portfolií v rámci jednoho načítání obrazovky. Tyto obrazovky zobrazují portfolia a pozice nebo grafy zobrazující IRR výkonnost portfolií a pozic. Místo toho, abychom testovali výkon celého načítání obrazovky, zaměříme se na samotný výpočet.

6.1.1 Generování testovacích dat

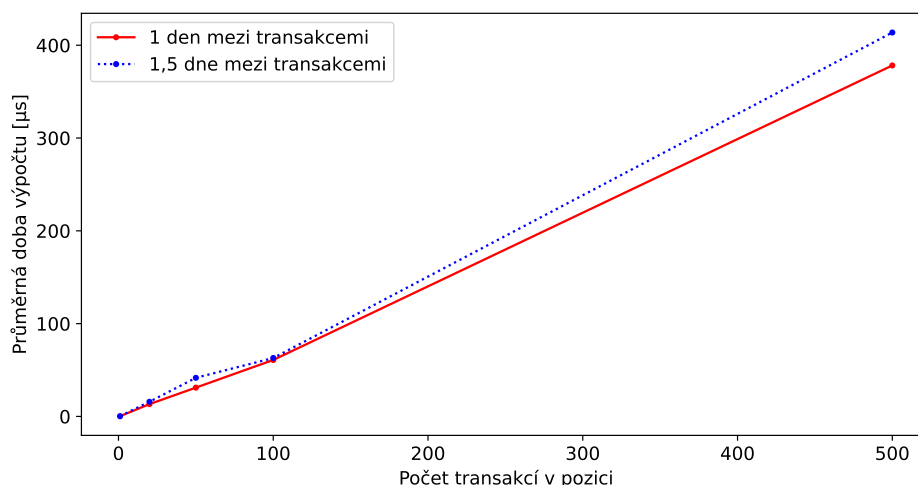
Pro pořádné otestování chceme testovat různé varianty počtu transakcí v pozici, počtu pozic v portfoliu a časového rozmezí mezi po sobě jdoucími transakcemi jedné pozice. Jelikož jde o větší množství kombinací, nebudeme generovat testovací data do souborů, ale vygenerujeme je před spuštěním konkrétního testu v paměti.

Ke každé pozici v portfoliu generujeme sadu testem určeného počtu transakcí. O množství a změně hodnoty rozhoduje náhoda s pevnou hodnotou „seed“. Množství jednotek transakce je s 10% pravděpodobností nastaveno na -10, pokud by následně celková suma nebyla záporná. V opačném případě jde o $+10$. Hodnota instrumentu začíná na čísle 100 a pro každou další transakci je hodnota instrumentu změněna o náhodně zvolenou hodnotu z intervalu $[-0{,}5; 1{,}5)$.

Aby mohl výpočet správně proběhnout, dochází v průběhu generování dat také k vytváření entit instrumentů, pozic, portfolií, měn a jednoho historického záznamu pro každou měnu a instrument.

6.1.2 Testovací prostředí

K měření využijeme knihovnu Benchmark.NET patřící mezi přední knihovny pro měření rychlosti programů vytvořených pro .NET včetně samotného prostředí .NET.[18] Měření provádíme na stolním počítači s procesorem AMD Ryzen 5 3600 s 16 GB operační paměti, operačním systémem Windows 10 21H2 a .NET 6.0.4. Během testování jsou všechna data načtená v paměti, čímž je eliminována latence jejich získávání z úložiště. V reálném případě by tato latence nebyla zanedbatelná,



Obrázek 6.1: Graf závislosti doby výpočtu IRR výkonnosti jedné pozice na počtu transakcí v pozici.

ale vzhledem k tomu, že rychlost reakce úložiště záleží na implementaci zásuvného modulu, nedává smysl snažit se ji v rámci tohoto testu měřit.

6.1.3 Analýza výsledků testování pozice

Pro otestování rychlosti výpočtu IRR výkonnosti pozice generujeme několik variant vstupních dat lišících se počtem transakcí v pozici a časovým rozmezím mezi nimi, což má vliv také na délku sledovaného intervalu. Jako počet transakcí jsme zvolili možnosti 1, 20, 50, 100 a 500, kdy hodnoty nad 50 transakcí jsou reálně dosažitelné jen při dlouhodobém používání a aktivním obchodování. Jsou tedy zařazeny spíše kvůli lepšímu pozorování vývoje s narůstajícím počtem transakcí.

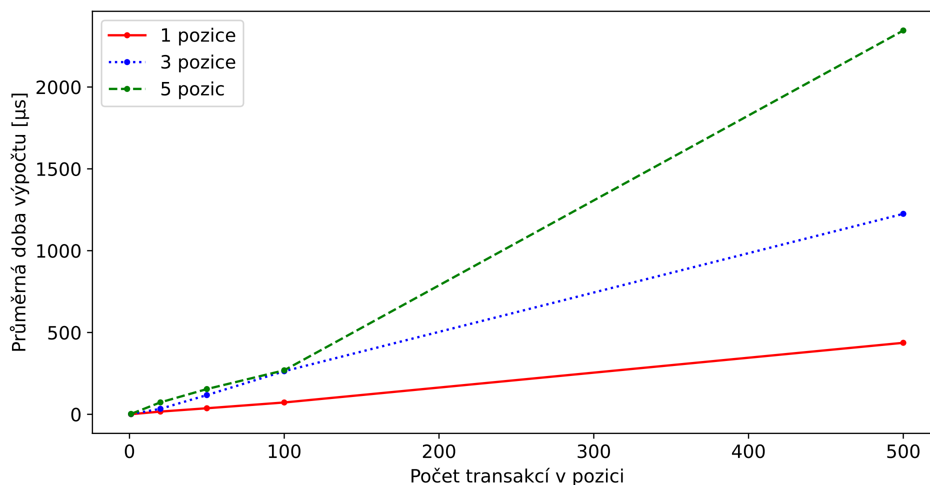
Výsledky ukazují, že doba výpočtu jedné pozice se pro malé a reálnější počty transakcí v pozici pohybují v řádu jednotek až desítek mikrosekund případně pro velké počty transakcí pak v řádech stovek mikrosekund (viz. tabulka A.1). Varianta s desetinným počtem dní mezi transakcemi dobu výpočtu nijak výrazně neprodloužila a to i přesto, že docházelo k výpočtu přes delší sledovaný interval (viz. tabulka A.2).

Důležité je, že s přibývajícím počtem transakcí v portfoliu je nárůst doby potřebné pro provedení výpočtu v obou případech lineární (viz. graf 6.1).

Vzhledem k zanedbání doby přístupu k datům z úložiště nejsme na základě těchto dat schopní říct, jak rychle se načte obrazovka obsahující výpočet IRR výkonnosti pozice s určitým počtem transakcí. Z údajů je ale zřejmé, že doba výpočtu IRR výkonnosti se na době načítání obrazovky nebude podílet natolik významně, aby bylo nutné vypočítané hodnoty ukládat a v případě potřeby pouze načítat z úložiště, jinými slovy, vyhýbat se provádění výpočtu.

6.1.4 Analýza výsledků testování portfolia

Výpočet IRR výkonnosti portfolia se od výpočtu výkonnosti pozice v některých ohledech liší, už jen proto, že se zde často počítá s větším množstvím



Obrázek 6.2: Graf závislosti doby výpočtu IRR výkonnosti portfolia v závislosti na počtu transakcí v jeho pozicích při různých množstvích pozic v portfoliu.

transakcí najednou. Zahájení výpočtu je navíc o něco složitější kvůli práci s více pozicemi a tedy i více cenami instrumentů.

Opět budeme během testování měnit několik parametrů, mezi kterými zůstane počet transakcí se stejnými hodnotami (1, 20, 50, 100 a 500). Jelikož časové rozmezí mezi transakcemi u pozic nevykazovalo výrazné změny v době výpočtu, rozhodli jsme se ho zde nahradit počtem pozic v portfoliu, což je v tomto případě zajímavější parametr. Testovat budeme pro tento parametr hodnoty 1, 3, a 5 pozic.

Ve výsledcích můžeme vidět, že doba výpočtu IRR výkonnosti portfolia s jednou pozicí je sice vyšší, ale blízká, době naměřené pro výpočet IRR výkonnosti pozice (viz. tabulka A.3). Podíváme-li se na výpočty přes více pozic je navýšení doby výpočtu výraznější (viz. tabulky A.4 a A.5). Stále se však při desítkách transakcí na pozici pohybujeme v řádu desítek až nižších stovek mikrosekund. Výrazně vyšší je až doba výpočtu při 500 transakcích na pozici, kdy se při 3 a 5 pozicích dostáváme přes hranici milisekundy.

V případě 1 pozice a 3 pozic v portfoliu výsledky ukazují lineární růst délky výpočtu s přibývajícím počtem transakcí (viz. graf 6.2). Zajímavější je situace u pěti pozic v portfoliu. Zde má křivka v grafu spíše exponenciální tvar. Dá se však předpokládat, že jde o chybu způsobenou použitými vygenerovanými daty, protože hodnoty do 100 transakcí jsou velice blízké, či téměř stejné jako hodnoty pro 3 pozice. Oproti tomu v případě 500 transakcí již rozdíly mezi dobami výpočtu pro jednotlivé počty pozic opět odpovídají lineárnímu růstu.

Při 500 transakcích jsme se tentokrát dostali už přes hranici milisekundy. Stále ale platí, že 500 transakcí na pozici je přemrštěné množství. I kdybychom tuto situaci zkusili převést na 30 pozic o 50 transakcích, což si soudě podle blízkosti hodnot pro 20 transakcí na pozici při 5 pozicích a pro 100 transakcí v rámci 1 pozice můžeme dovolit, takovýchto portfolií budeme mít v aplikaci jednotky. Doba výpočtu v řádu milisekund by tedy opět nezpůsobovala výrazné zpomalení načítání.

Závěr

Cílem této bakalářské práce bylo vytvořit aplikaci, která by umožnila menšímu nebo začínajícímu investorovi sledovat svá portfolia, a zároveň dávala uživateli kontrolu nad svými daty, byla dostatečně modifikovatelná a přitom jednoduchá na ovládání.

Na základě toho vznikla aplikace PortEval pro operační systém Windows s podporou zásuvných modulů pro přístup k úložišti a pro stahování dat z externích zdrojů, které dávají uživateli možnost zvolit si své úložiště a odkud má aplikace čerpat data, bez nutnosti provádět změny do samotné aplikace. Aby byla zachována jednoduchost ovládání, je vše nastavitelné přímo z grafického uživatelského rozhraní. I přes přítomnost automatického stahování, může uživatel přidávat záznamy manuálně například pro instrumenty, které dostupné externí zdroje neposkytují.

Také vytváření zásuvných modulů musí být pro případné vývojáře dostatečně jednoduché, a proto se v rámci nich nevyužívají žádná složitá rozhraní a jejich implementace je tedy přímočará.

Důraz byl kladen také na jednoduché úpravy jednotlivých záznamů v aplikaci. Z tohoto důvodu se související záznamy, které se typicky editují zároveň, nachází na jedné obrazovce a není tak nutné například při úpravách portfolia přecházet na jinou obrazovku s pozicí a zde teprve dělat změny transakcí. Vše je přehledně v rámci jedné obrazovky.

Pro analýzu jsou základní grafy a bylo tedy věnováno množství času na navržení vhodného editoru, aby operace vytváření grafu nezabrala příliš času. Často používané grafy je navíc možné uložit a používat jako součást nástěnky. U nástěnky byl opět investován čas do jednoduchosti používání a je tedy implementován systém drag-n-drop.

Při ohlédnutí za jinými existujícími produkty můžeme zkonstatovat, že PortEval nabízí většinu typických funkcionalit a přidává další, které nejsou běžné, jako jsou nástěnka, kompletně konfigurovatelné grafy nebo výpočet IRR výkonnosti. Oproti některým webovým aplikacím, kde jsou data v aplikaci řízena centrálně, pak aplikaci chybí podrobné informace o obchodovaných instrumentech jako jejich kapitálové statistiky nebo seznam vlastníků.

I tak je aplikaci možné dále rozšiřovat. Především pak přidáváním podpory pro další externí zdroje dat pomocí zásuvných modulů. Bylo by také možné provést úpravu, které umožní přidávat nové typy widgetů opět pomocí zásuvných modulů. Rozšířit by se dalo i množství typů datových řad grafů nebo počet sledovaných metrik.

Seznam použité literatury

[1] Board of Governors of the Federal Reserve System. Changes in U.S. Family Finances from 2016 to 2019: Evidence from the Survey of Consumer Finances. Dostupné z: https://www.federalreserve.gov/publications/files/scf20.pdf, 2020.

[2] Yahoo! Finance. Dostupné z: https://finance.yahoo.com/.

[3] Sharesight. Dostupné z: https://www.sharesight.com/.

[4] Kubera. Dostupné z: https://www.kubera.com/.

[5] Samuel A. Broverman. *Mathematics of investment and credit*, pages 264–266. ACTEX Publications, Inc, 2010.

[6] Simply Wall St. Dostupné z: https://simplywall.st/.

[7] StockMarketEye. Dostupné z: https://www.stockmarketeye.com/.

[8] Delta Investment Tracker. Dostupné z: https://delta.app/en.

[9] Personal Capital. Dostupné z: https://www.personalcapital.com/.

[10] ElectronJS. Dostupné z: https://www.electronjs.org/.

[11] statcounter. Desktop Operating System Market Share Worldwide. Dostupné z: https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202103-202203, Březen 2022.

[12] Qt. Dostupné z: https://www.qt.io/.

[13] Klaus Loeffemann. State of the Windows Forms Designer for .NET Applications. Dostupné z: https://devblogs.microsoft.com/dotnet/state-of-the-windows-forms-designer-for-net-applications/, Leden 2022.

[14] WPF. Dostupné z: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0.

[15] Xamarin Forms. Dostupné z: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms.

[16] Alberto Rodríguez. LiveCharts. Dostupné z: https://lvcharts.net/Home/About.

[17] OxyPlot. Dostupné z: https://oxyplot.github.io/.

[18] Benchmark.NET. Dostupné z: https://github.com/dotnet/BenchmarkDotNet.

[19] Dokumentace ReaderWriterLockSlim. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.threading.readerwriterlockslim?view=net-6.0#remarks.

Seznam obrázků

3.1 UML diagram části datového modelu s přímými vazbami na entitu portfolia. . . . 18
3.2 UML diagram části datového modelu bez přímých vazeb na entitu portfolia. Tyto entity stojí buď samostatně nebo jsou na zbytek datového modelu vázány nepřímo pomocí položky s typem entity a položky s identifikátorem entity, ke které se váží. . . . 19

4.1 Diagram zobrazující hlavní části aplikace a jejich propojení. . . . 28

5.1 Obrazovka zobrazující seznam vytvořených měn. . . . 44
5.2 Obrazovka zobrazující detail měny a jejího kurzu s možností editace. 44
5.3 Obrazovka zobrazující seznam evidovaných trhů. . . . 45
5.4 Editační obrazovka trhu. Čas otevření a zavření trhu se zadává v časové zóně trhu. . . . 45
5.5 Obrazovka se seznamem sledovaných instrumentů . . . 46
5.6 Obrazovka pro import instrumentů . . . 48
5.7 Editační obrazovka instrumentu a historie vývoje jeho hodnoty. Měnu instrumentu je možné měnit pouze, pokud se v historii hodnoty nenachází žádný záznam. . . . 48
5.8 Obrazovka se seznamem portfolií a rozbalitelnými seznamy pozic v rámci jednotlivých portfolií. . . . 49
5.9 Editační obrazovka portfolia, která umožňuje přidávat, upravovat a mazat transakce provedené v rámci tohoto portfolia. . . . 49
5.10 Obrazovka se seznamem skupin dní, během kterých se neobchoduje, s rozbalovacími seznamy jednotlivých dní v rámci skupin. . . 51
5.11 Editační obrazovka skupiny obchodních prázdnin s možností přidávat, upravovat a mazat jednotlivé volné dny, případně zkopírovat všechny z předchozího roku. . . . 51
5.12 Obrazovka s globálními nastaveními aplikace jako jsou výchozí údaje o měnách, volba referenční měny, konfigurace zásuvných modulů a změna přihlašovacích údajů. . . . 52
5.13 Obrazovka se seznamem úloh automatického stahování jednoho instrumentu nebo měny. . . . 54
5.14 Editační obrazovka vybrané úlohy automatického stahování jednoho instrumentu nebo jedné měny. . . . 54
5.15 Obrazovka se seznamem uložených uživatelsky definovaných grafů. 56
5.16 Editor konfigurace grafu. . . . 56
5.17 Nástěnka umožňuje uživateli definovat a rozmístit po obrazovce soustavu widgetů s tabulkami nebo grafy. . . . 57

6.1 Graf závislosti doby výpočtu IRR výkonnosti jedné pozice na počtu transakcí v pozici. . . . 60
6.2 Graf závislosti doby výpočtu IRR výkonnosti portfolia v závislosti na počtu transakcí v jeho pozicích při různých množstvích pozic v portfoliu. . . . 61

A. Přílohy

A.1 Instalace aplikace

Za předpokladu splnění požadavků na systém je aplikaci možné nainstalovat dvěma způsoby, sestavením ze zdrojového kódu nebo použitím již hotového sestavení.

A.1.1 Požadavky na systém

Ke spuštění aplikace je potřeba operační systém Windows 7 SP1 nebo novější a nainstalovaný .NET 6 nebo novější.

A.1.2 Sestavení

Pro sestavení aplikace je zapotřebí mít kromě splněných požadavků na systém také nainstalovaný nástroj dotnet ve variantě SDK nebo Visual Studio 2022. Následně proveďte následující kroky:

1) Otevřete Developer PowerShell ve Visual Studio 2022 nebo přes Start > Visual Studio 2022 > Developer PowerShell.

2) V adresáři projektu zadejte příkaz:

```
> dotnet publish
```

3) Po dokončení sestavení překopírujte všechny soubory z podadresáře „dist“ do adresáře, ze kterého budete chtít aplikaci spouštět.

A.1.3 Využití již hotového sestavení

Není nutné instalovat veškeré nástroje kvůli sestavení aplikace, protože součástí elektronické přílohy je také již hotové sestavení v archivu PortEval.zip. Toto sestavení stačí rozbalit do vybraného adresáře. Součástí elektronické přílohy je také databáze obsahující vzorová data.

A.1.4 Přidávání zásuvných modulů a konfigurace

Součástí aplikace jsou zásuvné moduly pro přístup k SQLite databázi a pro stahování dat z AlphaVantage.co a StockData.org. Další zásuvné moduly pro přístup k databázi je možné přidat do adresáře lib/db. Moduly pro stahování dat se vkládají do adresáře lib/downloader.

V případě použití SQLite jako úložiště se prázdná SQLite databáze nachází na cestě lib/db/DBPlugin.SQLite/database.db a můžete ji přesunout do libovolného jiného adresáře. Pokud chcete mít databázi uloženou v jiném adresáři nebo používat jinou databázi, můžete provést změnu nastavení před přihlášením.

A.2 Data získaná při měření rychlosti výpočtu IRR výkonnosti

Počet transakcí	**Doba výpočtu [μs]**	**99,9% konfidenční interval [μs]**	**Směrodatná odchylka [ns]**
1	0,24	[0,240 ; 0,248]	3,67
20	13,25	[13,230 ; 13,274]	19,14
50	31,03	[30,967 ; 30,100]	58,96
100	60,83	[60,693 ; 60,958]	123,88
500	378,22	[377,742 ; 378,696]	446,27

Tabulka A.1: Výsledky měření rychlosti výpočtu IRR výkonnosti pozice s časovým rozmezím mezi transakcemi 1 den.

Počet transakcí	**Doba výpočtu [μs]**	**99,9% konfidenční interval [μs]**	**Směrodatná odchylka [ns]**
1	0,25	[0,249 ; 0,253]	1,74
20	15,80	[15,773 ; 15,881]	22,48
50	41,61	[41,522 ; 41,691]	78,95
100	62,96	[62,819 ; 63,097]	116,07
500	413,72	[413,108 ; 414,340]	575,88

Tabulka A.2: Výsledky měření rychlosti výpočtu IRR výkonnosti pozice s časovým rozmezím mezi transakcemi 1,5 dne.

Počet transakcí	Doba výpočtu [µs]	99,9% konfidenční interval [µs]			Směrodatná odchylka [ns]
1	0,77	[0,762	;	0,774]	5,36
20	16,75	[16,705	;	16,802]	42,78
50	37,10	[37,007	;	37,197]	79,58
100	72,43	[72,204	;	72,657]	211,85
500	437,05	[436,527	;	437,576]	490,59

Tabulka A.3: Výsledky měření rychlosti výpočtu IRR výkonnosti portfolia obsahujícího 1 pozici.

Počet transakcí	Doba výpočtu [µs]	99,9% konfidenční interval [µs]			Směrodatná odchylka [ns]
1	1,55	[1,551	;	1,557]	2,96
20	33,55	[33,486	;	33,619]	62,17
50	117,55	[117,255	;	117,838]	272,78
100	263,46	[262,877	;	264,037]	513,96
500	1 225,10	[1 223,000	;	1 227,000]	1 702,80

Tabulka A.4: Výsledky měření rychlosti výpočtu IRR výkonnosti portfolia obsahujícího 3 pozice.

Počet transakcí	Doba výpočtu [µs]	99,9% konfidenční interval [µs]			Směrodatná odchylka [ns]
1	2,46	[2,448	;	2,465]	7,97
20	73,09	[72,788	;	73,395]	268,86
50	154,36	[154,117	;	154,608]	217,66
100	269,04	[268,340	;	269,745]	657,07
500	2 345,42	[2 339,000	;	2 352,000]	5 865,71

Tabulka A.5: Výsledky měření rychlosti výpočtu IRR výkonnosti portfolia obsahujícího 5 pozic.

A.3 Adresářová struktura elektronické přílohy

Součástí práce je elektronická příloha obsahující zdrojový kód aplikace a souvisejících částí, její sestavení a vzorová data ve formátu databáze SQLite.

```
/
├─ PortEval.zip                  Připravené sestavení aplikace (A.1)
├─ database.db                   Vzorová data (SQLite)
└─ src
   ├─ DBPlugin.SQLite             Zásuvný modul k SQLite databázi (4.12.1)
   ├─ Downloader.AlphaVantage     Zásuvný modul k Alpha Vantage (4.12.2)
   ├─ Downloader.Commons          Společné části modulů pro stahování
   ├─ Downloader.StockDataOrg     Zásuvný modul k StockData.org (4.12.2)
   ├─ PortEval                    Aplikační logika
   │  ├─ Data
   │  │  ├─ DataGridFiller        Třídy typu IDataGridFiller (4.4)
   │  │  ├─ EditableEntity        Editovatelné varianty entit (3.4.2)
   │  │  ├─ Logic                 Třídy provádějící výpočty nad daty
   │  │  └─ ValueProvider         Poskytování hodnoty instrumentu či měny
   │  └─ UI
   │     ├─ Component             Komponenty grafického rozhraní (4.3)
   │     ├─ Dialog                Dialogové obrazovky aplikace (4.2.3)
   │     ├─ Screen                Obrazovky aplikace (4.2)
   │     ├─ Updater               Pomocné třídy pro obnovu zobrazených dat
   │     └─ Widget                Jednotlivé typy widgetů (4.3.2)
   ├─ PortEval.API                Definice tříd použitelných v modulech
   │  └─ Entity                   Jednotlivé entity (3.4.3)
   └─ PortEval.Benchmark          Aplikace pro měření výkonu (6.1)
```