Česká zemědělská univerzita v Praze
Provozně ekonomická fakulta
Katedra informačního inženýrství

Diplomová práce

Hybridní model pro výpočet rozúčtování spotřeb elektrické energie

Bc. Vít Jašek

© 2020 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Vít Jašek

Systémové inženýrství a informatika
Informatika

Název práce

Hybridní model pro výpočet rozúčtování spotřeb elektrické energie

Název anglicky

Hybrid model for calculation of electric consumptions

Cíle práce

Práce je zaměřena na problematiku datového skladu a jeho uživatelské obsluhy. Cílem práce je na ukázkových modulech aplikace demonstrovat aplikační obsluhu databáze Oracle za pomocí jazyků PHP7, HTML5 a Javascipt a poskytnout uživatelsky přívětivý systém datového skladu, který bude podporovat a ulehčovat chod řízeného podniku.

Metodika

Práce se skládá ze dvou částí – přehledu teoretických východisek a praktické části, jež se věnuje implementaci. Metodika zpracování teoretické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků bude popsána problematika vývoje datového skladu z pohledu datového návrhu, definice procesů a implementace logiky a dalších částí systému. Implementace bude vytvořena s pomocí jazyků PHP, HTML, javascript a databázového stroje Oracle SQL. V praktické části práce bude s využitím uvedených jazyků demonstrována tvorba jednotlivých modulů systému aplikace datového skladu a jejich uživatelská obsluha z naprogramované aplikace. Aplikace bude otestována na dostatečnou robustnost datového skladu a z pohledu uživatele také dostatečnou uživatelskou přívětivost.

Doporučený rozsah práce

60 – 100 stran

Klíčová slova

ETL; datový sklad; PHP; SQL; Oracle

Doporučené zdroje informací

Ralph KIMBALL, Joe CASERTA. The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data. Wiley Publishing, Inc., 2004.

Ross, Ralph Kimball and Margy. The Data Warehouse Toolkit: The Defi nitive Guide to Dimensional Modeling, Third Edition. John Wiley & Sons, Inc., 2013.

Vrána, Jakub. 1001 tipů a triků pro PHP. Computer Press, Albatros Media a.s., 2017.

Předběžný termín obhajoby

2019/20 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 06. 04. 2020

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Hybridní model pro rozúčtování spotřeb elektrické energie" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne datum odevzdání ______________________________

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Vojtěchu Merunkovi, Ph.D. za vedení práce a cenné rady, Ing. Tomáši Liškovi, Ph.D. za možnost podílet se na vývoji aplikace HM a svojí ženě za trpělivost a motivaci.

Hybridní model pro výpočet rozúčtování spotřeb elektrické energie

Abstrakt:

Práce je zaměřena na programování webové aplikace v PHP7 nad datovým skladem. V práci se popisuje, jak korektně pracovat s datovým skladem i jak ho plnit daty ETL procesem. ETL proces je popsán z pohledu několika datových zdrojů a u každého zdroje je popsáno jaké výhody a nevýhody vybraný zdroj má. Protože ne každý zdroj lze použít na přímo, jsou v práci vysvětleny postupy na zpracování různých pomocných skriptů. V závěrečné fázi je popsán automat, který umí pomocí Cronu spouštět v ETL skripty paralelně.

Klíčová slova: PHP7, jQuery, Javascript, FTP, CSV, EHV, EJ, EE, ETL

Hybrid model of electricity distribution

Abstract:

This diploma thesis is focused on programming a web application using PHP7 and working with data warehouse. The thesis describes how to work correctly with the data warehouse and how to fill it using ETL data process. ETL process is described from the perspective of several data sources and for each source is described what advantages and disadvantages the selected source has. Because not every resource can be used directly, the work explains the procedures for processing various auxiliary scripts. In the final phase is described automaton, which is done by using Cron to run ETL scripts in parallel.

Keywords: PHP7, jQuery, Javascript, FTP, CSV, EHV, EJ, EE, ETL

Obsah

OBSAH **9**
SEZNAM OBRÁZKŮ **11**
SEZNAM TABULEK **12**
1 ÚVOD **13**
2 CÍL PRÁCE A METODIKA **14**
2.1 Cíl práce **14**
2.2 Metodika **14**
3 TEORETICKÁ VÝCHODISKA **15**
3.1 Hybridní model **15**
3.1.1 Určování měrné spotřeby 16
3.1.2 Faktory ovlivňující měrnou spotřebu EE 17
3.1.3 Energetické ztráty 18
3.2 Datový sklad **18**
3.2.1 Co to je datový sklad 18
3.2.2 Požadavky na datový sklad 20
3.2.3 Architektura datového skladu HM 20
3.2.4 Čtyřvrstvá datová architektura HM 21
3.2.5 ETL proces 23
3.3 Použité vývojové nástroje **34**
3.3.1 Nástroj na vývoj aplikace 35
3.3.2 Nástroj na vývoj databáze 35
3.3.3 Web server 35
3.3.4 Správce projektových knihoven 36
3.3.5 Nástroj na verzování softwaru 36
3.3.6 FTP klient 37
3.3.7 Správa serveru 37
3.4 Použité technologie **37**
3.4.1 Jazyk PHP 37
3.4.2 Pravidla práce s databází 42
4 VLASTNÍ PRÁCE **43**

4.1 Stavba aplikace na zelené louce ... 43
4.1.1 Aplikační prostředí ... 43
4.1.2 GitHub ... 43
4.1.3 Přípravy před začátkem programováním ... 46
4.1.4 Aplikační jádro ... 47
4.2 ETL proces ... 57
4.2.1 Pomocné skripty ... 58
4.2.2 Sumy spotřeb ... 63
4.2.3 Snímky z měřáků ... 63
4.2.4 Hrany ... 68
4.2.5 Plánovač úloh ... 68
5 ZÁVĚR ... 73
6 SEZNAM POUŽITÝCH ZDROJŮ ... 74
7 PŘÍLOHY ... 76

Seznam obrázků

Obrázek 1 - blokové schéma instalace měřících zařízení (SŽDC, 2019) 16
Obrázek 2 – princip datového skladu 19
Obrázek 3 - čtyřvrstvá architektura 22
Obrázek 4 – MVC (Zend, 2011) 40
Obrázek 5 - životní cyklus issue 44
Obrázek 6 - rozhraní Sourcetree 46
Obrázek 7 - tabulka cron_jobs 60
Obrázek 8 ERD cron jobů 69

Seznam tabulek

Tabulka 1 - před normalizací..29
Tabulka 2 - po normalizaci..30
Tabulka 3 - tabulka uživatelů konečná normalizace 30
Tabulka 4 - tabulka bydliště konečná normalizace 30

1 Úvod

Hybridní model je aplikace, která umožňuje účtovat spotřeby trakční elektrické energie mezi jednotlivé dopravce pro měřená i neměřená elektrická hnací vozidla (EHV) případně elektrické jednotky (EJ). V Hybridním modelu, kromě zmíněných, nalezneme také i informace o lokomotivách na diesel. O tom, jestli jsou EHV/EJ měřené nebo neměřené rozhoduje, jestli vozidlo má v sobě zabudovaný speciální měřák spotřeb, který kromě spotřeb a rekuperací obsahuje i GSM anténu a GPS lokátor. Jízda vlaku je vždy určená grafikonem, jehož body jsou určeny číselníkem, v ČR SR70.

Tyto záznamy jsou obsaženy v primárních zdrojích. Hybridní model tedy potřebuje proces, kterým záznamy získá a vhodnou formou validuje pro výpočetní část aplikace. Procesu, který všechny tyto operace zajišťuje se říká ETL, v češtině někdy i datová pumpa.

Ve teoretické části práce se zabývám typy datových zdrojů a možností jejich extrakce, transformace a nahrávání do databáze. Popisuji úskalí každého zdroje a typy chyb na které můžeme nejčastěji narazit. V závěrečné fázi teoretické části popisuji nástroje, technologie a pravidla programování, které jsem použil a považuji za nezbytné při tvorbě projektu.

V praktické části ukážu a popíšu aplikační prostředí a způsob práce s verzovacím software, která přímo vychází z prostředí. Dále ukážu a popíšu tvorbu jádra, které je nutné pro hladký chod a následně pohodlnější programování, při tvorbě nových modulů či úpravách. ETL proces jsem rozdělil podle datových zdrojů a u každého zdroje popsal, jak z datového zdroje čerpat. U každého datového zdroje jsou popsány typické chyby, které mohou v datech být. V závěrečné části se věnuji programování plánovače úloh, který umožňuje paralelní spouštění skriptů. Jeho uplatnění je právě v ETL procesu.

2 Cíl práce a metodika

2.1 Cíl práce

Práce je zaměřena na problematiku aplikace datového skladu a jeho uživatelské obsluhy. Cílem práce je na ukázkových modulech aplikace demonstrovat aplikační obsluhu databáze Oracle za pomocí jazyků PHP7, HTML5 a JavaScript a poskytnout přívětivý systém datového skladu, který bude podporovat a ulehčovat chod řízeného podniku.

2.2 Metodika

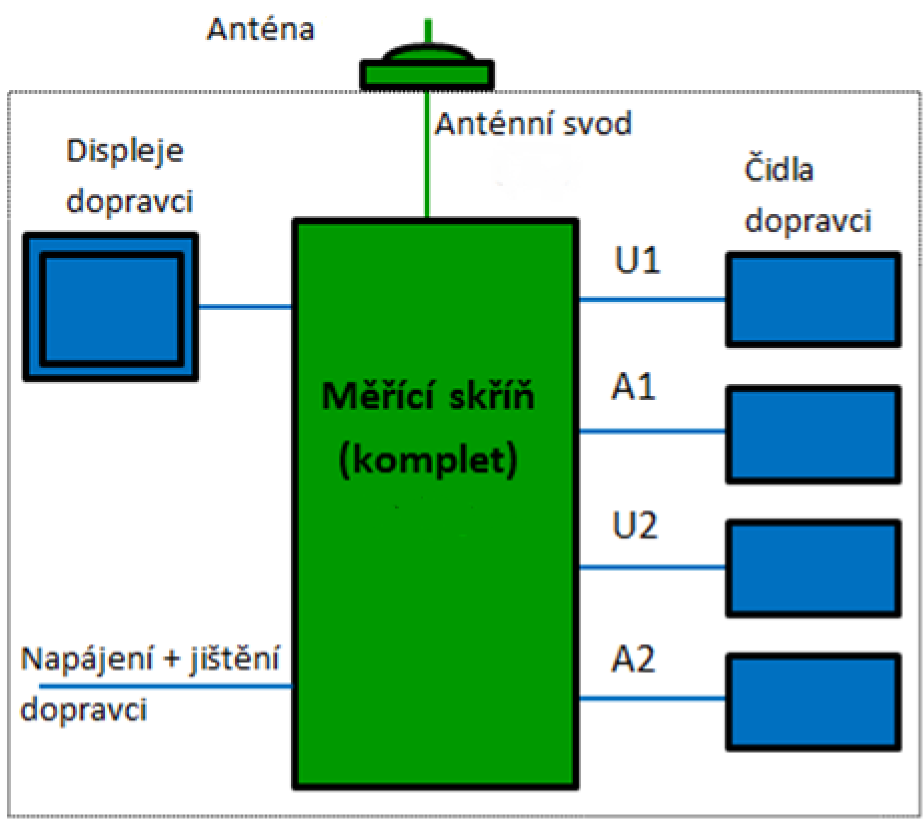
Práce se skládá ze dvou částí – přehledu teoretických východisek a praktické části, jež se věnuje implementaci. Metodika zpracování teoretické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků bude popsána problematika vývoje datového skladu z pohledu datového návrhu, definice procesů a implementace logiky a dalších částí systému. Implementace bude vytvořena s pomocí jazyků PHP7, HTML5, JavaScript a databázového stroje Oracle SQL. V praktické části práce bude s využitím uvedených jazyků demonstrována tvorba jednotlivých modulů systému aplikace datového skladu a jejich uživatelská obsluha z naprogramované aplikace. Aplikace bude otestována na dostatečnou robustnost datového skladu a z pohledu uživatele také dostatečnou uživatelskou přívětivost.

3 Teoretická východiska

Ve své diplomové práci se budu zabývat stavbou/programováním aplikace nad datovým skladem. Protože se jedná o aplikaci s netriviálními procesy nad netriviální databází, je potřeba si v teoretické části práce říct co je datový sklad, jak se s ním pracuje, v čem je to jiné od neskladové databáze. Pro práci nad datovým skladem platí jistá pravidla, kterých je třeba se držet, aby nedošlo k narušení konzistence dat, auditní stopy atd. Nejprve vysvětlím, co to je Hybridní model a k čemu slouží. V kapitole datového skladu se budu zabývat ETL procesem, jeho tvorbou, návrhem a pravidly, což je vlastně datový vstup skrze aplikaci do databáze. Výstup už je spíše záležitostí praktické části, proto bude popsán tam. Zde se dotknu pouze technik výstupu, které jsou nezbytné pro aplikaci a musí tedy být obsažené v teoretické části. Dále zde popíšu jazyky a technologie, které byly zvoleny pro vytvoření aplikace. K práci s těmito jazyky jsem použil programovací nástroje (programovací studia), které hrají důležitou roli hlavně v usnadnění práce, strukturování kódu a případně i v refaktoringu. Z těchto důvodů se o nich zde zmíním, aby se případný čtenář mohl inspirovat a nemusel experimentovat s něčím nevyzkoušeným.

3.1 Hybridní model

Hybridní model je aplikace, která umožňuje účtovat spotřeby trakční elektrické energie mezi jednotlivé dopravce pro měřená i neměřená elektrická hnací vozidla (EHV) případně elektrické jednotky (EJ). Krom výše zmíněných Hybridní model v sobě obsahuje i jízdy lokomotiv dieselových. Lokomotivy spadající do kategorie měřených vozidel, jsou vybaveny speciálním zařízením pro měření spotřeby trakční elektřiny. Zařízení v sobě, krom samostatného měřáku trakční elektrické energie, obsahuje i antény GPS a GSM umožňující posílání přesné polohy a naměřených údajů o EHV do datového skladu poskytovatele trakční energie. (SŽDC, 2019)



Obrázek 1 - blokové schéma instalace měřících zařízení (SŽDC, 2019)

3.1.1 Určování měrné spotřeby

Určování spotřeby elektrické energie jednotlivých EHV/EJ je komplikovaná problematika. Tato problematika má mnoho parametrů, které vždy nelze jednoznačně vyhodnotit vzhledem k technologiím dopravy. V této problematice se zabýváme hlavně měřením elektrické energie, rekuperací elektrické energie a ztrátami trakčních napájecích systémů. Zajímavé jsou pak přechody mezi trakčními systémy 3kV DC a 25kV AC. (ČD, 2017)

Spotřebu trakční elektrické energie (EE) lze rozdělit na složky:

- odběr činné EE EHV/EJ pro trakční účely, vytápění a klimatizaci vlakových souprav, restaurační vozy
- odběr činné EE pevnými elektrickými předtápěními zařízeními (EPZ)
- spotřebu EE ostatních zařízení nutných pro provoz železniční dopravní cesty (napájení zabezpečovacích zařízení, elektrický ohřev výměn, ostatní netrakční technologická spotřeba z trakce atd.)

- součet celkových činných ztrát EE vznikajících v trakční napájecí soustavě. (ČD, 2017)

Pro určování hodnot činné EE odebírané EHV/EJ se využívá zprůměrovaných hodnot měrných spotřeb EE, zjišťovaných teoretickými výpočty. Jednotkou měrné spotřeby EE za sledované období jsou kWh. Tyto kWh jsou vztaženy na užitečnou dopravní práci udávanou v hrubých tunokilometrech (kWh/hrtkm). Tato metoda vyhodnocení je dohodnuta mezi dodavatelem trakční EE a jednotlivými dopravci a běžně používána bez rozlišení využívaných typů a řad EHV/EJ, druhů vlaků i tratí, kde je výkon prováděn. Část EHV a EJ některých dopravců je v současnosti osazena elektroměry s využitelným přenosem dat. Je tedy možno využít elektroměrů EE instalovaných na EHV a EJ pro přesné určení odebrané EE. Údaje elektroměrů musí být následně upraveny korekčním činitelem, protože naměřené hodnoty nezahrnují energetické ztráty v trakčním napájecím systému (tj. v napájecích stanicích a v rozvodu trakční EE), respektují však nepravidelnosti v průběhu jízdy, a to i nepravidelnosti spojené s technologií dopravy. (ČD, 2017)

3.1.2 Faktory ovlivňující měrnou spotřebu EE

Měrná spotřeba EE u EHV v elektrické trakci jak pro nákladní dopravu, tak i pro osobní dopravu je závislá na velkém počtu faktorů:

- Dopravní faktory (složení a charakter vlakové soupravy, vytížení tratě, grafikon dopravy v daném úseku, způsob zabezpečení vlakové cesty atd.)
- Technika jízdy daná grafikonem vlakové dopravy a pokyny dopravní služby
- Přepravní faktory a druh vlaku pro nákladní dopravu:
    - Nex – nákladní expres
    - Pn – průběžné nákladní vlaky
    - Lv – lokomotivní vlak
    - Mn – manipulační nákladní vlaky
    - Służ – služební vlaky
- Přepravní faktory a druh vlaku pro osobní dopravu:
    - Ex a R (SC, EC, IC, Ex, EN, R, Sp, Sv) – expresy a rychlíky
    - Os – zastávkové osobní vlaky, ostatní vlaky osobní dopravy
- Typ EHV nebo EJ jejich technické řešení (druh regulace tažné síly a rychlosti dané trakčními charakteristikami)

- Technické řešení trakčních napájecích zařízení (DC nebo AC systém)
- Technické řešení a stav železničních vozů (typy vozů, jejich technický stav, brzdové procento)
- Konstrukce a technický stav železničního svršku (vliv případně předepsaných pomalých jízd, výluky traťových úseků)
- Vliv výškových a směrových poměrů na trati (‰ stoupání či sklonu, oblouky)
- Vliv plánovaných (výluky pro údržbové či investiční práce na trati) i mimořádných (neplánovaných) dopravních nepravidelností (vlivem havárie, poruch trakčního vedení) atd. (ČD, 2017)

3.1.3 Energetické ztráty

Energetické ztráty jsou dány součtem celkových činných ztrát EE vznikajících v trakčním napájecím systému a jsou dnes nejdiskutovanější a nejsložitější složkou týkající se vztahů vlastníka železniční dopravní cesty a dopravců. Mezi tyto ztráty patří všechny spotřeby EE, které lze vysledovat mezi bodem měření v trakční napájecí síti a měřením na EHV nebo EJ, z dostupných analýz pro zpracovatele.

- Pro systém DC 3 kV je hodnota stanovená procentních ztrát v napájecí síti jako celku v rozmezí 10 až 20 %
- Pro systém AC 25 kV, 50 Hz je hodnota stanovená procentních ztrát jako celku v rozmezí 5 až 10 % (ČD, 2017)

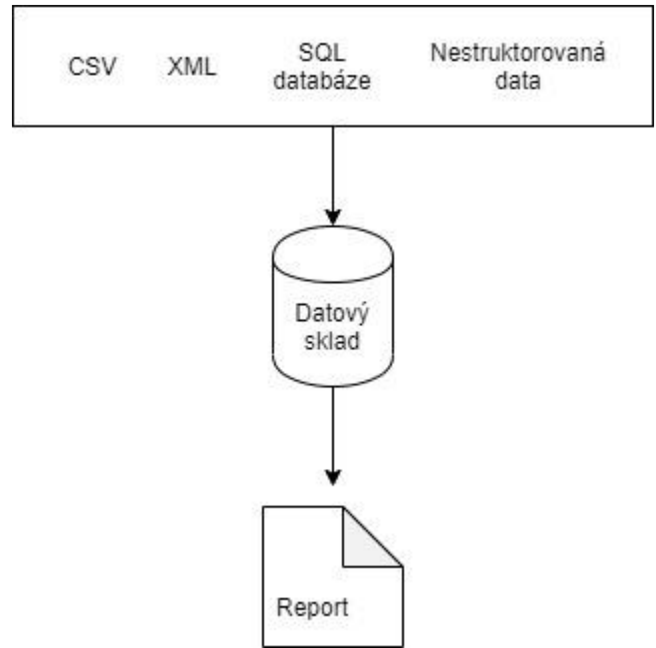
3.2 Datový sklad

V této kapitole se budu věnovat teorii datového skladu, tedy co je to datový sklad a k čemu je to vlastně dobrý. Vycházet budu hlavně z literatury Ralpha Kimballa. Zmíním zde zásadní pravidla pro práci s datovým skladem, architekturu datového skladu Hybridního modelu (HM). Poté popíši teoretickou rovinu ETL procesu a nastíním jeho implementaci v aplikaci. Detailnější implementaci se mu budu věnovat v praktické části.

3.2.1 Co to je datový sklad

Téma datového skladu, se jako takové, zabývá tříděním a správou velkého objemu dat, ať už za účelem přímého zisku nebo správného rozhodování vedoucího k zisku. Abychom mohli lépe pochopit, proč něco jako datový sklad můžeme potřebovat je potřeba umět si představit problém absence datového skladu. Pro představu uvedu příklad.

Firma A potřebuje každý měsíc svým odběratelům posílat fakturu za poskytnuté služby. Firma A nadále provozuje několik dalších strategicky důležitých systémů, které pracují s daty o odběratelích. Situaci komplikuje i to, že každý z těchto systémů produkuje výstupy v různých datových formátech. Firma A se tedy nachází v situaci, v které potřebuje systém, který pro ně automaticky každý měsíc extrahuje a vyčistí data ze zdrojových systémů, transformuje a normalizuje a nahraje do jednoho systému, který na základě těchto dat poskytne firmě A výstupy, díky kterým bude budoucí spolupráce s odběrateli hladká. Můžeme tedy říct, že firma A se nachází v situaci, kdy potřebuje datový sklad.



Obrázek 2 – princip datového skladu

Tímto principiálně velmi zjednodušeným příkladem jsem chtěl pouze demonstrovat co to je datový sklad a k čemu to může být dobré. Na začátku všeho jsou tedy nějaká nesetříděná data, která mohou mít různou podobu. Může jít o data v jiné SQL databázi, CSV soubory, XML sobory ale i nestrukturovaná data. Cílem datového skladu je tyto data utřídit a vyčistit tak, abychom byli schopní z tohoto objemu dat sestavit report, který organizaci provozující datový sklad, pomůže v efektivním rozhodování anebo prostě jen usnadní firemní procesy.

Datový sklad tak můžeme klidně chápat i jako nástroj na podporu v rozhodování. Samotný datový sklad v součtu nákladů na hardware, software a údržbu je velmi nákladný, avšak špatné rozhodnutí způsobené absencí datového skladu v podniku může být nákladnější. Datové sklady, bývají založené na transakčních databázích. Tyto databáze v sobě mohou mít více jak miliardu záznamů. Při nesprávném návrhu se můžeme dostat do situace, že dostat z nich užitečnou informaci může být až nemožný úkol. Každý datový sklad v sobě obsahuje ETL proces. ETL proces je klíčová funkce datového skladu. Kvalita návrhu a implementace ETL procesu určuje použitelnost datového skladu.

Na závěr této kapitoly přikládám formální definici dle Kimbala. Skladování dat je proces přebírání dat z databázových systémů, který extrahuje, čistí, potvrzuje a dodává zdrojová data do dimenzionálního datového skladu a poté podporuje a implementuje dotazování a analýzu za účelem rozhodování. (Kimball, a další, 2004)

3.2.2 Požadavky na datový sklad

Požadavky na datový sklad vycházejí ze základních cílů výstavby datového skladu. Mezi základní cíle patří skladování dat, ale i business inteligence. Informace uložené v datovém skladu musí být snadno přístupné, obsah datového skladu musí být pochopitelný. Data musí být jasně srozumitelná nejen pro vývojáře, ale i pro koncového zákazníka. Datová základna by měla odpovídat obchodnímu slovníku zákazníka. Zákazník by měl mít možnost data libovolně kombinovat. Výsledky dotazů by se měly zobrazovat rychle. Zjednodušeně můžeme všechny tyto požadavky shrnout. Systém datového skladu by měl být jednoduchý a rychlý. (Ross, a další, 2013)

Datový sklad by měl poskytovat konzistentní informace. To znamená, že pojmenování a definice dat je napříč systémem stejné. Data musí být důvěryhodná, vyčištěná a s garantovanou kvalitou. (Ross, a další, 2013)

3.2.3 Architektura datového skladu HM

Když se řekne architektura software, máme namysli návrh jednotlivých komponent programu software. Tento návrh může být prováděn například pomocí různých modelovacích nástrojů a může v sobě zahrnovat i návrhy a požadavky jak softwarové,

tak i hardwarová. Tyto požadavky se mohou s časem změnit. Částečně proto se používá slovo architektura, jakožto metafora k architektuře ve stavebnictví.

Požadavky na architekturu se mohou mezi vývojáři lišit, ale pro uživatele je důležitý, aby systém měl rychlou odezvu a nebrzdil ho tak v práci. Datový sklad hybridního modelu za účelem uživatelské přívětivosti a robustnosti řešení využívá 4 vrstvou datovou architekturu.

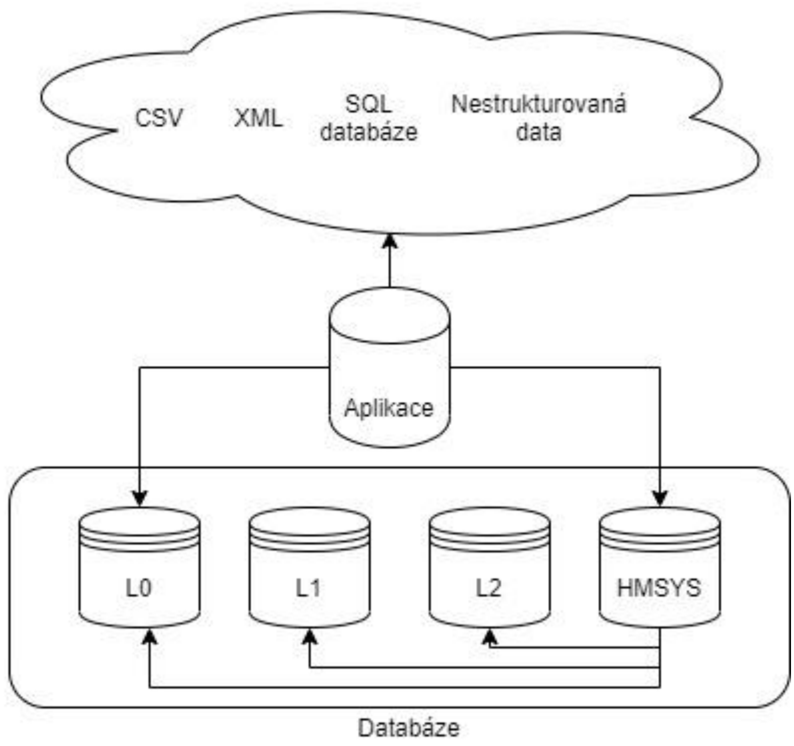
3.2.4 Čtyřvrstvá datová architektura HM

Pod názvem čtyř vrstvá datová architektura se skrývá databázový návrh, který dělí databázový celek na de-facto 4 schémata. Z praktického hlediska toto znamená, že každé schéma má jiný connection string a tváří se jako oddělený celek. Není tedy možné napřímo spustit dotaz z vrstvy nula do vrstvy jedna, takový dotaz typicky skončí chybou typu: tabulka nebo schéma neexistuje. Abychom byli schopní takovýto dotaz provést je třeba přidělit schématu práva na konkrétní tabulku (příkaz GRANT). Z principu návrhu databáze a datového skladu bychom neměli přidělovat práva jak se nám to zrovna hodí.

Rozdělení do čtyř schémat má následující dělení:

- Vrstva L0 – vrstva sloužící pro import dat a jejich validace.
- Vrstva L1 – vrstva sloužící pro normalizaci dat z vrstvy L0.
- Vrstva L2 – vrstva obsahující odvozená data, vzniklá z výpočtů nad daty z L1
- Vrstva HMSYS – systémová vrstva obsahuje data, která jsou nezbytná pro běh aplikace nad datovým skladem, obsahuje také výpočetní procedury a balíčky funkcí v PL/SQL.

Aplikace HM v sobě obsahuje connect stringy do všech vrstev databáze i když teoreticky by měla potřebovat pouze připojení do HMSYS a L0. Dotazy prováděné do ostatních vrstev jsou prováděny skrze HMSYS, kterému byly přiděleny oprávnění, a to i v případě, že bychom potřebovali spojit tabulky z L0 a HMSYS. Zjednodušený princip fungování naznačují na obrázku.



Obrázek 3 - čtyřvrstvá architektura

3.2.5 ETL proces

Když už máme datový sklad navržený a softwarově postavený, je potřeba ho naplnit daty. V první řadě tím můžeme ověřit, jak návrh datového skladu obstojí v praxi a v druhé řadě tím zjistíme, jak případně ho budeme muset ještě upravit. Plnění datového skladu je netriviální úkol, který (překvapivě) může zabrat spoustu času i když se z počátku může zdát, že je to vlastně jen kopírování dat. Jedná se tedy o časově, a tedy i finančně nákladnou součást datového skladu. Každý datový sklad tedy musí mít ETL proces, pomocí něhož je plněn. Proces ETL, znamená proces Extract-Transform-Load, v češtině také někdy bývá označován jako datová pumpa. Jedná se o proces, který se dá rozdělit na tři samostatné bloky, ale dohromady tvoří jeden nástroj. (Kimball, a další, 2004)

- Extract – extrahování/čtení/čerpání dat ze zdroje mimo datový sklad. Data mohou být v podobě SQL dotazu, XML, CSV a jiný.
- Transform – transformace – proces přizpůsobení dat cílové databázi. V praktické časti tomuto procesu říkám nemanipulativní validace.
- Load – aplikace – proces kdy se zapisují data do databáze.

Zdroje a zdrojová data mají dost často různorodou povahu, zdroje dat mohou být i různé operační systémy a databáze, což může mít negativní vliv na konzistenci dat. Při návrhu procesu čerpání je třeba všech těchto aspektů zvážit a dle nich poté navrhnout jakých metod využijeme. Další komplikace může přinést např. čerpání starších dat či jejich množství. V neposlední řadě musíme dbát na ukládání periodických dat. V datovém skladu se data ukládají zásadně tak, abychom nepřepsali nebo nesmazali stávající data.

Jak jsem naznačil výše extrahování, transformace a aplikace představují dílčí úkoly. V následujících podkapitolách každý tento proces rozeberu a poté popíši jeho aplikovanou verzi v Hybridním modelu.

3.2.5.1 **Extrahování**

Extrahování jako takové začíná studiem zdrojů dat. Tato činnost je nezbytně nutná, protože jak jsem popsal výše, zdroje jsou dost často různorodé povahy a proces extrahování musí zajistit hladký průběh následujících procesů transformace a aplikace.

Pokud bychom se zabývali pracností ETL procesů, tak proces extrakce klidně můžeme označit za nejpracnější, a tedy i časově nejnáročnější. (Kimball, a další, 2004)

Proces extrakce můžeme navrhnout sebe-lépe, ovšem výsledná kvalita dat se bude vždy odvíjet od kvality dat na vstupu. V případě, že jsou data nevyhovující, např.: úplně chybí data v jednom sloupci, tak je lepší data vůbec neimportovat. Pakliže nastane taková to situace, tak o ní musí existovat záznam v logu. O logování píši níže. Je vždy důležité si při návrhu extrahování položit otázku, kde je hranice kvality vstupních dat. Snaha o importování čehokoliv s sebou přináší spoustu problémů, které musí být ošetřeny v kódu. Tím se ovšem zvyšuje pravděpodobnost chybovosti. Je proto dobré provádět analýzu obsahu dat. Porozumění obsahu dat je zásadní pro určení toho nejlepšího přístupu k extrahování. Až když začneme pracovat s daty zjistíme jaké anomálie existují. Mezi anomálie patří:

- NULL hodnoty. Neošetřená hodnota NULL může shodit jakýkoli ETL proces, proto NULL hodnoty představují největší riziko. Spojení dvou nebo více tabulek na základě sloupce, který obsahuje NULL hodnoty způsobí ztrátu dat! Důležité je si pamatovat, že v relační databázi NULL se nerovná NULL
- Datumy v nedatumových polích. Datumy jsou velmi zvláštními prvky, protože jsou to jediné logické prvky, které mohou být v různých formátech, doslova obsahující různé hodnoty a mající přesně stejný význam. Databáze podporují zobrazování datumů v různých formátech, ale existují i aplikace, které ukládají datum do textových polí.

Jako nejčernější scénář si můžeme představit zamíchání dat a z toho plynoucí rozčarování koncových uživatelů nad výsledky.

Proces extrahování by v sobě měl také obsahovat kontrolu, případně být dopředu navržen, na kódování zdrojových dat. Pakliže jsou zdrojová data generována na jiném operačním systému, než je aplikace se datovým skladem, je potřeba pečlivě kontrolovat, jak který systém interpretuje určité znaky.

3.2.5.1.1 Datové zdroje

Proces extrakce pracuje s více datovými zdroji. Může se jednat o nové moderní systémy, ale i o staré proprietární systémy, vyžadující znalost starých programovacích jazyků. Při psaní a vymýšlení procesu extrakce se musíme připravit na práci se zdrojovými systémy omezenými na konkrétní jazyky. Ke standardům patří COBOL, FOCUS, EasyTrieve, PL/SQL, Transact-SQL a RPG. Důležité datové struktury procesu ETL jsou:

- ploché soubory
- datové sady XML
- nezávislé – pracovní tabulky SŘDB (systém řízení báze dat)
- ERD

V následujícím odstavci popíši datové zdroje, které používá systém HM. Ke každému zdroji popíši typické problémy, s kterými se můžeme a budeme setkávat. (Kimball, a další, 2004)

3.2.5.1.1.1 Soubory s oddělovači

Soubory s oddělovači, označovány jako CSV, jsou textové sobory, v kterých jsou datová pole oddělena některým z oddělovačů. Oddělovačem může být čárka, středník, popřípadě i tabulátor, řádky jsou oddělené enterem. Tento oddělovač pak říká, kde datové pole začíná a kde končí. Pakliže datové pole obsahuje v sobě oddělovač musí být toto datové pole v uvozovkách. CSV soubory můžeme snadno identifikovat podle koncovky souboru .csv, ale nelze se na to stoprocentně spolehnout, protože můžou klidně mít koncovku .txt anebo být prostě bez koncovky. Zpravidla první řádek obsahuje názvy datových polí(sloupců), ETL proces toto musí umět rozlišit, případně dle vzorových dat se dá usoudit, že tomu tak je vždy a první řádek můžeme ignorovat. Strukturu souboru si můžeme jednoduše představit jako tabulku např. v MS Excel. (Kimball, a další, 2004)

Při práci s CSV, je krom oddělovačů, potřeba dát si pozor na kódování souboru. Je třeba dát si pozor na znakovou sadou, obzvlášť v případě že CSV obsahuje jiné znaky

než ze základní ASCII (tyto znaky mívají znakové sady většinou stejné). Jako každý zdroj dat, CSV má svoje výhody a nevýhody.

Výhody:

- Jednoduchost.
- Rychlost.
- Předpřipravené funkce pro práci s CSV snad v každém programovacím jazyku.
- Možnost paralelního zpracování. (více souborů najednou)

Nevýhody:

- Složitější vyhledávání.
- Větší náchylnost k chybám.

3.2.5.1.1.2 SQL databáze

Použití databáze jako zdroje primárních dat se mi jeví jako nejoptimálnější zdroj. V případě, že jsou data dobře strukturovaná a neobsahují fatální chyby, tak použitím SQL databáze jako primárního zdroje, nám ušetří práci s programováním ETL. Největší výhodu v použití SQL databáze jako zdroje dat shledávám v datové jednoznačnosti. V SQL databázi najdeme snadno informace o datech například dle názvu sloupců, datových tipech a jejich případných délek. Velmi snadno lze odvodit jednotlivé kardinality, některé systémy umí i vygenerovat ERD a tím nám usnadní čtení datové struktury. Jako velkou výhodu shledávám snadné vyhledávání v datech – SQL patří mezi nejznámější programovací jazyky, a i relativní laik je schopný napsat dotaz do databáze. Poslední z výhod SQL databáze, kterou zde zmíním je možnost vytváření pohledů. Pohledem si můžeme přímo na primárních datech předpřipravit potřebná data a nemusíme je poté skládat v procesu transformace, jako tomu může být u jiných zdrojů.

ODBC (open database connectivity) byl vytvořen za účelem umožnění přístupu uživatelům k databázi z jejich Windows aplikací. Hlavní myšlenkou ODBC byla přenositelnost aplikací, což znamená, že pokud se v aplikaci změní databáze z třeba DB2 na Oracle aplikační vrstva se nezmění, neprogramuje a nemusí znova kompilovat. Místo toho se jednoduše změní ovladač ODBC. Nevýhodou ODBC je pak pokles výkonu. (Kimball, a další, 2004)

SQL databáze je velmi rychlý zdroj dat. Vhodně sestaveným SELECT dotazem se můžeme rychle dostat k potřebným datům, avšak i zde narazíme na úskalí, na která je třeba dát si pozor. Měli bychom dbát na to, aby sloupce v selekci byly indexované, v opačném případě by mohl dotaz skenovat celou databázi a dotaz by byl časově velmi náročný. Data pro extrakci bychom měli filtrovat zásadně už v dotazu do zdrojových dat, určitě bychom neměli načítat všechno a filtrovat to později. Opatrní bychom měli být i při použití klauzulí DISTINCT, SET, UNION, MINUS a INTERSECT, protože jsou pomalá. Někdy se jim samozřejmě nevyhneme, v tom případě bychom ale měli využít indexů. Vyhnou bychom se měli i používání negací (NOT, „<>“) databáze se může rozhodnout místo použití indexů prohledat celou tabulku. (Kimball, a další, 2004)

3.2.5.1.1.3 Webová služba

Webové služby poskytují standardní prostředky pro spolupráci mezi různými softwarovými aplikacemi, které mohou běžet i na různých platformách. Rozhraní webové služby je popsáno souborem na bázi XML, nesoucí zkratku WSDL (web service description language). Pro výměnu zpráv mezi klientským a serverovým počítačem se používá protokol SOAP. SOAP poskytuje standardní, rozšířitelný, skladatelný framework pro výměnu zpráv v XML. (W3C, 2004)

Vazba WSDL SOAP může být buď RPC (remote procedure call) anebo document. Oba tyto „styly“ WSDL mohou být ještě děleny na encoded anebo literal. Mezi těmito zápisy WSDL jsou marginální rozdíly, které ovšem mohou způsobit, že služba na klientském stroji nezafunguje, za předpokladu, že klient očekával jiný formát.

Import dat pomocí webové služby může být velmi výhodný. Veškeré metody a datový typy jsou popsaný ve WSDL souboru. Tento soubor je i pro člověka dobře čitelný, takže lze navrhnout klienta buď čistě dle něj, ale věřím, že většina programátorů spíše využije možnosti automatického vygenerování, nějakým generátorem. Oproti přístupu s CSV nabízí webová služba výhodu v jednoznačných návratových typech, na druhou stranu nelze bez pomocných mechanismů kontrolovat, jestli jsou data všechna anebo jestli nedošlo během stahování odpovědi ke ztrátě připojení. Další důležitá výhoda nesporně je možnost přímé kontroly funkce služby a výstupních dat přes testovací software např. SOAP UI.

3.2.5.1.2 Logování procesu extrakce

Další důležitou funkcionalitou celého procesu ETL je nepochybně logování. Protože ji nejvíce využijeme během procesu extrakce, tak se jí budu věnovat tady. Data ze vstupních datových zdrojů nesmí být v žádném případě nijak měněna nebo manipulována, musíme logovat jakoukoliv nesrovnalost, kterou v datech nalezneme. Důležitou součástí procesu extrahování se tedy stávají logy – logovací funkce. Tyto logovací funkce mohou ve své podstatě logovat cokoliv. Já ze své zkušenosti zde doporučím logy, které mi velmi usnadnily hledání chyb, ztracených dat a další podobně nepříjemné chyby. Logování by mělo v sobě mít především informace o počtu dat, případně velikost v Kbi a časové razítko, kdy byly data publikována a datum jejich stažení. Výhodné také je dělat logy jak do souborů, tak i do databáze, a to především z důvodu, že když spadne připojení, do databáze už nic nezalogujeme. Souborové logy bychom neměli kumulovat v jednom souboru, ale dělit je alespoň dle periodicity importů dat a to čistě z důvodu pozdějšího usnadnění při hledání v lozích.

3.2.5.2 **Transformace**

Dalším krokem v ETL procesu je transformace. V závislosti na kvalitě vstupních datech můžeme odvozovat pracnost tohoto úkolu. Proces transformace si lze představit také jako proces čištění a konformace. Metadata generovaná v rámci kroků čištění a přizpůsobení doprovázejí reálná data až na plochu uživatele. Než se pustíme do samotné transformace za pomocí různých validací dat, tak si definujeme kvalitu dat. Kvalitní data jsou:

- Přesná. Hodnoty a popisy v datech popisují data pravdivě a uvěřitelně. Pokud se například vlak pohyboval na území Česka v datech bude uvedena oblast ČR.
- Jednoznačná. Hodnoty a popisy v datech mají pouze jeden význam a nelze je interpretovat jinak. Vlak jedoucí do stanice nesoucí nějaký název/kód se těžko může nacházet na dvou místech zároveň. A stejně to musí platit v datech.
- Konzistentní. Hodnoty a popisy v datech používají jednu konstantní notační konvenci. Například, oblasti kde se vlaky pohybují mohou být značeny zkratkami CR, SK, DE, případně – pokud údaj chybí nebo celými názvy.

Aby byly názvy konzistentní, musí používat jednu jedinou konvenci a držet se jí.

- Kompletní. Existují dva aspekty úplnosti. První je zajištění toho, aby jednotlivé hodnoty a popisy v datech byly definovány (nenulové, neprázdné) pro každou instanci. Druhý aspekt je že agregovaný počet záznamů je kompletní.

V následujících podkapitolách rozeberu možnosti, jak lze vhodně provést transformaci dat.

3.2.5.2.1 Datová normalizace

Datová normalizace je proces, při kterém dochází k separaci souvislých dat a jejich rozdělení do zvláštních tabulek, podle stanovený pravidel normálních forem. Normalizace má za cíl odstranění redundance dat a zajištění konzistence. Dalším jevem normalizace je nárůst skutečného počtu záznamů v databázi. Po normalizaci může vzniknout více tabulek, tyto tabulky jsou mezi sebou provázány klíči jakožto identifikátory odkazující se cizí tabulky. Literatury obvykle uvádí 5 normálních forem, přičemž mezi 3. a 4. se nachází Boyceho-Coddova normální forma, která bývá označována jako 3,5NF. V praxi nám obvykle postačí normalizovat do 3.NF. Pro lepší demonstraci uvedu jednoduchý příklad. (Zendulka, 2015)

Představme si tabulku uživatelů se základními informacemi jméno, příjmení a adresa.

Uživatel	Bydliště
Pavel Novák	Nikoly Tesly, Praha 6 160 00

Tabulka 1 - před normalizací

Informace obsažené v takovéto tabulce jsou z pohledu systému těžko uchopitelné. Např.: vyhledávání dat v takto postavené struktuře by bylo velmi složité. Z hlediska datového skladu pak takováto tabulka nepokrývá situaci, že se uživatel přestěhuje a my bychom chtěli mít uloženou informaci o současném, ale i bývalém bydlišti.

Na zjednodušeném příkladu ukážu normalizaci v krocích. Nejprve rozdělení dat tak aby byla atomická.

Jméno	Příjmení	Adresa	Město	PSČ
Pavel	Novák	Nikoly Tesly	Praha 6	16000

Tabulka 2 - po normalizaci

Následně tabulku rozdělím na dvě tabulky uživatel a bydliště. Poté oběma tabulkám přidělím primární klíče, tabulce bydliště přidělím ještě cizí klíč odkazující se do tabulky uživatel. Posledním krokem je přidání sloupce f_validní do tabulky bydliště. Tento příznak indikuje, jestli je adresa platná či ne. Příznak by klidně mohl být i v tabulce uživatel a indikovat tak, jestli je uživatel aktivní.

Uživatel_klíč	Jméno	Příjmení
1	Pavel	Novák

Tabulka 3 - tabulka uživatelů konečná normalizace

Bydliště_klíč	Uživatel_klíč	Adresa	Město	PSČ	f_validní
1	1	Nikoly Tesly	Praha 6	16000	1

Tabulka 4 - tabulka bydliště konečná normalizace

3.2.5.2.2 Validace a čištění dat

V závislosti na využití datového skladu se mohou hodit ošetřovací algoritmy, po jejichž aplikaci budeme moci zaručit určitou kvalitu dat. Požadavky na kvalitu dat se mohou lišit, ale z principu můžeme data rozdělit do tří skupin.

- Správná data – taková data nejsou na první pohled poškozená a vypadají formálně správně. Validátor v procesu transformace tyto data ve své první iteraci vyhodnotí jako správná bez nutnosti korekce.
- Opravitelná data – data jsou jistým způsobem poškozená, ale existují mechanismy, jak je opravit. Například pokud máme v datech čas, který jde po pěti minutách (ale může obsahovat i hodinový skok) a následující čas chybí lze do určitého množství řádků pětiminutovou posloupnost.
- Neopravitelná data – v datech chybí důležitý údaje anebo obsahují díry. Taková data nelze nijak opravit a musí se označit jako neplatná.

Validace samotných dat může probíhat nad daty načtenými v operační paměti. Pakliže data jsou validována předem v operační paměti, tak se připravíme o kompletní auditní stopu. Tedy nebude v budoucnu možné vypátrat, jak vypadaly původní data a proč

k validaci došlo takovým způsobem, jak došlo. Je tedy výhodnější data do databáze nahrát v procesu extrakce a poté provádět validace nad nahranými daty. Z klasifikace dat, kterou popsal výše je výhodné validace stavět tak, aby byly z dat nejdříve vyčleněna (označená jako neplatná) data neopravitelná. Následně nad opravitelnými daty provést jejich korekci, a nakonec spustit validaci která ověřuje formální správnost nad správnými daty.

3.2.5.2.3 Typy chyb a jejich korekce

Při procesu extrakce jsme již zajistili, že data je vůbec možné do datového skladu uložit. Je vhodné již při procesu extrakce také vytipovat jaké typy chyb se v importovaných datech nalézají. Uvedu zde výčet chyb a návrh na jejich korekci.

- Duplicity v datech – může nastat situace, že v importovaných záznamech existují dva totožné záznamy. Transformační proces v sobě musí obsahovat logiku, která rozhodne, který z těchto dvou záznamů se zneplatní (případně nebude importovat).
- Díry v datech – data obsahují záznamy, kde v sloupci chybí podstatná informace. Různými metodami (interpolace, odhad z předchozího a následujícího, průměr) se dají hodnoty doplnit. Avšak toto je možné jen některých dat. Tam kde to možné není je třeba data zneplatnit.
- Nekonzistence – je třeba zajistit, aby záznamy v sloupcích měly stejný formát. Například datový sklad může vyžadovat, aby všechny záznamy o trakcích byly velkými písmeny a dohromady. Nepřípustné jsou záznamy Ac16, ac16, AC 16.

3.2.5.3 **Load**

Proces ukládání dat do databáze může následovat ihned po extrakci nebo jí může předcházet transformace. V každém případě se jedná o proces, který má také svoje úskalí a je potřeba dát si na pár věcí pozor.

3.2.5.3.1 Přípravy před loadem dat

V této podkapitole bych rád zmínil několik tipů pro nahrávání dat do databáze. Samotný proces nahrávání musíme postavit tak aby byl co nejvíce efektivní. To s sebou samozřejmě přináší spoustu překážek.

3.2.5.3.1.1 Správa indexů

Indexy zvyšují rychlost při dotazování se do databáze, ale výrazně snižují rychlost při ukládání do databáze. Tabulky obsahující mnoho indexů mohou náš proces loadu téměř zastavit, když s nimi správně nenaložíme. Předtím, než se začneme nahrávat data do tabulky, bychom měli všechny indexy dropnout, poté provést load a po nahrání dat, tabulku opět oindexovat. Ve zkratce by náš proces mohl vypadat takto:

- Rozdělit update a insert příkazy od sebe.
- Dropnout indexy nepotřebný pro update příkazy.
- Provést update příkazy.
- Dropnout všechny ostatní indexy.
- Provést insert.
- Znova oindexovat tabulku. (Kimball, a další, 2004)

3.2.5.3.1.2 Správa partitions

Partitions v databázi znamená, že tabulka jako celek je rozdělená dle klíčů na několik menších dílů. Můžeme si to představit jako Ottův slovník naučný, který je rozdělen abecedně rozdělen místo toho, aby tvořil jednu obří bichli. Pro partitions se v češtině neuchytil žádný název, resp. používá se počeštění anglického, tedy partyšnování.

Partitions mají ohromnou výhodu v tom, že dokážou dramaticky zvýšit rychlost vyhledávání v databázi. Uvedu příklad. Představme si tabulku, do které jsme celý rok importovali každý měsíc zhruba 5 milionů záznamů. Tabulku jsme partyšnovali po dnech v každém měsíci, klíč partition pro 30.1.2020 vypadá tedy tako 20200131. Tabulka má za rok tedy zhruba 60 miliónů záznamů. Pokud bychom chtěli vyhledat záznam z jednoho dnu v určitém měsíci, čekali bychom na výsledek poměrně dlouho. Díky partitions můžeme dotaz postavit tak, že místo vyhledávání v 60 milionech budeme vyhledávat v pár desítkách tisíců. V selekci se stačí omezit na ten jeden určitý den a SŘBD bude vyhledávat v malém souboru dat.

Daní za partitons je nutnost jejich vytváření v load procesu. Bez vytvořeného partition nelze do tabulky vkládat data. Při navrhování ETL procesu na to musíme myslet a vhodným způsobem vytvářet partitons před ukládáním dat do cílové tabulky. (Kimball, a další, 2004)

3.2.5.3.2 Nahrávání dat

Samotné nahrávání dat v sobě může zahrnovat několik výzev. Našim primárním cílem je zvládnutí a nahrání velkého balíku dat do databáze.

- Oddělení insertů od updatů – Některé databáze v sobě obsahují funkcionalitu update else insert. Tato funkcionalita je vynikající a usnadňující datový tok, ale je velmi pomalá. ETL proces by tedy v sobě měl obsahovat funkci, která oddělí updaty od insertů. Tím se operace nahrávání dat rozdělí na dva úkoly, tedy update existujících a poté bulk-insert.
- Používání bulk-insertů místo klasických insertů – použitím bulk-insertů snížíme zátěž na databázový stroj a zvýšíme rychlost loadu dat.
- Paralelizace – pakliže to datová základna umožňuje, je vhodné data rozdělit a nahrávat je paralelně. Paralelizace má příznivý vliv na rychlost ETL procesu, avšak je třeba dát si pozor na hardwarové limity databázového stroje.
- Minimizace updatů – update na velkým objemem dat může být velmi časově nákladný. Jejich minimizace by tedy měla být naší prioritou. Pakliže potřebujeme udělat opravdu hodně updatů může někdy vhodnější řešení provést truncate celé tabulky a pomocí bulk-insertu ji nahrát celou znovu. (Kimball, a další, 2004)

3.2.5.3.3 Inkrementální load

Inkrementální load je proces, který se spouští periodicky, aby datový sklad byl synchronizovaný se svým zdrojem. Interval synchronizace může být libovolný nebo v přímém čase. ETL proces který používá inkrementální load musí u dat rozlišovat čas od a čas do. Díky tomu bude schopen provádět malé inkrementální přírůstky a nepřepisovat tak historická data.

Co když ale potřebuje z nějakého důvodu provést celý load znovu? K datům, co jsou již v databázi můžeme přistoupit třemi způsoby.

- Zneplatnění předchozích záznamů. Zneplatněním předchozího záznamu mám na mysli, že u všech minulých záznamů vnikne příznak (sloupec), který říká tenhle záznam není platný, byl nahrazen jiným – novějším. Zneplatňování se hodí zejména tam, kde je třeba držek kompletní auditní stopu, případně tam, kde jsou data využívána např.: k analýzám nějakých chyb a jejich oprav.

Naopak zneplaňování se příliš nehodí tam, kde jsou opravdu velké objemy dat (řádově stovky milionu řádků). Zneplatňování a nahrávání dat stále dokola způsobí výkonnostní propad a znemožní jakékoliv vyhledávání v datech.

- Update dat. V některých ojedinělých případech může být update dat výhodný, ale ve většině případech, obzvláště tam kde jsou vyšší objemy dat, narazíme na nízký výkon příkazu update a tedy jeho nepoužitelnost.
- Mazání (delete) dat. Ačkoliv je mazání dat proti samotné myšlence datového skladu, je často naprosto běžnou praxí. Je tomu tak, protože se jedná o nejjednodušší nápravu chyb v datech. Má to samozřejmě nevýhody, jako jsou ztráta auditní stopy a budoucí ne dohledatelnost dat. Samozřejmě, lze k tomu přistoupit například verzováním dat a označováním příznakem deleted. Tím se dostávám ke dvěma přístupům mazání a to jsou:
    a. Fyzické mazání dat – Ve většině případů nikdo nebude chtít vidět data, která mají být smazána. Pokud to tedy business-inteligence dovoluje můžeme všechna data smazat.
    b. Logické smazání – Je považováno za bezpečné mazání dat. Provádí se tak, že založíme nadřazenou tabulku, která bude obsahovat sloupec deleted. Podřízená tabulka pak musí obsahovat cizí klíč, který je primárním klíčem nadřazené tabulky. (Kimball, a další, 2004)

3.3 Použité vývojové nástroje

Pro naprogramování jakékoliv aplikace by nám teoreticky měl stačit jednoduchý textový editor, kompilátor a konzole, v které program otestujeme. Takovýto styl vytváření aplikací, ale rozhodně není komfortní a při hlubším zanořování může docházet k zvýšené chybovosti a složitému ladění chyb. Proto programátoři využívají vývojových studií/editorů. Tyto pokročilé editory často nabízí funkce jako zvýrazňování závorek, podbarvování překlepů, kontrolu syntaxe a podobné. Vývoj robustní aplikace není triviální, a proto bych chtěl v následujících podkapitolách doporučit nástroje pro vývoj na základě vlastních zkušeností.

3.3.1 Nástroj na vývoj aplikace

Úmyslně jsem zvolil nadpis jako vývoj aplikace. Hybridní model se totiž skládá ze dvou částí. První část je aplikační a druhá je databázová. Aplikační část, ačkoliv je složená ze PHP7, HTML5 a Javascriptu, lze vytvářet pomocí jednoho vývojového studia. Zde bych doporučil vývojové studio Visual Studio Code. Visual Studio Code je rychlý editor, který je zadarmo i pro komerční užití. Nabízí spoustu funkcí jako je našeptávání, kontrolu syntaxe, ale i pokročilý refaktoring. Studio obsahuje svůj vlastní plug-in instalátor, pomocí jehož je možné nainstalovat do studia spousty vylepšení, díky nimž lze ve studiu programovat jakýkoliv jazyk. Tato vylepšení vyvíjí bohatá komunita programátorů a spousta z nich je zadarmo a na srovnatelné úrovni s komerčními nástroji. (Microsoft, 2020)

3.3.2 Nástroj na vývoj databáze

Protože ve své práci pracuji s databází Oracle nemohu zde doporučit nic jiného než Oracle sql developer. Samozřejmě i zde platí, že jsou vývojáři, co nedají dopustit na konzoli, ovšem použitím Oracle sql developeru, si lze velice usnadnit život. Program je dostupný zadarmo na stránkách Oracle, avšak pro jeho stažení je třeba se registrovat. Oracle developer má v sobě podporu na programování PL/SQL, dále nabízí funkce jako je našeptávání, automatické párování závorek nebo formátování SQL. Lze s ním ovšem i provádět pokročilé spravování databáze, jako je například sledování scheduleru. Nástroj v sobě integruje velmi dobrou podporu na psaní funkcí a procedur. (Oracle, 2020)

3.3.3 Web server

Vývoj webové aplikace vyžaduje stroj, který bude umět vyřizovat http požadavky – HTTP server. Webovou aplikace můžeme vyvíjet buď přímo na serveru, myšleno vzdáleném počítači, a nebo si nainstalovat HTTP server na svůj lokální stroj. Vývoj na lokálním stroji je pohodlnější, ale musíme si dát na pár věcí pozor. Typicky, když vyvíjíme na prostředí MS Windows, ale produkční server je na stroji kde běží systém na bázi unixu, tak aplikace fungující na lokálu nemusí na serveru fungovat. Důvodem mohou být třeba velká a malá písmena v názvech souborů (Windows toto nerozlišuje). Pro vývoj na lokálním stroji doporučuji nástroj XAMPP. Tento nástroj/program v sobě obsahuje Apache HTTP server a PHP interpreter. Chtěl bych zde upozornit, že je dobré si stáhnou XAMPP se stejnou verzí PHP jako je ta, která běží na serveru. Zároveň je

dobré mít PHP nastavené totožně se serverem, lze tak předcházet problémům při deploymentu. (Apachefriends, 2020)

3.3.4 Správce projektových knihoven

Každý, kdo pracoval na nějakém projektu určitě narazil na situaci, že potřeboval funkcionality, které svým rozsahem tvoří samostatnou část projektu. Soubor (balík) takovýchto funkcí se nazývá knihovny. Ne vždy máme čas vyvíjet věci od znova (vynalézat znova kolo), a tak není od věci šáhnout po nějaké veřejné nebo placené knihovně. Každému, kdo se ocitne v této situaci určitě doporučuji spravovat knihovny pomocí aplikace. Správa pomocí aplikace velmi usnadní jejich instalaci do projektu, aktualizace, případně mazání. Já jsem využil aplikace Composer. Composer je nástroj pro správu knihoven v PHP. (Composer, 2019) Composer má na své stránce vyhledávač knihoven, kde si můžeme vyhledat co potřebujeme a poté jednoduše nainstalovat do projektu. Composeru se budu ještě věnovat v praktické části práce.

3.3.5 Nástroj na verzování softwaru

Verzování aplikace znamená uchování historie změn všech souborů aplikace. Aplikace vznikající v etapách se bude etapu od etapy lišit a abychom mezi jednotlivými verzemi v etapách mohli přeskakovat budeme potřebovat verzovací software. Já budu ve své práci používat Git, konktrétně webovou službu GitHub využívající Git. GitHub můžeme rozdělit na dvě části.

První část je webové rozhraní, lze využít pro managování projektu. Tím myslím akce typu zápis zadání, rozdělení na dílčí úkoly(issues), případně rozdělení těchto úkolů mezi programátory. Webové rozhraní umožnuje pomocí jednotlivých labelů rozlišit typ úkolu, stav daného úkolu a prioritu. Webové rozhraní dále nabízí wiki, kam můžeme psát například dokumentaci k projektu dle jednotlivých změn prezentovaných zákazníkovi.

Druhá část je samotné verzování kódu aplikace. Já k tomuto účelu využívám aplikaci Sourcetree, což je klientská aplikace pro Git. Sourcetree nabízí širokou škálu funkcí v přehledném rozhraní s jasnou vizualizací změnového stromu. Na Sourcetree jsem přešel z aplikace Github for Windows, protože nabízí mnohem lépe zpracované rozhraní a nemá očesanou funkcionalitu narozdíl od Github for Windows aplikace. Github for Windows je velmi zjednodušená aplikiace, což má za následek, že každá

složitější akce se musí dělat přes konzoli. Verzování se budu ještě věnovat v praktické částí práce.

3.3.6 FTP klient

Poté co kód na lokálním stroji naprogramuji, je potřeba jeho funkčnost ověřit na serveru. K účelům nahrávání souborů na server využívám aplikace WinScp, klient pro SFTP, FTP pro MS Windows. Kromě správy souborů na vzdáleném počítači aplikace umí i editovat práva k těmto souborům a umí i přímou editaci, což je praktické, když je potřeba udělat rychlý úpravy v kódu přímo na serveru.

3.3.7 Správa serveru

Pro správu serveru nebo spouštění jednorázových skriptů používám aplikaci Putty, pomocí kterého se připojím přes SSH na vzdálený počítač, který poté skrze konzoli mohu spravovat.

3.4 Použité technologie

V této kapitole uvedu použité technologie od programovacího jazyka, jeho technik použití a pravidel. Dále uvedu použitý knihovny(frameworky) a k nim napíši proč jsem je použil. Nakonec zde uvedu nutný základ pravidel pro databázi Oracle a jeho použití v datovém skladu.

3.4.1 Jazyk PHP

PHP je skriptovací multiplatformní programovací jazyk, který slouží k tvorbě od webových stránek, až po rozsáhlé webové aplikace. (PHP.net, 2020) Syntaxe jazyka je odvozena od jazyka C a přebírá z něj i některé funkce. V PHP lze psát jak desktopovou aplikaci (např. spuštění skriptů přes konzoli) tak webovou.

3.4.1.1 Objektově orientované programování (OOP) v PHP

Pravděpodobně proto, že je PHP relativně jednoduchý jazyk se spoustou již připravených funkcí, může programátory lákat ke psaní špagetového kódu. Takto napsaný kód je těžko udržitelný a znovu-použitelnost je nulová. Jedním ze způsobů, jak se tomuto vyhnout je OOP. Zapouzdření, znovu-použitelnost, dědičnost a polymorfizmus jsou základní pilíře OOP.

3.4.1.1.1 Zapouzdření

Pakliže objekt obsahuje atributy/metody, které nechceme, aby byly z venku přístupné můžeme takto učinit pomocí modifikátorů přístupnosti.

- private atributy/metody nejsou z venku viditelné
- protected atributy/metody nejsou z venku viditelné, ovšem jsou viditelné třídě dědící (potomkovi)
- public atributy/metody jsou viditelné všem

Kdybychom napsali třídu např.:

```
class test{
  function otestuj(){
    return "hello";
  }
}
```

tak metoda otestuj bez modifikátoru přístupnosti bude automaticky public.

3.4.1.1.2 Znovupoužitelnost

Jedná se o styl programování, kdy jednou vzniklá metoda v rámci třídy je používána dále aplikací bez nutnosti opakovaného programování. Programátor, který převezme kód nezkoumá, kód objektu, ale pouze implementuje.

3.4.1.1.3 Dědičnost

Představme si, že máme dvě třídy A a B. Dědičnost pak definuje vztah mezi těmito třídami, kdy třída B dědí od třídy A. Třída B poté zdědí všechny public a protected atributy a metody třídy A. Pakliže třída B nepřetíží metody třídy A použijí se metody třídy A. (Vrána, 2017)

V PHP je dědičnost realizována pomocí klíčového slova extends.

```
class A{

}

class B extends A{

}
```

3.4.1.1.4 Polymorfizmus

Principem polymorfizmu je, že metoda může být volána pod stejným jménem z odvozených objektů, avšak provádět odlišnou činnost. Rozšířením příkladu z předchozího odstavce o polymorfizmus dostaneme:

```
class A{
  function ohlasSe(){
    return "Ja jsem A";
  }
}

class B extends A{
  function ohlasSe(){
    return "Ja jsem B";
  }
}
```

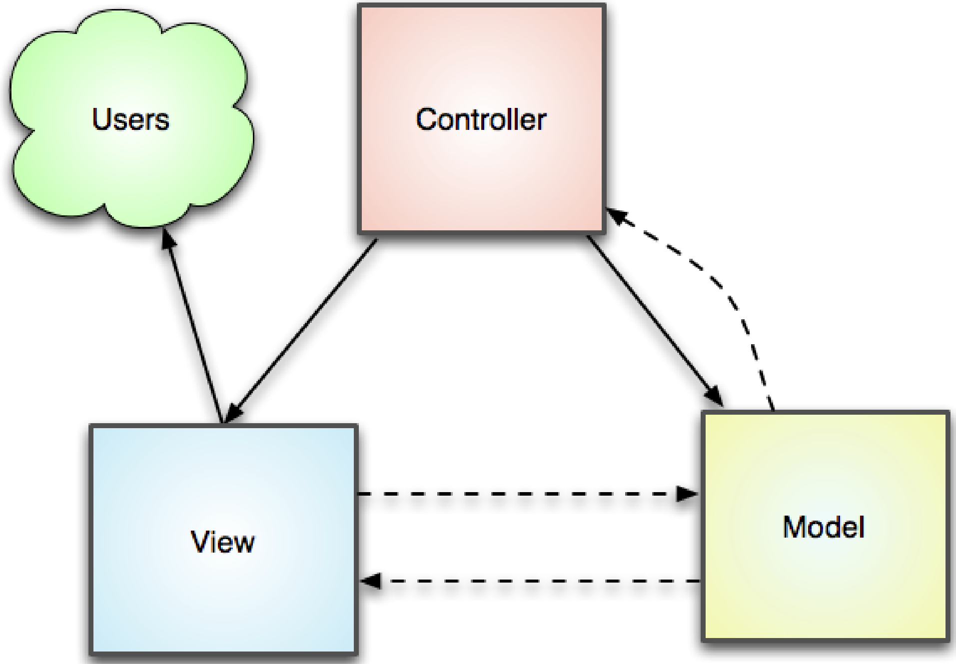
PHP umožňuje pomocí klíčového slova final zakázat přetěžování. K přetížení metod předka dochází automaticky, není třeba psát dalších klíčových slov jako je například override v jiných jazycích. Všechno, co je protected nebo public můžeme v potomkovi automaticky přetížit. Je třeba si dávat pozor a pečlivě používat modifikátory přístupu, případně udělat metody final. Zároveň bych zde rád poukázal na z mého pohledu nevýhodu PHP a to je nemožnost přetěžování metod rámci jedné třídy. Není možné např. vytvořit třídu s více konstruktory lišící se v parametrech. Toto lze ovšem řešit např. pomocí statických konstruktorů. Můžeme říct, že z tohoto pohledu je v PHP spíše takový pseudo polymorfismus, ale pro účely použití jazyka nám to stačí. Uvedu příklad nemožnosti přetěžování metod.

```
class B extends A{
  function ohlasSe(){
    return "Ja jsem B";
  }

  function ohlasSe($id){ //nelze, nebude fungovat
    return "Ja jsem B s ID  ".$id;
  }
}
```

3.4.1.2 **Model view controller**

Objektově orientované programování samo o sobě nezaručuje přehlednost kódu a dobrou údržbu v budoucnu. V aplikaci by se neměl míchat front-end s back-endem (HTML s PHP) v jednom PHP skriptu. Způsob, jakým jsem toho docílil je držení se pravidel model view controller. Model view controller nebo taky ve zkratce MVC je filozofie návrhu webové aplikace. Cílem tohoto návrhu je oddělení front-endu od back-endu a routingu. (Zend, 2011)



Obrázek 4 – MVC (Zend, 2011)

3.4.1.2.1 Model

Jak jsem již naznačil výše, model je tvořen jednou hlavní třídou (PHP skriptem). Obsahem tohoto skriptu jsou funkce, jejichž návratové hodnoty jsou hodnoty jsou předány kontroléru. Model využívá pomocných tříd pro zobrazování hodnot na obrazovku. Typická funkce modelu tedy obsahuje SQL, z kterého vezme data, ty zpracuje a pošle kontroléru.

3.4.1.2.2 View

Z technologií na straně serveru jsem pro view zvolil šablonovací PHP engine, Twig. Twig jsem použil, protože umožnuje efektivně oddělit front-end od back-endu, má svojí vlastní syntaxi a není možné ho míchat do jednoho souboru s PHP kódem. Zápis ve twigu umožňuje zobrazovat obsahy proměnných, iterovat nebo větvit kód. Dále umí číst atributy objektů přes jeho properties ve tvaru getHodnota(). To může být velmi praktické, protože veškeré promněné stránky můžeme schovat do jednoho objektu. Šablony je možné dynamicky skládat pomocí klíčového slova include. Díky tomu není potřeba např. kopírovat neustále se opakující se hlavičky a menu html. (Twig, 2019)

Technologií a použitým frameworkem na straně klienta je jQuery JavaScriptová knihovna. Její použití usnadňuje věci jako je manipulace s html objekty (DOM), obsluhu událostí, animace nebo AJAX volání. JQuery umí velmi pěkně obsluhovat události typu kliknutí na tlačítko a díky tomu umožňuje psaní čistšího kódu něž samotný JavaScript. (jQuery, 2019)

Poslední texhnologií ve view je použití HTML5. HTML5 přineslo nové sémantické značky, nové formulářové elementy, podporu pro grafické elementy a multimédia. Díky těmto změnám došlo k velkému posunu v přehlednosti a kvalitě psaného HTML. Použitím HTML5 se ovšem připravíme o podporu starších prohlížečů.

3.4.1.2.3 Kontrolér

Kontrolér je třída, která v sobě obsahuje funkce, co volají příslušné funkce z modelu. V praktické části ho budu také nazývat action router. Jedná se tedy o třídu, která obsluhuje události spuštěné uživatelem. Třída v sobě obsahuje definice funkcí, které musí být obsaženy v poli definic, protože z těchto definic se vybírá funkce dle příchozího parametru act (akce spuštěná uživatelem) z globálních proměnných

$_POST anebo $_GET. Kontrolér by v sobě neměl obsahovat jakoukoliv implementaci metod pro obsluhu z view. V rámci projektu je pak implementace kontroléru pouze jednou a pro všechny moduly je tento hlavní kontrolér zděděn.

3.4.1.3 **Práce s databází**

Pro práci s databází Oracle využiji framework Dibi. PHP v sobě obsahuje funkce na práci s Oracle, tyto funkce najdeme pod knihovnou OCI8. Dibi mi práci s touto knihovnou usnadní, protože dovoluje zjednodušení zápisu SQL příkazů a jejich parametrizaci. Dibi podporuje dva typy připojení k databázi a to statické a pak pomocí objektu `DibiConnection`. Statický přístup má jednu velikou výhodu a tou je ho stačí vytvořit jednou connection a poté kdekoliv v projektu mohu zavolat `dibi::` s potřebnou metodou. V případě, kdy potřebuji držet připojení do dvou různých schémat/databází použiji přístup s objektem `DibiConnection.` (Dibi, 2020)

3.4.2 Pravidla práce s databází

V tomto odstavci bych rád popsal rozdíly mezi klasickou webovou aplikací a datovým skladem. U klasické webové aplikace se budeme při návrhu databáze snažit o její normalizaci a to alespoň do BCNF. U datového skladu ovšem toto nemusí být žádoucí. Například u datového skladu nám tolik nevadí redundance dat, či naopak je žádoucí. Pro datový sklad Hybridního modelu jsou nastaveny pravidla následovně:

- u každého záznamu v DB musí existovat auditní stopa, to znamená musí existovat záznamy o tom který uživatel a v jaký čas záznam vložil
- v celé databázi není povoleno záznamy mazat. Místo mazání záznamů se záznamy takzvaně zneplatňují. Zneplatnění musí také splňovat požadavky na auditní stopu.

4 Vlastní práce

4.1 Stavba aplikace na zelené louce

Programování (skoro) každé aplikace začíná zadáním. Máme tedy před sebou zadání a prázdnou projektovou složku. Nyní je tedy vhodná chvíle zamyslet se nad tím co všechno budu pro splnění zadání potřebovat a jaké technologie k tomu budu potřebovat.

4.1.1 Aplikační prostředí

Pro Hybridní model byla zřízena tři pracovní prostředí, tedy aplikační server a databáze, každé má svoje pravidla, kterých je třeba se držet.

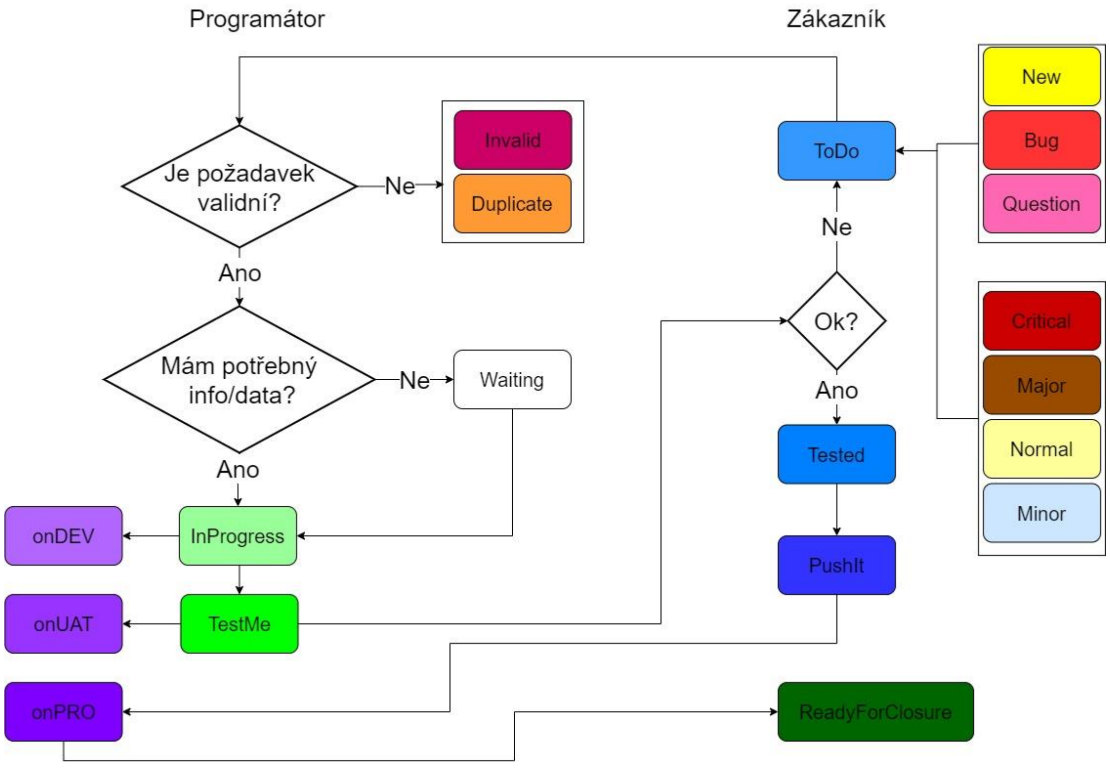
1. DEV – developerské prostředí. Jedná se o prostředí určené programátorům. Programátoři zde mohou provádět experimenty a testy a výjimečně něco prezentovat. DEV je prostředí kde je povolené v databázi libovolně mazat.
2. UAT – (User Acceptance Testing) prostředí pro uživatelské testy a akceptaci. Programátoři zde mohou také libovolně nahrávat kódy, ale pouze takové, které byly řádně otestovány na funkčnost ideálně na DEV. Uživatelé si zde zkoušejí nově vzniklé moduly nebo případné opravy chyb. Když uživatelé akceptují, kód splňuje požadavky a zadání, vytvoří se balíček, který je nasazen na PRO. V opačném případě se opakuje kolečko DEV-UAT. Na UAT se bez povolení nesmí nic mazat ani spouštět.
3. PRO – produkční prostředí. Pro produkční prostředí platí přísná pravidla, která musí být dodržována, v opačném případě hrozí pokuty a různé penále. Nahrávání na produkci probíhá v předem odsouhlasený čas. V tento čas je ohlášená odstávka produkčního prostředí a na produkci jsou nahrány akceptované kódy z UAT. Pravidla ohledně spouštění a mazání jsou stejná jako na UAT, ovšem na PRO jsou obvykle vypnuty různé testovací funkce.

4.1.2 GitHub

Každé zadání se skládá z jednotlivých úkolů. V případě, že zadání vypracováváme sami, je asi zbytečné ho nějakými způsoby dále dělit. Přidělávali bychom si akorát práci navíc. GitHub v takovém případě využijeme při zpracování prací navíc nebo separátních úkolů, které musí být schvalovány zákazníkem každý zvlášť. V opačném

případě, kdy máme k dispozici team je GitHub nebo jemu podobný software nezbytností.

GitHub může sloužit i jako nástroj pro komunikaci se zákazníkem. V takovém případě je dobře mít dohodnutý životní cyklus issue (issue je označení pro úkol na GitHubu). Při správně nastaveném životním cyklu se velmi očistí komunikace a jednotlivé kroky vývoje jsou transparentní. V GitHubu pro komunikaci se zákazníkem byl zaveden následující životní cyklus.



Obrázek 5 - životní cyklus issue

- New – nový požadavek, programování nové funkcionality do aplikace
- Bug – chyba, kterou je třeba opravit.
- Question – dotaz na zákazníka nebo dodavatele.

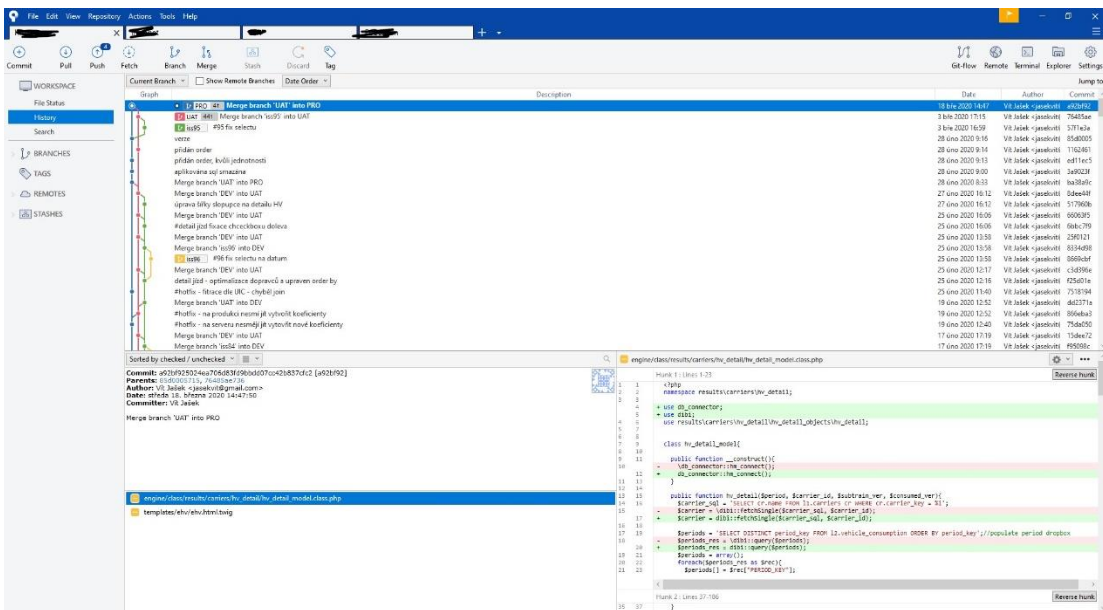
Tyto tři označení mohou mít na sebe navázanou prioritu.

A. Critical – kritická priorita, obyčejně tuto prioritu dostávají problémy bránící v práci s aplikací. Nutnost řešit v nejkratším možném termínu.
B. Major – vysoká priorita, problém je nutné řešit prioritně. Většinou se používá tam, kde chceme rozlišit které úkoly je potřeba odbavit dříve.

C. Normal – normální priorita, problém není nadřazen ostatním jako je tomu u Major priority.
D. Minor – nízká priorita, trivialita nemající zásadních vlivů na chod aplikace (například změna barvy tlačítka).

- ToDo – označení pro zadání na kterém se ještě nezačalo pracovat a nebo bylo vrácen k přepracování.
- Invalid – s požadavkem není něco v pořádku, například není uznaná reklamace.
- Duplicate – označení pro zadání, které už je jednou zadáno.
- Waiting – zadání je potřeba o něco doplnit a čeká se na jeho doplnění.
- InProgress – probíhá zpracování zadání.
- OnDev – řešení zadání se nachází na developerském prostředí.
- TestMe – zadání je zpracováno a připraveno k otestování zákazníkem.
- OnUAT – řešení zadání je nahráno na testovacím prostředí.
- Tested – řešení zadání je otestováno zákazníkem.
- PushIt – schválení zákazníkem, že řešení může být nahráno na produkční server.
- OnPro – řešení zadání se nachází na produkci.
- ReadyForClosure – požadavek je možno uzavřít jako vyřešený.

Práci s GitHubem z programátorského pohledu jsem prováděl skrze aplikaci Sourcetree. Každý nový projekt v GitHubu má jako první a hlavní větev aplikace nastavený master. Z důvodů větší přehlednosti jsem si vytvořil tři větve DEV, UAT, PRO. Všechny tři větve musí a odpovídají přesně tomu co je na aplikačních serverech. Musí tomu tak být, protože může docházet ke zpoždění ve schvalování zadání a nikdo si po delší době nebude pamatovat jaký změny prováděl v které větvi. Další zásada se týká vytváření nových kódových větví. Název každé větve začíná iss poté následuje číslo větve, které odpovídá úkolu v GitHubu. Nové větve se zásadně vytváří z větve PRO. Je to z důvodu slučování větví, kdy se tímto minimalizuje riziko konfliktů. Při provádění komitů je dobré na začátek textu uvést #čísloiss. Toto číslo se poté ukáže v GitHubu s příslušným textem a náhledem změn.



Obrázek 6 - rozhraní Sourcetree

4.1.3 Přípravy před začátkem programováním

Než přejdu na programování a řešení problémů, které jsem musel v projektu řešit, chtěl bych zde ještě věnovat odstavec různým nastavením. Za velmi důležité, před začátkem programování, považuji rozhodnutí o technologiích a knihovnách, které budu potřebovat.

4.1.3.1 Příprava lokálního stroje

Programovací studio jsem si zvolil Visual Studio Code. Studio jako takové je již po stažení připravené na programování, ale je dobré v něm provést pár nastavení. Při programování v týmu (ale i obecně) je dobré dodržovat programovací standard. Tento standard buď určuje samotný tvůrce jazyka a nebo si ho předepisují sami firmy. Studio se dá nastavit tak, aby samo formátovalo kód po uložení. Dodržování standardu nám zajistí konzistentní formu kódu, lepší čitelnost a lepší orientaci. Když pominu fakt, že tato nastavení jsou více méně dobrovolná, tak nutností je sjednocení oddělovačů, tipicky dvě mezery nebo tabelátor, jejich množství a jejich automatické odmazávání na koncích řádků. Toto nastavení nás ušetří zbytečných konfliktů při sjednocování větví v gitu. Moje poslední doporučení se týká instalovaných doplňku do studia. Zde doporučím PHP Intelephence – našeptávání a kontrola syntaxe, HTML Snippets –

našeptávání a automatické párování HTML a Bracket Pair Colorizer – barevné párování závorek.
Jako programovací jazyk byl zvolen PHP7. Kvůli co největší minimalizaci potencionálních chyb je dobré mít nastavený lokální stroj tak, aby se co nejvíce podobal serveru. Není tedy nic jednoduššího než si na serveru vytvořit jednoduchý skript a z něj zjistit vše potřebný.

```
<?php
  phpinfo();
?>
```

Na stránce (Apachefriends, 2020) poté stačí stáhnout aplikaci XAMPP s verzí PHP jako je na serveru. Lokální stroj je potřeba ještě nastavit, aby komunikoval s databází Oracle. K tomu je zapotřebí stáhnout Oracle instant client a postupovat s instalací podle instrukcí ve spodu stránky.

Následující krok je v XAMPP panelu pod tlačítkem config otevřít php.ini a v něm odstranit středník u extension=php_oci8_12c.dll. Tím bychom měli mít nainstalované a připravené PHP na lokálním stroji. Správné nastavení můžeme ověřit například dotazem.

```
SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') FROM DUAL;
```

4.1.4 Aplikační jádro

Před programováním jakékoliv aplikace je dobré se zamyslet k čemu, komu a jak dlouho taková aplikace bude sloužit. U jednoduché aplikace (například obyčejný prezentační web), je zbytečné stavět robustní jádro. Taková aplikace se velmi pravděpodobně v budoucnosti nebude nijak měnit ani rozšiřovat. Návratnost stavby složitého jádra u takové aplikace je tedy nulová a tímto přístupem, který se dá nazvat kanón na vrabce, akorát ztratíme čas. Naopak aplikace jako jsou datové sklady a informační systémy vyžadují robustní jádro, díky kterému pozdější úpravy a nové moduly budou mnohem snazší na programování. Dobré aplikační jádro tedy usnadňuje implementaci a údržbu/servis aplikace.

4.1.4.1 **Správa knihoven**

Nyní když jsme si v předchozí kapitole připravili nástroje, je ideální čas stáhnout knihovny. Ke stažení a správě knihoven využiji Composer. Composer je zapotřebí stáhnout a nainstalovat, instalátor nám automaticky nastaví ve prostředí Windows proměnnou PATH tak, abychom mohli používat Composer přímo z příkazové řádky. Na stránce Composeru je poté odkaz na všechny balíčky, které se přes Composer dají spravovat. Když si najdeme knihovnu co potřebujeme, je zapotřebí ještě zkontrolovat závislosti (nejdůležitější je verze PHP, zbytek se dá většinou doinstalovat nebo se stáhne sám). Pakliže nesplňujeme verzi PHP, lze ve verzích jednoduše listovat a najít si tu pro, která nám bude fungovat. V projektové složce poté musíme vytvořit soubor composer.json. Do něj pak stačí napsat názvy balíčků z Composeru s příslušnou verzí. Náš soubor může vypadat například takto:

```
{
    "require": {
        "dibi/dibi": "^4.0",
        "twig/twig": "^2.5",
        "components/jquery": "^3.3",
        "phpmailer/phpmailer": "^6.0"
    }
}
```

Visual Studio Code má v sobě podporu pro terminál. Můžeme tedy přímo ve studiu do terminálu napsat composer install. Vše se automaticky stáhne a vytvoří se složka vendor. Je rozumné tuto složku přidat do .gitignore, zmenšíme tím zabraný prostor na GitHubu.

4.1.4.2 **Adresářová struktura projektu**

Pro lepší pochopení následujících kapitol, rozeberu v této podkapitole adresářovou strukturu projektu. Aplikace nyní ve svém jádře už obsahuje složku vendor, ostatní složky a skripty jsou založeny mnou. Adresářová struktura projektu je následující:

- cron_scripts – PHP skripty, který spouští serverový plánovač úloh.
- css – kaskádové styly. Pro každou stránku je zde podložka s názvem odvozeným od skriptu, který stránku spouští. V kořenu složky je pouze general.css který obsahuje styly společný pro celý projekt.

- engine – třídy, kontroléry a routery. Ve zkratce se zde nachází back-endová funkcionalita projektu.
- img – obrázky projektu.
- js – JavaSriptové soubory, platí zde stejná pravidla jako pro css.
- templates – Twig soubory s pravidly nastavenými stejně jako u css

V kořenu adresáře se nachází ještě nezbytný skripty, tvořící (většinou) jednotlivé odkazy v menu aplikace.

4.1.4.3 **Postup načítání skriptů**

Abych nemusel ručně skripty do sebe vkládat, tak jak se to v PHP klasicky dělá, využiji autoload. Tento přístup má velkou výhodu v tom, že kód je mnohem čistější a bližší klasickému OOP. Každý skript, který v sobě neimplementuje třídu, na svém začátku načítá skript _includes.php.

```
require_once('_includes.php');
```

Tento skript v sobě obsahuje pomocí konstant za definované connection stringy do databází, ftp uložiště a verzi aplikace. Protože je nutné rozlišovat aplikační prostředí skript v sobě načítá další skript a to _servername.php. Tento skript je čistě pro účely ladění na lokálním stroji.

```
<?php
define('SERVER', 'DEV');
?>
```

Ze skriptu _includes.php se již spouští init.php z engine složky. Tento skript kontroluje, jestli spuštění požadované akce vyžaduje přihlášení a jestli je uživatel přihlášen. V projektu existují skripty, který se spouštějí automatem, v takovém případě je jako autor akce Systém s id=1. Dále se ze skriptu spouští autoloader a to jak projektový tak z vendoru (generovaný Composerem). Následuje ukázka kódu, který ověřuje přihlášení.

```
if ($REQUIRE_LOGIN) {
  if (!isset($_SESSION['loged_user'])) {
    header('Location: ' . SYSTEM_HOST . '/login.php');
  }
} else {
  define('USER_ID', 1);
```

```
  define('USER_NAME', 'System');
}
```

Důležitou součástí init.php je spouštění autoloaderů.

```
require_once(INCLUDE_PATH . 'engine/autoloader.class.php');
require_once(INCLUDE_PATH . '/vendor/autoload.php');
```

Nyní je na místě vysvětlit si autoloader. Jak jsem již psal systém obsahuje dva. Tím generovaným se nemá smysl zabývat a ani ho nedoporučuju měnit, Composer ho totiž při aktualizacích pře generovává. Autoloader automaticky načítá soubory tříd a rozhraní při jejich použití. Nutnou podmínkou je pojmenování podle zavedené konvence tedy třída.class.php, případně interface.interface.php. Nutné je taky mít správně jmenné prostředí v třídách a při použití ve skriptech mít správně použití. Autoloader v sobě implementuje funkci spl_autoload_register() pomocí které lze za definovat funkci pro automatické nahrávání tříd. Celé použití vypadá takto.

```
private static function autoload($class){
    $class = str_replace('\\', '/', $class);
    $required_file = INCLUDE_PATH . 'engine/class/' . $class . '.class.
php'; //klasika chceme class
    if(!file_exists($required_file)){ // možná jsme chtěli interface
      $interface = INCLUDE_PATH . 'engine/class/' . $class . '.interfac
e.php';
      if(file_exists($interface))
        require_once($interface);
    }else{
      require_once($required_file);
    }
  }

  public static function register(){
    try{
      spl_autoload_register(array(__class__, 'autoload'), true);
    }catch (Exception $e){
      var_dump($e);
    }
  }
```

4.1.4.4 **Databázový konektor**

Hybridní model využívá celkem pět konekcí do dotabází. Dibi má dva přístupy jak přistupovat do databáze. První přístup je takzvaně statický, kde dibi si pamatuje připojenou relaci. Programátor se pak může kdekoliv v projektu dotazovat následujícím způsobem.

```
$db_time = dibi::fetchSingle('SELECT TO_CHAR(SYSDATE, "MM-DD-
YYYY HH24:MI:SS") FROM DUAL');
```

Druhý přístup je připojení do databáze, pomocí objektu Dibi connection. Tento typ připojení samozřejmě musíme uložit do proměnný, Dibi ho nepodrží, tak jako u statického způsobu. Těchto připojení můžeme vytvořit neomezeně mnoho na rozdíl od statického způsobu, který se vždy přepíše, respektive přepojí. Dotaz do databáze pomocí objektu:

```
$db_time = $db->fetchSingle('SELECT TO_CHAR(SYSDATE, "MM-DD-
YYYY HH24:MI:SS") FROM DUAL');
```

Výhodné řešení pro obě situace je vytvořit třídu, která v sobě obsahuje funkce, pomocí kterých se můžeme mezi databázemi přepínat. Funkce mají jeden parametr, který je nepovinný a slouží pro rozlišení mezi statickým a dynamickým připojením. Funkce jsou statický, protože zde instancování nedává smysl. Následuje ukázka kódu funkce, ostatní jsou stejné akorát mají jiné konstanty samozřejmě.

```
public static function hm_l1_connect($static = true) : Connection{
    $connection = null;
    $con_arr = array(
      'driver' => DB_HOST_HM,
      'username' => DB_USERNAME_HM_l1,
      'schema' => DB_SCHEMA_HM_l1,
      'password' => DB_PASS_HM_l1,
      'database' => DATABASE_HM,
      'charset' => CHARSET,
    );

    try{
      if($static){
        $connection = dibi::connect($con_arr, DB_SCHEMA_HM_l1);
```

```
      }else{
        $connection = new Dibi\Connection($con_arr, DB_SCHEMA_HM_l1);
      }
    }catch(Exception $ex){
      echo ('Připojení do l1 se nezdařilo');
      exit;
    }
    return $connection;
  }
```

4.1.4.5 **MVC**

V teoretické části práce jsem psal o modelu MVC, který ve své podstatě tvoří velmi důležitou část jádra aplikace. Z adresářové struktury popsané výše pak vyplývá:

- Model a Kontrolér se nachází ve složce engine
- View patří css, img, js, templates

Proces vykreslení obsahu stránky funguje tak, že v kořenovém adresáři je skript, který slouží jako uvozovací skript jednotlivého modulu stránky. Všechny funkcionality týkající se tohoto modulu se odkazují na něj. Skript v sobě obsahuje routing, tedy obsloužení všech událostí vyvolaných po načtení stránky. Celý proces zde popíši rozdělený do jednotlivých kapitol.

4.1.4.5.1 Kontrolér

Kontrolér je ve své podstatě routing, o kterém jsem psal v předchozím odstavci. Routing zastává třída, která se jmenuje action_router, třída se napříč moduly dědí. Pro action_router platí jedno zásadní pravidlo, třída nesmí obsahovat jakoukoliv implementaci. Implementace se dělá v modelu, v kontroléru tedy mohou být maximálně ošetření proti PHP notice jako je například tahání hodnoty z pole z indexu který neexistuje.

Třída v sobě obsahuje pole actions s názvy akcí a funkce které mají stejné názvy jako ty akce v poli. Doporučuji pole akcí udělat private a to z důvodu dědění. Potomek třídy ho totiž (za podmínky stejného názvu) přetíží a akce, které bychom chtěli mít v základním action_routeru přestanou fungovat. Přetížení pole akcí poté můžeme udělat ručně v konstruktoru.

```php
  private $actions = array(
    'logout',
  );

  public function __construct($actions = array()){
    $this->extend_actions($actions);
  }

  protected function extend_actions($actions){
    $this->actions = array_merge($this->actions, $actions);
  }
```

Metody třídy dělající routing by neměly být zvenčí viditelné (private protected), routing by se měl spouštět pouze pomocí atributu. V potomkovi z logiky mohou být metody pouze protected. Kontrolér by měl obsahovat jednu metodu, která se provede když uživatel vyvolá akci, která není definována. Všechny akce se volají pomocí call_user_func_array(), PHP funkce, která zavolá funkci/metodu s parametry zadanými uživatelem. Implementace volání metod a jedné metody v návaznosti na předchozí příklad.

```php
  public function take_action($act, $params){
    if(in_array($act, $this->actions)){
      call_user_func_array(array($this, $act), $params);
    }else{
      $this->default_action($act, $params);
    }
  }

  private function logout($params){
    session_destroy();
    header("Refresh:0");
  }
```

Názorný příklad použití.

```php
if(isset($_POST['act'])){
  $action_router = new action_router();
  $action_router->take_action($_POST['act'], array($_POST));
}elseif(isset($_GET['act'])){
  $action_router = new action_router();
```

```
  $action_router->take_action($_GET['act'], array($_GET));
```

4.1.4.5.2 View

View se stará o zobrazení požadovaného obsahu uživateli. Jako šablonovací technologie byl zvolen Twig, který umožňuje efektivně oddělit back-end od front-endu. Pro maximální zefektivnění je nutné vytvořit třídu, která bude sloužit jako obsluha Twigu. Třída v sobě má pomocí konstant za definovaný cesty k společným css a js. Všechny skripty a styly se ukládají do polí, které se ve Twigové šabloně pomocí cyklů vkládají do hlavičky. Tyto pole jsou atributy třídy, ke kterým se přistupuje pomocí metod get nebo set;

```
private $twig_loader;
private $twig;
private $javascript;
private $css;

public function __construct(){
$this->
setTwig_loader(new Twig_Loader_Filesystem(INCLUDE_PATH . 'templates'));
    $this->setTwig(new Twig_Environment($this->getTwig_loader()));
    $this->javascript = array(); //pole javascriptů
    $this->css = array(); //pole stylů
}

public const GENERAL_CSS = 'css' . DIRECTORY_SEPARATOR . 'general.css';
public const GENERAL_JS = 'js' . DIRECTORY_SEPARATOR . 'general.js';
```

Přidání dalších technologií například jQuery, které nejsou vždy nezbytně nutné je pak zajišťováno pomocí metod.

```
public function use_jquery(){
  $this->
pushJavascript('vendor' . DIRECTORY_SEPARATOR . 'components' . DIRECTOR
Y_SEPARATOR . 'jquery' . DIRECTORY_SEPARATOR . 'jquery.js');
}
```

O vykreslení šablon se stará metoda render, prvním parametrem je úplná cesta k šabloně a druhým parametrem jsou proměnné šablony. Twig šablona má svojí vlastní syntaxi, která vychází z PHP. Twig umožňuje práci ve dvou režimech, a to s cache a bez cache. Cachování může být výhodné zvláště když načítáme velkou část obsahu neměnnou. V takovém případě může mít cachování positivní vliv na výkon. Typicky cachovanoné jsou css styly a JavaScripty. To může být nevýhodné, obzvláště pokud děláme časté změny v zmiňovaných souborech. Zde nám vypnutí cache nepomůže, protože moderní prohlížeče css a js soubory cachují také. Řešení je tedy css a js verzovat. Předvedu zde ukázku kódu renderovací funkce a hlavičku html z twig šablony.

```
 public function render(string $template, array $fields){
   $renderer = $this->getTwig();
   $fields['javascript'] = $this->getJavascript();
   $fields['css'] = $this->getCss();
   $fields['app_version'] = APP_VERSION;
   return $renderer->render($template, $fields);
  }
```

```
<head>
    <meta charset="UTF-8">
    <title>
        {% block title %} {% endblock %}
    </title>

    {% for style in css %}
    <link rel="stylesheet" type="text/css" href="{{ style }}?v={{ app_v
ersion }}" />
    {% endfor %}
    {% for js in javascript %}
    <script type="text/javascript" src="{{ js }}?v={{ app_version }}"><
/script>
    {% endfor %}
</head>
```

4.1.4.5.3 Model

Úkolem modelu je především práce s daty. Typická metoda modelu obsahuje SQL dotaz jehož výsledek zpracuje, zformátuje výstup a vrátí tento výstup kontroléru. Při programování modelu bychom se měli snažit připravovat výstup tak, aby ho kontrolér

mohl rovnou předat view bez modifikací. Výhodné je pro každý model si individuálně připravit datový kontejnery. V těchto kontejnerech můžeme implementovat metody pro datové manipulace jako je například výstup v CSV. Jelikož Twig podporuje čtení dat přímo z objektů nabízí se jako kontejner třídy. Má to minimálně jednu výhodu v tom, že je to přehlednější než asociativní pole, které bychom do Twigu posílali, kdybychom neměli data v objektech. Existují i situace, kdy je kontejner zbytečný. Jestliže stačí načíst data z databáze a poslat je na výstup vystačíme si s šikovně vytvořeným SQL. Tím mám na mysli SQL dotaz s aliasy. Dibi totiž jako výsledek metody fetchAll() vrací pole objektů, což je přesně to co bychom vytvářeli pomocí kontejneru.

Když kontejner přece jenom potřebujeme, uděláme to tak, že vytvoříme jednoduchou třídu s atributy představující proměnné v Twig šabloně. Lze na to jít dvěma způsoby:

1. Třída, která má všechny atributy public
2. Třída, která má všechny atributy private, ale má k nim příslušné metody get(), set(). Zní to jako hodně práce navíc oproti prvnímu způsobu, ale není, metody get() a set() můžeme nechat vygenerovat studiem.

Třída může dále obsahovat metody jako to_string(), to_csv_row() a podobně. Kontejner může představovat i databázovou tabulku a v takovém případě může implementovat funkce ukládání do databáze.

Kromě kontejneru pod model spadá ještě třída, kterou jsem nazval json_result. Jedná se o pomocnou třídu, jejíž účelem je odeslat Javascriptu informaci, že nějaký proces skončil správně nebo s chybou. Její využití tak nalezneme především při AJAXových voláních. Třída v sobě obsahuje předpřipravené metody vracející objekt, který se předá funkci json_encode a echem se pošle obvykle jako výsledek nějakého AJAXu. Kód třídy vypadá takto.

```
class json_result{
  public $message;
  public $success;
  public $code;

  public static function get_exception_json_result($message, $code = 0)
:self{
    $result = new self();
```

```
    $result->code = $code;
    $result->message ='Chyba: ' . $message;
    $result->success = 0;
    return $result;
  }

  public static function get_all_ok_json_result():self{
    $result = new self();
    $result->code = 1;
    $result->message = 'vše ok';
    $result->success = 1;
    return $result;
  }
}
```

4.2 ETL proces

Cílem Hybridního modelu je výpočet spotřeb elektrické energie. Aby bylo co počítat je potřeba do systému každý měsíc nahrávat nová data pomocí takzvaných dávek. Data z primárních zdrojů je nejdříve potřeba naimportovat, poté validacemi očistit, a nakonec dát databázi pokyn, že data jsou připravena a že může zahájit výpočet.

Importování dat probíhá buď plným automatem, který se spouští každý měsíc v pravidelném intervalu nebo ručně tlačítkem z obrazovky výpočty. ETL před tím, než cokoliv spustí nejdříve ověřuje, zdali jsou data v primárních systémech dostupná. Když je splněná tato podmínka je do databáze zaveden proces výpočtu. Tento proces čeká na aplikaci až naimportuje a validuje data. Pakliže při importu nebo validaci nějaký úkol skončí s chybou je proces výpočtu zapsáním příznaku ukončen a celý import se považuje za chybný. Proces výpočtu má navíc ještě nastavený time-out, když aplikace nestihne všechny svoje úkoly provést včas, proces výpočtu se ukončí. Průběh celého procesu lze z aplikace sledovat pomocí monitorovací konzole.

Hybridní model má tři primární zdroje. Dva jsou z CSV souborů a třetí je Oracle SQL databáze. CSV zdroje se od sebe liší přístupem k souborům. Mezitím co jeden se nachází přímo na disku serveru, druhý se musí stahovat pomocí FTP z uložiště.

4.2.1 Pomocné skripty

Předtím než začnu popisovat jednotlivé kroky ETL u všech datových zdrojů chtěl bych zde popsat třídy nebo funkční moduly co díky ETL vznikly a našly uplatnění dále v aplikaci.

4.2.1.1 E-mailing

ETL proces musí fungovat vždy na 100%. Když by vlivem čehokoliv spadl jakýkoliv skript nelze považovat ETL proces za dokončený. Nekompletní data jsou uživateli k ničemu. Proto každý skript v ETL procesu v sobě obsahuje logiku, která při detekci chyby odešle uživateli e-mail s podrobností. E-mail také slouží jako potvrzení pro uživatele, že provedli akci korektně, e-mail tak slouží jako důkaz.

Pro e-mailing byla využita knihovna PHPMailer. Tato knihovna v sobě obsahuje již připravené funkce, které velmi usnadňují práci s e-mailem. Jako formát e-mailu jsem zvolil HTML. Veškeré e-mailové šablony jsou napsané v Twigu. Konstruktor PHPMaileru přijímá jako parametr bool hodnotu, která přepíná, jestli chceme vypisovat chyby nebo ne. Lze toho využít při ladění a posílat si e-maily na localhost. Potřebujeme k tomu však další aplikaci. Já použil Papercut. Ukázka kódu demonstruje inicializaci PHPMaileru s přepínačem na localhost.

```
if($localhost){
  $mail = new PHPMailer(true);
  $mail->SMTPDebug = 2;
}
else{
  $mail = new PHPMailer(false);
  $mail-> Host = 'server.mailer.cz';
}
$mail->CharSet = $mail::CHARSET_UTF8;
$mail->Encoding = $mail::ENCODING_BASE64;
$mail->ContentType = $mail::CONTENT_TYPE_TEXT_HTML;
$mail->isSMTP();
$mail->Port = 25;
```

4.2.1.2 Loger

V celém procesu ETL je spousta operací, o kterých je dobré mít zpětnou vazbu. Skripty v ETL bývají pouštěny na pozadí v režimu přesměrování výstupu na null. Zpětně tak

nelze zjistit proč skript selhal, respektive k jaké chybě došlo. I kdybychom přesměrovali výstup na textový soubor, tak by za chvíli nakynul tak, že by se v něm nikdo nevyznal. Z tohoto vyplývá potřeba logeru. Log klidně můžeme provádět po každém úspěšném řádku. Dobré je logovat konec skriptu. Můžeme si tak být 100 % jistí, že se skript ukončil a není zaseklý. Výhodné je u logů nepoužívat koncovku .txt ale .log. Při prohlížení například ve Visual Code se nám text v logu hezky čitelně podbarví.

Požadavky na log se mohou systém od systému lišit. V každém případě u logované akce by měl být čas abychom měli přehled kdy se událost stala. Následuje ukázka jednoduchého logeru.

```
class data_processing_log{
  private $resource;

  public function __construct(string $filename){
    $this->resource = fopen($filename, 'a');
  }

  public function log_to_file(string $line){
    $date = (new \DateTime())->format('Y-m-d H:i:s');
    $line = $date . ": " . $line . PHP_EOL;
    fwrite($this->resource, $line);
  }
}
```

4.2.1.3 Databázový helper

Práce s Dibi umožňuje si velmi ulehčit život při psaní sql. Klasickým případem jsou příkazy insert a update. V klasickém sql člověk velmi snadno udělá chybu typu chybějící čárka. Z těchto důvodů jsem vytvořil třídu, kterou jsem nazval db_model. K čemu je to dobré vysvětlím na praktickém příkladě.

Představme si, že máme následující tabulku:

HMSYS.CRON_JOBS			
P	*	CRONJOB_KEY	NUMBER (10)
F	*	CRONJOBSTATE_KEY	NUMBER (10)
F	*	CRONJOBTYPE_KEY	NUMBER (10)
		PARAMETER	VARCHAR2 (50 BYTE)
		PARENT_PID	NUMBER
	*	I_DATE	DATE
	*	I_AUTHOR	NUMBER (6)
		U_DATE	DATE
		U_AUTHOR	NUMBER (6)
CRON_JOBS_PK (CRONJOB_KEY)			
CRON_JOBS_CRON_JS_FK (CRONJOBSTATE_KEY)			
CRON_JOBS_CRON_JT_FK (CRONJOBTYPE_KEY)			
CRON_JOBS_PK (CRONJOB_KEY)			

Obrázek 7 - tabulka cron_jobs

Pro práci s touto tabulkou vytvoříme následující třídu:

```
class cron_job_db_model{
  public $CRONJOB_KEY;
  public $CRONJOBSTATE_KEY;
  public $CRONJOBTYPE_KEY;
  public $PARAMETER;
  public $PARENT_PID;
  public $I_DATE;
  public $I_AUTHOR;
  public $U_DATE;
  public $U_AUTHOR;
  public $modifikators = ['%sql', '%i', '%i', '%s', '%i', '%sql', '%i',
 '%sql', '%i'];

  public function __construct(){
  }

  public function get_db_columns(){
    return db_model::get_db_columns($this);
  }

  public function get_db_columns_with_modifikators($snaps_values)  {
    return db_model::get_db_columns_with_modifikators($snaps_values, $t
his);
  }
```

```
  public function set_db_columns($snaps_values){
    db_model::set_db_columns($snaps_values, $this);
  }

}
```

Pracnost vytvoření takovéto třídy není velká, jak se na první pohled může zdát. Atributy se dají vykopírovat z Oracle developera, hromadnou řádkovou editací před ně připsat public $ a zakončit středníkem. Největší pracnost spočívá v psaní modifikátorů. Db model implementuje 3 důležité funkce, které popíšu, vysvětlím, jak fungují a předvedu ukázku kódu.

1. get_db_columns – vrátí jako pole které má jako hodnoty názvy sloupců tabulky. Princip je jednoduchý, pomocí PHP funkce převede atributy objektu na pole hodnot. Poslední hodnotu zahodíme, to jsou vždy modifikátory.
   ```
   public static function get_db_columns($object){
     $vars_array = array_keys(get_object_vars($object));
     array_pop($vars_array);
     return $vars_array;
   }
   ```
2. set_db_columns – nastaví atributy třídy na hodnoty předané buď objektem nebo polem. Atributy předaného objektu nebo pole by měly odpovídat pořadím atributům třídy, pro kterou model použiji. Princip je opět jednoduchý iteruje se přes hodnoty a těmi se plní postupně atributy třídy.
   ```
   public static function set_db_columns($values, $object){
     if(is_object($values)){
       foreach($values as $atribut => $value){
         $object->$atribut = $value;
       }
     }else{
       $this_columns = get_object_vars($object);
       $i = 0;
       $count = count($this_columns) - 1;
       $keys = array_keys($values);
       foreach($this_columns as $atribut => $value){
         if ($i >= $count) continue;
         $object->$atribut = $values[$keys[$i]];
         $i++;
       }
     }
   }
   ```

3. get_db_columns_with_modifikators – vytvoří pole kde klíčové hodnoty odpovídají sloupcům v tabulce a přidá k nim modifikátor. Modifikátor, respektive Dibi pak zaručí to, že do zadaného sloupce nikdo nevloží hodnotu co tam nepatří. Princip spočívá ve vytvoření pole, kde se skombinuje název sloupce s modifikátorem a přiřadí se hodnota z poskytnutého pole nebo objektu. Návratová hodnota je pole.

```
public static function get_db_columns_with_modifikators($values,
$object){
  if(is_object($values)){
    $this_columns = get_object_vars($values);
    $i = 0;
    $count = count($this_columns) - 1;
    $db_cols_with_dibi_modifikators = array();
    foreach($this_columns as $atribut => $value){
      if($i >= $count)continue;
      $db_cols_with_dibi_modifikators[$atribut . $object->
modifikators[$i]] = $value;
      $i++;
    }
    return $db_cols_with_dibi_modifikators;
  }else{
    $this_columns = get_object_vars($object);
    $i = 0;
    $count = count($this_columns) - 1;
    $db_cols_with_dibi_modifikators = array();
    foreach($this_columns as $atribut => $value){
      if($i >= $count) continue;
      $db_cols_with_dibi_modifikators[$atribut . $object->
modifikators[$i]] = $values[$i];
      $i++;
    }
    return $db_cols_with_dibi_modifikators;
  }
}
```

4.2.2 Sumy spotřeb

Jedná se o CSV zdroj a taky nejjednodušší zdroj dat z pohledu ETL procesu.

4.2.2.1 **Extrakce**

Extrakce toho CSV probíhá pomocí Cron skriptu, který pravidelně čte vzdálenou složku (složku na jiném stroji kam je síťový přístup). Soubor se pokaždé přepíše, jeho verzování se neřeší.

4.2.2.2 **Transformace**

Soubor téměř žádnou transformaci nepotřebuje. Před jeho načtením se odmazává hlavička a kontrolují se prázdné řádky.

4.2.2.3 **Load**

Do databáze se nahrává pouze ten řádek, jehož perioda (rok-měsíc) se shoduje s požadavkem. Předtím než je řádek vložen, proces zneplatní předchozí záznam, pokud existuje.

4.2.3 Snímky z měřáků

Snímky jsou provozní název pro CSV soubory obsahující záznamy z elektroměrů montovaných na EHV/EJ. Jak už jsem psal výše soubory jsou stahované z FTP, kam jsou každý měsíc nahrávané. Z hlediska extrakce nás ještě zajímá, jak rozpoznat soubory, které já potřebuji od ostatních. Soubory mají jmennou konvenci, kdy název souboru obsahuje 12timístné číslo identifikátoru elektroměru UIC, datum začátku období, datum konce období. Název souboru může tedy vypadat následovně 123456789451_20191201_20191231.csv.

4.2.3.1 **FTP konektor**

PHP má v sobě podporu pro práci se vzdáleným uložištěm. Jedná se o sadu funkcí, které jsou v manuálu popsány jako FTP functions. (PHP.net, 2020) Výhodné je tyto funkce implementovat do vlastní třídy a vytvořit si tak svůj FTP konektor.

Já jsem se rozhodl připojení a login na FTP rozdělit do dvou metod. Připojení se děje v konstruktoru a login pak ve metodě connect. Nechybí ani destruktor, který připojení ruší.

```
  private $connection;
  private $login;

  public function __construct(string $server, int $port){
    $this->setConnection(ftp_connect($server, $port));
  }

  public function connect(string $user, string $password):bool{
    if($this->getConnection()){
      $this->setLogin(ftp_login($this->
getConnection(), $user, $password));
      ftp_pasv($this->getConnection(), true);
      return true;
    }else{
      return false;
    }
  }
   public function __destruct(){
    if($this->connection) ftp_close($this->connection);
  }
```

Další důležité funkce, co budeme po FTP konektoru potřebovat jsou seznam souborů ve vzdálené složce a získání souboru. Pro účely logů se určitě ještě budou hodit na časové razítko souboru a velikost souboru. Postupně jsou to:

```
  public function get_files_paths_array_list(string $dir_name){
    if($this->getConnection()){
      $list = ftp_nlist($this->getConnection(), $dir_name);
      $return_list = array();
      foreach($list as $file){
        if(ftp_size($this->
getConnection(), $dir_name . '/' . $file) > 0)
          $return_list[] = $dir_name . '/' . $file;
      }
      return $return_list;
    }
  }

  public function get_file(string $local_path, string $remote_path){
    if($this->getConnection()){
```

```
      ftp_get($this->
getConnection(), $local_path, $remote_path, FTP_BINARY);
      return true;
    }else{
      return false;
    }
  }

  public function get_file_stamp($file_path):int{
    return ftp_mdtm($this->getConnection(),$file_path);
  }

  public function get_file_size($file_path):float{
    return ftp_size($this->getConnection(), $file_path);
  }
```

4.2.3.2 Webová služba

V plánech do budoucna je od CSV souborů odstoupit a provádět import snímků pomocí webové služby. Odpadne tím potřeba soubory stahovat na server a extrakce se tak zúží na načtení dat z webové služby do paměti a poté import do databáze. Další výhodou je prohlížení obsahu dat při hledání chyb. Ve webové službě lze pomocí softwaru, například SOAP UI, provést dotaz na konkrétní data. Není nutno vyhledávat ručně mezi soubory na FTP, jedná se tedy o pohodlnější způsob ladění, který nějakou vteřinu ušetří. Poslední výhodou webové služby je, že podle WSDL lze klienta přijímajícího data kompletně vygenerovat.

4.2.3.3 Extrakce

Proces extrakce u snímků spočívá v ověření dostupnosti a stažení z FTP uložiště. Po několika iteracích ladění jsem dospěl k verzi, která spočívá v tom, že se vytvoří nová složka ve složce csv_files. Název složky je datum a čas založení úkolu importu dat. Příklad názvu složky 2020-01-25_19-01-26. Do složky jsou naimportovány všechny soubory s časem, který odpovídá požadované periodě. Ve složce dále vzniknou dva textový soubory obsahující logy. První log obsahuje informace o souborech na FTP, druhý log obsahuje informace o souborech stažených. Jednoduchým porovnáním lze zjistit zdali nedošlo při stahování k chybě.

4.2.3.4 **Transformace**

Transformace probíhá až po loadu, ale trošku předběhnu a budu se držet pořadí podle zkratky. Transformací u snímků mám na mysli vyhledávání typizovaných chyb v datech a pokus o jejich korekci. Každá chyba má svůj číselný identifikátor v tabulce možných chyb. Jeden kód slouží i pro označení dat, pro které korekci nelze udělat a je třeba je vyřadit.

Typy chyb, na které můžeme narazit a jich případné metody korekce vypíšu v následujícím výčtu.

- Není uvedená trakce – v hybridním modelu se rozlišují následující trakce AC16, AC50, AC60, DC. Vše ostatní je považováno za chybu.
    - V první řadě je soubor prověřen, že obsahuje alespoň jeden záznam o trakci. Pakliže ne, všechny záznamy jsou označeny za neplatný a proces je ukončen.
    - Pakliže chybí záznam o trakci může být doplněn dle přecházejícího, respektive následujícího záznamu.
- Není uvedená oblast spotřeby – principiálně jde o to, rozlišit jeslí jel vlak na území ČR nebo EU.
    - Z korekcí jsou nejdříve vyloučeny UIC patřící zahraničním dopravcům.
    - Poté se ověřuje stejně jako u trakce existence alespoň jednoho záznamu, pravidla platí stejná jako u trakce.
    - Chybějící záznam o oblasti spotřeby může být doplněn dle předchozího, ale za podmínek že časový rozdíl mezi přechozím snímkem není větší než 5 minut. Dále lze takto po sobě provádět korekci pouze u 15 výpadků po sobě, následující snímky jsou neplatný a jsou vyřazený z výpočtu.
- Hodnoty spotřeb a rekuperací musí být rostoucí – hodnoty na měřáku spotřeb pouze rostou, jakákoliv změna směrem dolů znamená vadný elektroměr. Korekce na tento problém není.
- Hodnota na měřákách spotřeby a rekuperace nesmí obsahovat skok – měřáky jsou v MWh skoková změna větší než 5 znamená pravděpodobnou výměnu elektroměru.
    - Všechny následující hodnoty musí být přepočítány o rozdíl skoku.

- Hodnota činný ho odběru překročila povolené rozpětí – hodnota činného odběru by se měla pohybovat v intervalu <0; 0,1> v rámci dvou sobě jdoucích snímků. Na tuto chybu není korekce.

4.2.3.5 **Load**

Celý load se dá shrnout do několik kroků.

1. Zneplatnění snímků z předchozí dávky. Data by byla duplikovaná a logicky by vyšly dvojnásobné spotřeby
2. Prověření dat, že sloupce obsahují to, co mají, aby tam kde mají být čísla byla čísla.
3. Založení db modelů.
4. Vytvoření partitions.
5. Insert. Zde bych rád rozvedl jeden svůj poznatek. K vkládání CSV do databáze se nabízí bulk-insert. Úskalím bulk-insertu v Oracle je, že do něj nelze vkládat sekvenci. Nabízí se tedy možnost nechat si na generovat klíče sekvencí a poté vložit do bulk-insertu. Ovšem výsledná doba trvání byla k mému zklamání stejná jako když použiji sekvence v klasickém insertu. Z toho usuzuji, že sekvence jsou úzkým hrdlem loadu. Existuje však ještě jeden faktor výrazně ovlivňující rychlost. Tím faktorem jsou transakce. Dibi má v základním nastavení zapnutý autocommit, po každém insert, update nebo delete udělá sama commit. Toto se naštěstí řešit dá, například takto.

```
try {
  dibi::begin();
  foreach($insert_arrays as $insert_array){
    dibi::query('INSERT INTO %n %v', tables_keys::snaps, $insert
_array);
  }
  dibi::commit();
}catch (\Exception $ex){
  dibi::rollback();
}
```

4.2.4 Hrany

Posledním datovým zdrojem, Oracle SQL databáze, jsou provozně nazývaný hrany. Jako hrany si můžeme představit data o vlaku, který jede po grafikonu, kde počáteční a koncová hrana je určená číselníkem kódů stanic. V České republice se tomuto číselníku říká SR70 a lze ho najít na stránkách správy železniční cesty. (SŽDC, 2019)

4.2.4.1 Extrakce

V dřívějším návrhu probíhala extrakce dat pomocí PHP skriptu, kde byl SQL dotaz do databáze na požadovanou dávku dat. Tento způsob extrakce se ale neosvědčil, byl příliš pomalý. Proto se přešlo na nový způsob extrakce, a to pomocí db-linku a databázové procedury. Vzhledem k tomu, že vlastně probíhají dva kroky v jednom (extrakce a load) nebudu u hran už popisovat load. Celý proces extrakce – loadu zde shrnu v bodech.

1. Zneplatnění předchozích záznamů se stejnou periodou.
2. Zajištění potřebných informačních dat ze zdrojové databáze.
3. Vytvoření partitions.
4. Spuštění databázové procedury

4.2.4.2 Transformace

Z analýz dat po 3měsíčním období bylo i zde vytipováno k jakým chybám v datech dochází a pro některé byly navrženy korekce.

- Hnací vozidlo je na dvou místech současně – na jeden identifikátor vlaku existuje více hran se stejným pořadím.
    - Hrany je třeba vyselektovat a pozměnit jim identifikátor vlaku.
- Vlak má nulovou váhu – nelze opravit
- Vlak má špatně uvedenou číslo hnacího vozidla – číslo hnacího vozidla buď není uvedeno vůbec anebo není 12timístné. Korekce neexistuje.
- Vlak má neplatné datum – může nastat situace, že zdrojový systém nedostane správně datum a databáze si tam tak doplní rok 0. Tyto hrany jsou z výpočtu vyřazeny.

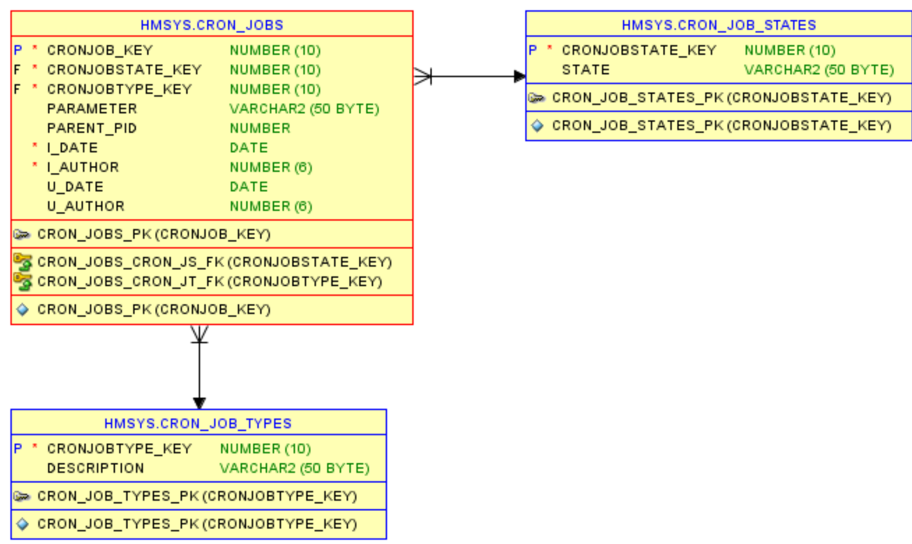
4.2.5 Plánovač úloh

Aplikace pro svůj provoz, zejména pro ETL, potřebuje plánovač úloh. Jako plánovač úloh by zvolen linuxový software Cron. Cron je software, který v určitý čas spouští

námi zadané úlohy. V hybridním modelu je naprogramovaná logika, která se stará o to, aby Cron spouštěl právě ty skripty, které čekají na odbavení. Celé to funguje tak, že je jeden hlavní PHP skript – cron_service.php, který Cron v opakovaných intervalech 5 minut spouští. Tento skript v sobě obsahuje příkazy, které spouští další skripty. Tyto příkazy jsou v cyklu, který iteruje, dokud nenarazí na limit. Limit spuštěných jobů je nastaven na 30. Celý tento proces je detailně logován logerem, který jsem výše popsal. Logy jsou dělený po měsíci, tedy každý měsíc vzniká nový soubor s logy.

4.2.5.1 **Databázový základ**

Cron service je něco jako hlídač, hlídající panel s tlačítky. Jednou za 5 minut se podívá na panel zjistí, že nějaký tlačítka blikají, tak je zmáčkne a zase si jich 5 minut nevšímá i když některá stále blikají. Tlačítka v této metafoře představují skripty, které se po spuštění mohou, ale i nemusejí přepnout do stavu hotovo, protože čekají až se ukončí jiný skript. Aby celý proces takto fungoval, vymyslel jsem v následující databázovou strukturu.



Obrázek 8 ERD cron jobů

Cron joby mohou nabývat následujících stavů:

1. Čeká
2. Běží
3. Ukončený
4. Chyba

Vysvětlení významu sloupců:

- cronjob_key – primární klíč
- cronjobstate_key – cizí klíč do tabulky stavů
- cronjobtype_key – cizí klíč do tabulky typů procesů
- parameter – parametr = klíčový atribut pro skript, aby věděl s jakými daty má pracovat
- parent_pid – příslušnost jobu k spuštěnému procesu
- i_date – datum zavedení
- i_author – autor zavedení
- u_date – datum zneplatnění/ukončení
- u_author – autor zneplatnění/ukončení

4.2.5.2 Skripty spouštěné plánovačem úloh

Protože skript cron_service.php jsem již popsal výše, plynule přejdu na praktickou ukázku kódu.

```
require_once('_includes.php');

$limit_jobs = 30;
$log = new data_processing_log(INCLUDE_PATH . 'csv_files' . DIRECTORY_S
EPARATOR . (new \DateTime())->format('Y-m'). '-cron_service.log');
$log->log_to_file('Načteno _includes.php');

try{
  db_connector::hm_connect();
  $waiting_jobs = dibi::fetchAll('SELECT CRONJOB_KEY, CRONJOBSTATE_KEY,
 CRONJOBTYPE_KEY, PARAMETER, PARENT_PID FROM CRON_JOBS WHERE cronjobsta
te_key = %i ORDER BY I_DATE', constants::cron_job_wait);
  $log->
log_to_file(dibi::translate('SELECT CRONJOB_KEY, CRONJOBSTATE_KEY, CRON
JOBTYPE_KEY, PARAMETER, PARENT_PID FROM CRON_JOBS WHERE cronjobstate_ke
y = %i ORDER BY I_DATE', constants::cron_job_wait));
```

```
  foreach ($waiting_jobs as $job){
    sleep(1);
    $running_jobs = dibi::fetchSingle('SELECT COUNT(cronjobstate_key) F
ROM cron_jobs WHERE cronjobstate_key = %i', constants::cron_job_runnig)
;
    $log->
log_to_file(dibi::translate('SELECT COUNT(cronjobstate_key) FROM cron_j
obs WHERE cronjobstate_key = %i', constants::cron_job_runnig));
    if($running_jobs > $limit_jobs) {
      $log-
>log_to_file("Kontrola: běžící $running_jobs > limit $limit_jobs . Konč
ím.");
      exit;
    }
    switch ($job->CRONJOBTYPE_KEY) {
      case constants::cron_type_get_ehvs:
        shell_exec('nohup /usr/bin/php -
q '.INCLUDE_PATH.'cron_scripts/download_snaps.php '.$job-
>PARAMETER.' '.$job-
>CRONJOB_KEY.' > /dev/null &'); //spustí příkaz v konzoli shell
        $log->
log_to_file("Spusteno: cron_scripts/download_snaps.php ".$job->
PARAMETER." ".$job->CRONJOB_KEY);
        break;
   .
   .
   .
```

Na ukázce můžeme vidět vytvoření logu a logování každého kroku. Dále si z databáze vytáhneme všechny joby ve stavu čekající. Poté iterujeme přes čekající joby a provádíme kontrolu, jestli počet běžících jobů nedosáhl maxima. Pokud ne, ve switch podle typu jobu spustíme příslušný skript.

Začátek spuštěného skriptu z ukázky vypadá následovně:

```
require_once(__DIR__.'/../_includes.php');

$period = $argv[1];
$job_key = $argv[2];
```

```
download_csv_by_period($period, $job_key);

function download_csv_by_period($period, $job_key){
  $log = new data_processing_log(INCLUDE_PATH . 'csv_files' . DIRECTORY
_SEPARATOR . "log-snaps.txt");
```

5 Závěr

Jako cíl práce jsem si stanovil demonstraci aplikační obsluhy datového skladu. V práci jsem představil stavbu webové aplikace nad datovým skladem od úplného začátku. Nejprve jsem popsal a ukázal, jak vybudovat jádro aplikace.

V jádře aplikace jsem předvedl, jak efektivně spravovat použité knihovny a poté ukázal, jak je používat pomocí automatického načítání. U jádra aplikace jsem ukázal, jak postavit adresářovou strukturu, tak aby byla jasná i pro programátora, který projekt uvidí poprvé. Další důležitou ukázkou, která je součástí jádra je MVC, kterým oddělím front-end od back-endu.

V práci jsem popsal ETL proces, který využívá tři datové zdroje. Tyto zdroje jsem podrobně rozebral z hlediska jejich charakteristik a předvedl ukázky, jak z těchto zdrojů čerpat data. Pro každý z těchto zdrojů jsem ukázal pomocné skripty, které jsou nezbytně nutné pro úspěšnou extrakci. U transformace jsem popsal typy chyb, se kterými se můžeme u jednotlivých datových zdrojů setkat a jak se s těmito chybami zachází.

V práci jsem představil základ problematiky stavby aplikace nad datovým skladem, sloužící pro výpočet měření spotřeb elektrických hnacích vozidel či elektrických jednotek. Hybridní model je webová aplikace nad datovým skladem, která má 20 uživatelů v produkčním prostředá využívajících aplikaci každý den. V ostrém provozu je nasazena a běží necelé dva roky. Každý měsíc aplikace importuje zhruba 10 miliónů řádků dat a s nimi provádí databáze výpočty spotřeb.

6 Seznam použitých zdrojů

Apachefriends. 2020. XAMPP. *XAMPP Apache.* [Online] 2020. [Citace: 25. 01 2020.] https://www.apachefriends.org/index.html.

Composer. 2019. Composer. *Composer A Dependency Manager for PHP.* [Online] 2019. [Citace: 25. 01 2020.] https://getcomposer.org/doc/00-intro.md.

ČD, Sborník. 2017. Vědeckotechnický sborník ČD. *Vědeckotechnický sborník ČD.* [Online] 2017. [Citace: 25. 02 2020.] https://vts.cd.cz/documents/168518/195501/43_komplet.pdf/456569d2-67fd-4f9b-badc-be7451cbe548.

Dibi. 2020. Dibi. *Database Abstraction Library.* [Online] 2020. [Citace: 26. 01 2020.] https://dibiphp.com.

jQuery. 2019. jQuery the javascript framework. *jQuery.* [Online] 2019. [Citace: 26. 12 2019.] https://jquery.com.

Kimball, Ralph a Joe, Caserta. 2004. *The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data.* místo neznámé : Wiley Publishing, Inc., 2004.

Microsoft. 2020. Visual Studio Code. *Visual Studio Code.* [Online] 2020. [Citace: 24. 01 2020.] https://code.visualstudio.com/docs.

Oracle. 2020. SQL developer. *Oracle SQL developer.* [Online] 2020. [Citace: 24. 01 2020.] https://www.oracle.com/cz/tools/downloads/sqldev-v192-downloads.html.

PHP.net. 2020. PHP hypertext preprocesor. *PHP.net.* [Online] 2020. [Citace: 05. 01 2020.] https://www.php.net.

Ross a Margy, Ralph Kimball and. 2013. *The Data Warehouse Toolkit: The Defi nitive Guide to Dimensional Modeling, Third Edition.* místo neznámé : John Wiley & Sons, Inc., 2013.

SŽDC. 2019. Portál provozování dráhy. *Číselník SR70.* [Online] 2019. [Citace: 01. 03 2020.] https://provoz.szdc.cz/Portal/ViewArticle.aspx?oid=34462.

SŽDC. 2019. Správa železnic. *Trakční elektřina.* [Online] 2019. [Citace: 20. 02 2020.] https://www.szdc.cz/dodavatele-odberatele/energetika/trakcni-elektrina.

Twig. 2019. Twig framework. *The flexible, fast, and secure template engine for PHP.* [Online] 2019. [Citace: 26. 12 2019.] https://twig.symfony.com.

Vrána, Jakub. 2017. *1001 tipů a triků pro PHP.* místo neznámé : Computer Press, Albatros Media a.s., 2017.

W3C. 2004. W3C. *Web Services Architecture.* [Online] 2004. [Citace: 14. 02 2020.] https://www.w3.org/TR/ws-arch/.

Zend. 2011. MVC. *Zend framework & MVC.* [Online] 2011. [Citace: 20. 12 2019.] https://web.archive.org/web/20111104151457/http://framework.zend.com/manual/en/learning.quickstart.intro.html#learning.quickstart.intro.mvc.

Zendulka, Jaroslav. 2015. Normalizace. *Databázové systémy.* [Online] 2015. [Citace: 28. 02 2020.] http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/5_2.pdf.

7 Přílohy