Univerzita Karlova

Pedagogická fakulta

Katedra informačních technologií a technické

BAKALÁŘSKÁ PRÁCE

Obohacení funkcionality G Suite využitím App Script

Extending the functionality of G Suite using App Script

Michal Tuček

Vedoucí práce:	PhDr. Jiří Leipert, Ph.D.
Studijní program:	Specializace v pedagogice (B7507)
Studijní obor:	Informační technologie se zaměřením na vzdělávání

2020

Odevzdáním této bakalářské práce na téma Obohacení funkcionality aplikací G Suite využitím App Script potvrzuji, že jsem ji vypracoval pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Dále potvrzuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

V Praze dne 1. 5. 2020

Na tomto místě bych rád poděkoval vedoucímu mé bakalářské práce PhDr. Jiřímu Leipertovi, Ph.D., za odborné vedení, trpělivost a rady, které mi v rámci vypracování této práce poskytl. Zároveň bych zde rád poděkoval i za podporu ze strany mé rodiny a přátel, především pak mé přítelkyni Markétě Svobodové za její neuvěřitelnou trpělivost a nápomocnost.

ABSTRAKT

Tato práce se zabývá teorií cloudu, přičemž se následně úzce zaměří na platformu G Suite a její nástroje. V další části představuje Google App Script, pomocí něhož můžeme jednotlivé nástroje G Suite mezi sebou vzájemně propojovat a obohacovat jejich vlastní funkcionalitu. Na několika praktických ukázkách vysvětlí metodiku práce s Google App Script a čtenáře zasvětí do této problematiky na úroveň začátečníka.

KLÍČOVÁ SLOVA

JavaScript, App Script, Google, G Suite, cloud, programování, skriptování, kódování, HTML, webové aplikace, vývoj, dokumenty Google, tabulky Google, kalendář Google, Gmail

ABSTRACT

This work deals with cloud theory, focusing closely on the G Suite platform and its tools. In the next part, it presents the Google App Script, with which we can connect the individual G Suite tools and enrich their functionality. There will be introduction to the methodology of working with Google App Script with a few practical examples which should introduce the reader to this issue at the beginner level.

KEYWORDS

JavaScript, App Script, Google, G Suite, cloud, programming, scripting, coding, HTML, web applications, development, Google Docs, Google Sheets, Google Calendar, Gmail

Obsah

1 Úvod 6

2 Cloud 8

2.1 Co je to cloud 8

2.2 Cloudové úložiště a jeho výhody 8

2.3 Aplikační vybavení cloudové platformy 9

2.4 Cloudová platforma G Suite 10

3 Google App Script 11

4 Vytvoření nového skriptu 12

4.1 Samostatný skript (Standalone script) 12

4.2 Skript vázaný k danému souboru (Bounded skript) 13

5 Editor 14

6 První skript 17

6.1 Publikování 19

6.2 Práce se vstupními daty 20

6.3 Logování a debugging 27

7 Objekty pro manipulaci s nástroji G Suite 33

8 Tabulka, vázaný skript a služby třetích stran 50

9 Zobecnění problematiky a tvorba doplňku 84

10 Závěr 115

Seznam použitých informačních zdrojů 116

Seznam příloh 119

1 Úvod

Od dob internetu byl zcela reformován způsob, jakým získáváme, spravujeme a sdílíme data. Z prvotního nasazení na lokálních terminálech se dnes běžně odkudkoliv připojíme k internetu i pomocí mobilních dat na chytrém telefonu. Této skutečnosti se také přizpůsobily i naše životy, kdy jsme od papírových diářů přešli k digitálním kalendářům, od dopisní korespondence k emailům a místo chození do knihoven získáváme informace přes internetové vyhledávače. V rámci posledních let se spíše začalo zaměřovat na přenos aplikační logiky na cloud za účelem jednoduchosti, méně výkonnějšího terminálu, ale pořád s možností služby naplno využívat, s benefitem přístupu odkudkoliv. Terminál nyní vůbec nemusí disponovat velkým úložištěm a vysokým pracovním výkonem, ale stačí malé místo na data a vše ostatní se synchronizuje na cloud. V minulosti, pokud chtěl člověk přenést data, musel je buď vypálit na CD, uložit na paměťové karty nebo flashky. Dnes se dají data předat prakticky instantně přes cloudové úložiště, kdy jen specifikujeme email příjemce pro zabezpečený přístup. Jakékoliv změny, které budou provedeny, budou druhému člověku ihned k dispozici.

Jelikož jsem nadšenec do moderních technologií a snažím se jít s dobou, tak se o tyto novinky zajímám a řídím se českým příslovím “Kolik jazyků znáš, tolikrát jsi člověkem.”. A to si myslím, že platí i na úrovni jazyků programovacích. Technologií je nyní tak rozsáhlé množství, že než se člověk v nějaké uchytí, měl by nejprve vyzkoušet více možností a tím zjistit, co mu sedí a co ho baví, jelikož každý je unikát a každý vyniká v něčem jiném. Při návrhu svých aplikací jsem často potřeboval jednoduchý administrační systém a ve finále to nebylo nic jiného než tabulka, nad kterou se muselo postavit nějaké rozhraní. Tak jsem si řekl, proč nepřejít od databázového uspořádání dat k tabulkového kalkulátoru, čímž se skloubí dvě věci do jednoho. Tabulka pak figuruje v aplikaci jako databáze dat, ale zároveň jako administrační rozhraní. Těžko se sdělí běžnému uživateli, ať si otevře například phpMyAdmin, nebo Adminer, a data si tam sám zapíše. Ale pokud se mu předloží data v podobě, na kterou je zvyklý, jako jsou například Excel či Google tabulka, tak to pro něj bude mnohem pohodlnější a pokud bude aplikace dostatečně jednoduchá, tak tím naprosto plnohodnotně vyhovuje svému účelu. To byl první moment, kdy jsem začal vnímat sílu a možnosti využití cloudových nástrojů.

Před nástupem MS Office 365 vycházela každých pár let nová verze. Pokud ji člověk potřebuje využít, musí si ji zakoupit, tzn. co nová verze, to další peníze. Osobně jsem koupil poslední verzi MS Office v roce 2007. Pro své účely se stále dá využít, má to určité možnosti a funkce, ale přeci jen, když už do toho člověk investuje jednou, proč si musí kupovat po několika letech novou verzi pouze s malými vylepšeními. V tento moment jsem se začal zajímat o možnosti alternativ, jako je například LibreOffice nebo OpenOffice. Uvědomil jsem si však, že v tu dobu jsem měl již chytrý telefon s operačním systémem Androidem, tím pádem i Google účet, tzn. mohl jsem využívat aplikace od Google, přesněji nástroje platformy G Suite. V porovnání s ostatními Open Source možnostmi měl Google výhodu v tom, že data budu mít dostupná odkudkoliv, což se mi hodilo například při studiu, protože ruku na srdce, kolikrát se stalo, že si člověk doma zapomněl flashku s potřebnými informacemi či soubory. Disponovalo to i další funkcionalitou, jako například ukládání historie změn, kde jsem se později mohl podíval na to, co jsem napsal a následně to mohl přehodnotit a vylepšit. Stejně tak i další výhodou je fakt, že je vše instantně k dispozici na jakémkoliv počítači bez nutnosti instalace a člověk tak není omezen aplikačním vybavením daného terminálu.

Na cloudové platformě je možné využít existující nástroje, které mezi sebou můžeme do jisté míry integrovat a rozvíjet jejich funkcionalitu. Osobně v tom vidím potenciál, než abych měl mnoho různých uzavřených aplikací, raději budu mít více univerzálních doplňků, které budu moci mezi sebou propojit a vzájemně používat.

2 Cloud

2.1 Co je to cloud

Termínem cloud (česky mrak či oblak) označujeme soustavu vzdálených serverů vykonávající různorodé funkce, jež jsou mezi sebou vzájemně propojené a vytváří tak jeden velký celek. Položíme-li si otázku, kde se cloud fyzicky nachází, tak odpovědí jsou datová centra po celém světě. Nejedná se tak o jednu konkrétní technologii, ale spíše o několik dílčích prvků, které svým propojením vytvářejí jednotný ekosystém. Různorodými funkcemi serverů míníme jejich určení k vykonávání specifických úkonů / služeb, mezi které můžeme zahrnout například webovou poštu, streamování videí, a tedy doručování obsahu obecně, či naopak jeho ukládání a vzdálenou správu. Místo využívání služeb a dat fyzicky uložených na vlastním disku lokálního osobního počítače, můžeme tak k těmto službám a datům přistupovat vzdáleně, a to bez závislosti na úložišti tohoto zařízení – data nám tak budou dostupná z jakéhokoliv zařízení a místa, kde je možné připojení k internetu.[1][2][3]

2.2 Cloudové úložiště a jeho výhody

V návaznosti na definici cloudu jako takového pak můžeme říct, že se jedná o vzdálený disk umístěný v datovém centru, jehož kapacita je pronajímána. Nikdy se však reálně nebude jednat pouze o jeden samostatný disk, ale o diskové pole RAID a to z důvodů zabezpečení dat vůči ztrátovosti z hlediska fyzického poškození disku. Z důvodu samotné redundance dat jsou pak oproti jednotnému úložišti dat na jednom vlastním lokálním disku data duplikována na disky další, a to i vícekrát v závislosti na úrovni RAID. Došlo-li by k poškození lokálního disku uživatele, hrozilo by, že data jsou nenávratně ztracena – v případě cloudu jsou však tato data zrcadlena a stačí pouze vadnou diskovou jednotku vyměnit bez ztráty dat, a to klidně i za provozu. Stejně tak je zde možnost dynamicky hýbat s kapacitou samotného úložiště bez potřeby řešení hardwarové stránky ze strany uživatele. Jako příklad si můžeme uvést známější služby jako Dropbox, OneDrive či Disk Google.[4][5]

Dropbox je webové úložiště, přičemž stejně jako ostatní cloudová úložiště umožňuje uživatelům synchronizovat, ukládat a sdílet dokumenty. OneDrive je cloudové úložiště od Microsoftu, které umožňuje uživatelům ukládat a chránit soubory, sdílet je s ostatními a dostat se na ně odkudkoli na všech svých zařízeních. Google Disk je cloudová služba od Googlu, umožňující uživatelům úschovu a sdílení dat, která mohou upravovat v integrovaném kancelářském balíku Google Docs. Tímto se Google Disk stal velkou konkurencí kancelářským aplikacím Office od Microsoftu.[6]

2.3 Aplikační vybavení cloudové platformy

Je rozdíl mezi obyčejným cloudovým úložištěm a cloudovou platformou. Cloudové úložiště samo o sobě umožňuje pouze ukládání a správu dat mimo lokální úložiště, kdežto cloudová platforma může mimo jiné disponovat cloudovým úložištěm. Obecně tedy můžeme říci, že se jedná o ucelenou infrastrukturu nejrůznějších aplikací určených ke specifickým operacím. Na příkladě textového souboru můžeme tedy říci, že v případě cloudového úložiště ho vnímáme pouze jako soubor, který můžeme přesunout, přejmenovat, zobrazit jeho obsah či odstranit, nedisponuje-li cloudové úložiště nějakými nadstandardními rozšiřujícími nástroji. Na druhou stranu cloudová platforma nám může ze sady svého vlastního dostupného aplikačního vybavení nabídnout k využití například textový procesor, kdy tak s tímto souborem můžeme manipulovat až na úrovni jeho vlastního obsahu. Vyjma formátování a základní editace mohou tyto editory disponovat i rozšířenou funkcionalitou, jako například verzováním jednotlivých sad změn dokumentu v případě aplikace Google Dokumenty. Dokument může v daný moment upravovat více uživatelů zároveň a k dispozici mají i vestavěný nástroj pro vzájemnou komunikaci v rámci dokumentu. Cloudová platforma nemusí být nutně omezena pouze na vlastní aplikace, ale může pomocí otevřeného API rozhraní integrovat i aplikace třetích strach.[7][8]

2.4 Cloudová platforma G Suite

Jedná se o cloudovou platformu firmy Google obsahující řadu nástrojů a aplikací pro spolupráci, komunikaci, ukládání dat, jejich prezentaci a vlastní správu. Platforma disponuje vlastním kancelářským balíkem obsahující webové aplikace pro správu dokumentů, tabulek a prezentací. Pro správu těchto i dalších typů souborů nám poslouží integrovaného prostředí cloudového úložiště Google Disk. Mimo jiné zde nalezneme i aplikace pro správu denní agendy jakožto Gmail, Kalendář, Google Tasks a další. Všechny tyto služby jsou dostupné i na mobilních platformách Android a IOS. Největší výhodou je využití této platformy pro prakticky neomezené množství uživatelů. Bohatou nabídku možností oceňují jak firmy v korporátní sféře, tak například i studenti a učitelé na akademické půdě. Právě kvůli tomu je nastaven cenový model na bázi předplatného dle počtu uživatelů. Všechny služby Google je možné provozovat i na vlastní doméně, kde můžeme libovolně nastavovat oprávnění, emaily a spravovat uživatele i skupiny jako takové. To můžeme uskutečnit díky rozsáhlé možnosti administrace, zabezpečení a integrace. Mezi největší přednosti patří nakonec také možnost provádění úprav v reálném čase, bezproblémovou kompatibilitu a výkonnou kontrolu sdílení.[9]

3 Google App Script

Jde o skriptovací jazyk řídící se pravidly ECMAScript založený původně na JS 1.6 s implementací i některých vlastností verzí 1.7 a 1.8 pro účely vývoje jednoduchého vývoje webových aplikací na platformu G Suite. Dnes používá V8 JavaScript Engine, což je stejný modul, jež využívá samotný webový prohlížeč Google Chrome či softwarový systém Node.js. Samotná technologie byla nasazena v roce 2009 a to původně pouze pro verze Google Apps Premier a Education Edition platformy G Suite. Veškeré projekty vytvořené za pomoci Google App Script jsou provozovány na straně serveru, jež je součástí infrastruktury Google.[10]

Jedná se o nástroj, který by měl dosáhnout jednoduché automatizace rutinních úkolů, jak skrze všechny nástroje poskytované Google, tak i skrze služeb třetích stran. Mimo jiné pohání všechny tzv. doplňky pro Google Docs, Sheets a Slides, jež obohacují dané služby o další funkcionalitu. Skript tak může být samostatný, vázaný pouze na daný soubor, či je-li jeho funkcionalita vyžadována k dispozici více souborům, tak může být publikován právě ve formě doplňku.[11]

Stejně jako v případě tabulek, dokumentů a dalších nástrojů i App Script má svůj hlavní panel, kde je k nahlédnutí přehled všech projektů a jejich informačních protokolů.

4 Vytvoření nového skriptu

4.1 Samostatný skript (Standalone script)

Jelikož Google App Script není běžnou součástí dostupných nástrojů pro běžné uživatele, bude třeba nástroj, bude ho potřeba připojit jako další aplikaci pro Google Disk.

1) Otevřete Google Disk a v oblasti zobrazených souborů a složek klikněte do prázdného místa pravým tlačítkem myši.

2) Z nově rozbaleného seznamu najeďte na možnost Více a následně z dalšího postranního podseznamu zvolte položku Připojit další aplikace.

3) Nově se vám otevře vnořené okno G Suite Marketplace, skrze který je možné instalovat nové aplikace a doplňky pro tuto platformu.

4) Do vyhledávače G Suite Marketplace zadejte "Google App Script" a stiskněte enter.

5) Až se vám zobrazí výsledky vyhledávání, požadovaná aplikace by měla být umístěna jako první, nést totožný název a mít minimálně přes milion stažení. Tuto položku rozklikněte a ujistěte se, že autorem doplňku (text hned pod názvem) je Google. Ve finále klikněte na podlouhlé tlačítko Instalovat.

6) V rámci instalace budete vyzváni k určitých oprávnění, jež aplikace ke svému provozu vyžaduje. Kliknutím na volbu Pokračovat se v novém okně webového prohlížeče přihlaste ke svému Google účtu, pod kterým chcete aplikaci využívat a vyčkejte, až bude instalace dokončena.

7) Nyní postupujte stejně, jako v krocích 1 a 2, ale místo zvolení položky Připojit další aplikace nyní vyberte nově přidanou možnost Google App Script.

8) Následně se vám v novém okně otevře nový projekt Google App Script.

4.2 Skript vázaný k danému souboru (Bounded skript)

1) V prostředí Google Disk vytvořte nový, či otevřete existující soubor, ke kterému chce vytvořit nový skript.

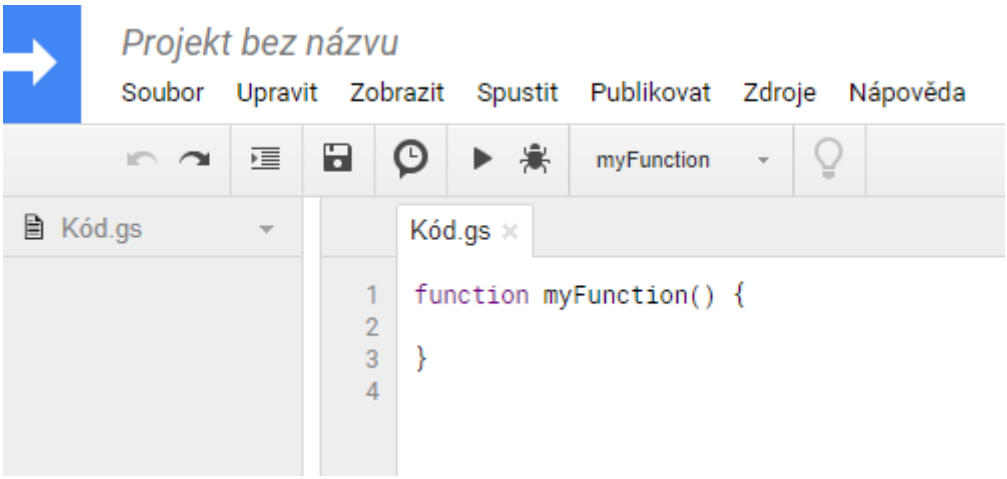
2) V otevřeném souboru vyberte z nejhornějšího menu položku Nástroje a v nově rozbaleném seznamu zvolte možnost Editor skriptu.

3) Následně se vám v novém okně otevře nový projekt Google App Script, který je svázán s daným souborem, pro který jste jej vytvořili.

5 Editor

Po otevření souboru skriptu se dostaneme do rozhraní jednoduchého online editoru, který nám nabídne základní funkcionalitu pro podporu vývoje. Existuje zde i možnost projekt vytvořit přímo na lokální pracovní stanici a následně ho propojit s existujícím projektem na straně cloudu. Toto řešení nám tak poskytne možnost využití vlastního editoru, který může být obohacen o rozšířenou funkcionalitu v případě specifických požadavků, pakliže by byl projekt většího rozsahu, či jen pro komfortnější vývoj.

Pod samotným menu nalezneme hlavní lištu s nejčastěji používanými nástroji. V užším levém panelu jsou vypsány veškeré soubory projektu, přičemž obsah zvoleného souboru nalezneme v pravém širším panelu.



Obr. 1 - Rozložení prvků online editoru

Než vůbec začneme programovat samotný skript, je třeba si nejprve představit některé ze základních nástrojů editoru, jež při práci často využijeme.

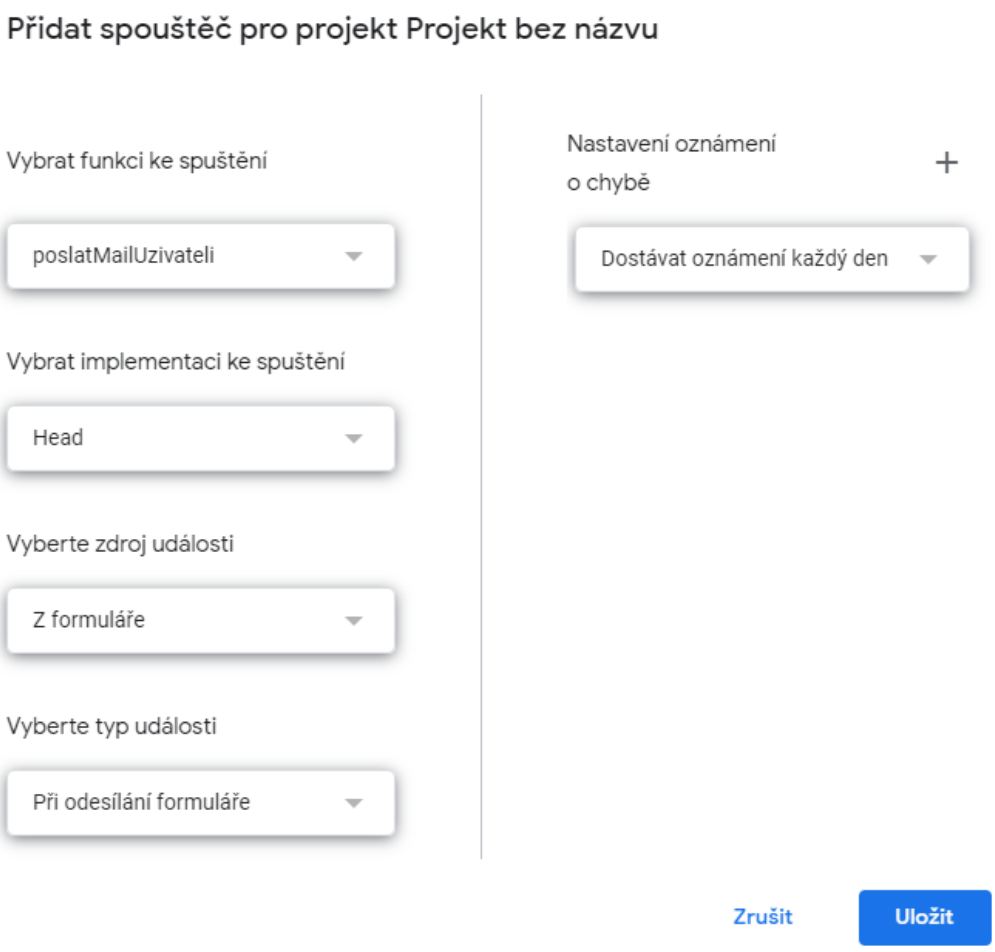
Jedním z takových nástrojů jsou spouštěče projektu, které jsou reprezentovány ikonou hodinových ručiček v bublině na panelu nástrojů. Spouštěče obecně využijeme za účelem vyvolání vlastní funkce vykonávající specifickou operaci, nastane-li situace odpovídající námi určeným pravidlům. Vezmeme-li si ku příkladu formulář, můžeme vytvořit funkci odesílající informativní email se zadanými daty uživateli, který formulář vyplnil. Následně pro tento formulář nastavíme spouštěč, který při každém odeslání formuláře výše zmíněnou funkci zavolá.

Při tvorbě každého spouštěče tak vytváříme pomyslné scénáře, kterým je vždy třeba specifikovat:

1. Co bude zdrojem události?
2. O jaký typ události se bude jednat?
3. Jaká funkce se bude po naplnění pravidel spouštět?

V případě našeho příkladu si nazveme danou funkci například poslatMailUzivateli(), přičemž by nastavení spouštěče bylo následující:

1. Zdrojem události bude formulář.
2. Typem události bude odeslání formuláře.
3. Volat se bude funkce poslatMailUzivateli().



Obr. 2 - Ukázkové nastavení spouštěče

Mimo jiné si můžeme na úrovni jednotlivých spouštěčů specifikovat četnost zasílání chybových reportů, které budou odesílány na email vlastníka skriptu.

V hlavním panelu App Script je k dispozici přehled spouštěčů napříč všemi projekty. Mimo jiné zde i existuje záložka Moje spuštění, kde nalezneme výpis protokolů a chyb všech spuštěných funkcí, včetně těch volaných spouštěči. Tyto informace nám následně pomohou při ladění těchto funkcí, jelikož právě sem můžeme logovat aktuální obsah určitých proměnných či námi specifikované zprávy. Ty nám pomohou za účelem lepší orientace v projektu, kdy si před výpisem chybové hlášky můžeme všimnout například poslední zalogované zprávy, která nám zde poslouží jako pomyslný orientační bod v těle funkce.

Při vývoji jednotlivých skriptů zde často budeme využívat jednotlivých služeb Google prostřednictvím jejich vlastních knihoven. Určité operace těchto knihoven, budou však ke své funkcionalitě vyžadovat i specifická oprávnění k přístupu k daným službám. Bude-li náš script například zapisovat nějaká výstupní data do dokumentů, využijeme knihovnu DocumentApp, přičemž bude uživatel vyzván k povolení oprávnění skriptu pro zápis do jeho dokumentů.

Chceme-li pro daný skript vymezit s jakými všemi službami bude pracovat, klikneme v menu editoru na záložku Zdroje a vybereme položku Pokročilé služby Google. V následně zobrazeném vyskakovacím okně nalezneme seznam všech dostupných služeb a jejich verzí. Služby jsou defaultně vypnuty a chceme-li tak nějakou využít, můžeme ji zde zapnout, přičemž se propíše do souboru manifestu jako nová závislost projektu.

Soubor manifestu obsahuje veškeré informace o projektu, jako jsou právě závislosti, oprávnění, časové pásmo a další. Soubor je prvotně u každého projektu skrytý a chceme-li ho zobrazit můžeme tak učinit kliknutím v menu na záložku Zobrazit, kde vybereme položku Zobrazit soubor manifestu. V seznamu souboru projektu by se nám měl následně nově zobrazit soubor appscript.json.

6 První skript

Ačkoliv je možnost obohacování jednotlivých nástrojů G Suite o vlastní funkcionalitu velmi lákavá a na první pohled nemusí vypadat složitě, určitě doporučuji mít alespoň základní znalost jakéhokoliv programovacího jazyka, nejlépe však JavaScriptu, jelikož i ten zde může být použit. Dalším významným plusem bude zkušenost s vývojem webových aplikací, byť i jen prezentací, z důvodu orientace v HTML, se kterým se zde také v některých případech setkáme. V neposlední řadě bude výhodou i všeobecné ponětí o jednotlivých metodách HTTP protokolu, se kterými se zde také pracuje.

Jak už bývá zvykem při prvotních hrátkách s novým jazykem, i zde nejprve vytvoříme ukázkový program, jehož hlavním účelem bude vypsání věty “Hello, World!” do jednoduché HTML šablony. Jelikož zde zatím nevyužijeme dalších služeb G Suite, bude se jednat o skript samostatný ve formě jednoduchého uživatelského rozhraní, na kterém si ukážeme syntaxi jazyka, některé ze základních funkcí a tříd, přenos dat mezi šablonou a skriptem a jejich logování.

Často v rámci jednoho projektu budeme pracovat s více soubory, jelikož i logiku samotného skriptu je dobrým zvykem pro větší přehled rozdělit dle funkcionality do více samostatných celků. Abychom vytvořili šablonu, musíme v menu kliknout na záložku Soubor, vybrat položku Nové a následně Soubor HTML. Jelikož se bude jednat o hlavní stránku naší webové aplikace, vycházejme s konvence pojmenovávání při vývoji webových aplikací a soubor tak nazvěme index a těla přidáme nadpisový tag h1 s textem “Hello, World!”.

```
<!DOCTYPE html>
<html>
  <head>
    <base target="_top">
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

Pokud by se jednalo o běžný webový server, index by se automaticky pokládal za hlavní soubor, který by se také na výstupu zobrazil. V tomto případě však vše řídí skript, který momentálně nemá o existenci šablony žádnou informaci. Je tedy třeba šablonu inicializovat a předat ji na výstup. Po celou dobu vývoje je nám k dispozici rozsáhlá dokumentace a pokud bychom nahlédli do příručky Web Apps, dozvěděli bychom se následující:

"Oba samostatné skripty i vázané skripty k daným G Suite aplikacím je možné transformovat do webových aplikací, splňují-li základní dva požadavky. Zaprvé, skript musí obsahovat jednu z funkcí doGet(e) či doPost(e). Zadruhé, výstupem zmíněných funkcí musí být buďto objekt HTML služby HtmlOutput, nebo objekt TextOutput služby Content." [12]

Jelikož naším cílem je na výstup vypsat HTML šablonu, použijeme tedy metodu doGet(), ve které vrátíme HTML objekt. V metodě doGet() tedy předáme na výstup pomocí třídy HtmlService obsah souboru index.html. Budeme-li vycházet z ukázkového kódu v dokumentaci práce se šablonami, měl by náš skript vypadat následovně:

```
function doGet(e) {
  return HtmlService.createHtmlOutputFromFile('index');
}
```

Nyní je náš projekt připraven a můžeme publikovat ho publikovat, jako webovou aplikaci.

6.1 Publikování

Abychom vůbec byli schopni aplikaci otestovat, je třeba nejprve projekt soukromě pod aktivním účtem publikovat. Tuto možnost nalezneme v menu pod záložkou Publikovat, kde zvolíme možnost Zprovoznit jako webovou aplikaci. Zobrazí se nám nové okno s dodatečným nastavením pro publikování, kdy musíme zvolit verzi projektu, přičemž v případě prvotního publikování je vždy nová. Je zde použito i základní verzování, kdy se dělají snapshoty stavu kódu během ukládání. Buďto nové změny zaktualizují stávající snapshot dané verze, nebo vytvoří verze nová se stručným popisem provedených úprav. Dále je zde možnost ještě nastavit pod jakým účtem se bude aplikace spouštět a kdo všechno bude mít k aplikaci přístup.

V našem případě chceme projekt publikovat privátně a spouštět ho pod naším účtem, tak nic nezměníme a pouze klikneme na tlačítko Deploy. K dispozici nyní máme odkaz na aktuální verzi projektu, na kterém si můžeme aplikaci prohlédnout a otestovat. Doporučuji ale využít druhého odkazu, který najdeme hned pod tím prvním pod slovy “latest code”. Ten nám umožní sledovat změny bez nutnosti publikovat novou verzi, což nám velmi usnadní testování během vývoje.

Přejdeme-li na výše zmíněný odkaz, vidíme, že se nám zobrazuje “Hello, world!”. Skript tedy inicializoval šablonu a její obsah vypsal do prohlížeče, jak jsme požadovali. Ačkoliv je náš první projekt dokončen, moc reálné využití v praxi nemá. V případě webové aplikace totiž často očekáváme interakci ze strany uživatele, který nám předává určitá data, jež je naším úkolem zpracovat.

6.2 Práce se vstupními daty

Pojďme tedy naši aplikaci obohatit o vstupní pole, jehož obsah následně vypíšeme zpět do šablony. Skript by například mohl jako vstupní datum přijmout jméno a datum narození. Na výstup by pak vrátil datum se dnem svátku pro jméno a počet uplynulých dní od data narození.

Začněme nejprve úpravou stávající šablony, do které přidáme vstupní pole:

```
<label>Křestní jméno: </label><input id="name" type="text">
<label>Datum narození</label><input id="date" type="date">
<button id="btn">Odeslat</button>
```

Uživatel bude moct vyplnit své křestní jméno a zvolit své datum narození a pomocí tlačítka Odeslat aktivujeme skript. Jenže otázkou je, jakým způsobem můžeme vyvolat ze šablony funkci App Scriptu? Za tímto účelem využijeme JavaScript, za pomoci kterého vytvoříme EventListener, jež bude sledovat, zda-li došlo ke stisknutí odesílacího tlačítka. Vyvolání samotné funkce skriptu pak provedeme vyvoláním naší funkce nad google.script.run, přičemž si ji nazveme například processInputData(). Výsledný kód bude tedy vypadat následovně:

```
<script>
  document.getElementById('btn').addEventListener('click',
function()                                                    {
    google.script.run.processInputData(
      document.getElementById('name').value,
      document.getElementById('date').valueAsNumber
    );
  });
</script>
```

Pozn.: U data místo parametru value voláme parametr valueAsNumber, jelikož výstupem této metody je unix timestamp, se kterým se mnohem lépe pracuje při inicializaci nového Date objektu v App Scriptu.

Přejděme nyní na stranu App Scriptu a zamysleme se, jakým způsobem navrhneme a následně i zrealizujeme řídící logiku. Jelikož si datum narození předáváme ve formě unix timestampu, stačí jen tuto hodnotu předat konstruktoru objektu Date a následně jen tento objekt v určitém formátu vypsat. Složitější situace bude v případě svátku, kde potřebujeme vůbec zajistit informace o tom, jaký svátek se daný den v roce slaví.

Ovšem aplikace tohoto typu již existují, takže je stačí pouze upravit do formátu, se kterým se nám bude pohodlně pracovat. Jelikož máme dny jednotlivých měsíců v roce, nabízí se data vložit do vícerozměrného pole, kdy jednotlivá pole budou reprezentovat měsíce jejichž prvky pak reprezentují jednotlivé dny.

Následně pak nad tímto polem vytvoříme funkci, jejíž vstupní proměnnou bude křestní jméno, jež se pokusí v poli najít. Ačkoliv se zde nabízí užití funkce forEach(), zvolíme funkci some(), která nám umožní sofistikovaněji ukončit procházení prvků pole v momentě, kdy najdeme shodu a vrátíme na výstup cyklu true. Výstupem této funkce bude objekt Date s datem svátku. V případě, že však jméno nebude v poli nalezeno, vrátíme na výstupu null.

```
const nameDays = [
  ["", 'Nový rok', 'Karina', 'Radmila', 'Diana', 'Dalimil', 'Tři
králové', 'Vilma', 'Čestmír', 'Vladan', 'Břetislav', 'Bohdana',
'Pravoslav', 'Edita', 'Radovan', 'Alice', 'Ctirad', 'Drahoslav',
'Vladislav', 'Doubravka', 'Ilona', 'Běla', 'Slavomír', 'Zdeněk',
'Milena', 'Miloš', 'Zora', 'Ingrid', 'Otýlie', 'Zdislava',
'Robin', 'Marika'],
  ...
];

function getNameDayDateByName(name) {
  var date = null;
  nameDays.some((monthNameDays, month) => {
    if (monthNameDays.indexOf(name) != -1) {
      date = new Date(
        today.getUTCFullYear(),
        month,
        monthNameDays.indexOf(name) + 1
      );
      return true;
    }
    return false;
  });
  return date;
}
```

Pozn.: V případě měsíce se používá index, kdy 0 reprezentuje první měsíc v roce Leden, takže nejsou vyžadovány žádné úpravy. Naopak u jednotlivých dnů měsíce je však číslování pojato začátkem od 1 a tedy je třeba index pole o 1 navýšit.

Pro lepší orientaci v projektu tento kód uložíme do nového souboru skriptu, který nazveme nameDays, čímž je na první pohled jasné, co můžeme v souboru očekávat za obsah. Všechny soubory skriptu jsou automaticky inicializovány, takže zde není třeba volat funkce podobné include či require, které můžeme znát například z PHP.

Abychom si zároveň předvedli, jakým způsobem je realizováno používání proměnných v šablonách, vytvoříme si šablonu novou a pojmenujeme ji například result.html, přičemž bude mít následující strukturu:

```
<?= firstName ?> slaví svátek <?= nameDayDate ?>
<br>
Od Vašeho narození uběhlo již <?= daysSinceBirth ?> dní.
```

Mezi znaky **<?=** a **?>** můžeme umístit proměnnou či funkci skriptu, která bude následně nahrazena jejím obsahem, či výstupem. V případě, že bychom chtěl funkci pouze vyvolat, ale již následně nepracovat s jejím výstupem, který nemusí být vždy nutně očekáván, můžeme využít znaků **<?** a **?>**.

Teď už jen zbývá tyto samostatné celky spojit dohromady uvnitř funkce processInputData() a výsledek předat zpět šabloně. Zde využijeme matematických zaokrouhlení a absolutní hodnoty při výpočtu uplynulých dnů od narození, za pomoci vestavěného objektu Math. K dispozici nám je také mimo jiné i objekt třída Utilities, která nabízí i funkci formatDate() pro formátování datumu. Ta mezi vstupními parametry požaduje i časovou zónu, kterou získáme pomocí objektu Intl, jež slouží k internacionalizaci a vrátí nám tak aktuální časové pásmo.

```
const today = new Date();
// hours * minutes * seconds * milliseconds
const oneDay = 24 * 60 * 60 * 1000;

function processInputData(firstName, birthDate) {
  var nameDayDate = getNameDayDateByName(firstName);
  birthDate = new Date(birthDate);

  var html = HtmlService.createTemplateFromFile('result');
  html.firstName = firstName;
  html.nameDayDate = nameDayDate == null ?
    '\[neznámé datum\]'
    : Utilities.formatDate(
      nameDayDate,
      Intl.DateTimeFormat().resolvedOptions().timeZone,
      "dd.MM."
    );
  html.daysSinceBirth = Math.round(
    Math.abs((today.getTime() - birthDate.getTime()) / oneDay)
  );
  return html.evaluate().getContent();
}
```

Zásadní rozdíl je zde však při práci se šablonou. Stejně, jako prvně i zde inicializujeme šablonu, ale místo toho, abychom ji hned předali na výstup funkce si ji tentokrát uložíme do proměnné, abychom nejprve mohli definovat jednotlivé proměnné, s nimiž chce pracovat. To učiníme tak, že zavoláme dosud neexistující vlastnost objektu HTML a přiřadíme jí novou hodnotu. V případě proměnné firstName tedy zavoláme html.firstName, do které hodnotu firstName zapíšeme, čímž ji nadefinujeme. Při vracení je třeba ještě zavolat funkci evaluate(), která zajistí, aby proměnné a funkce byly řádně nahrazeny. Jelikož budeme pouze přidávat obsah k existující šabloně, zavoláme navíc ještě metodu getContent(), abychom na výstupu měli čistý HTML kód.

Nyní už jen zbývá výstup App Scriptu na straně uživatelského rozhraní uchopit a vypsat. K tomu ale budeme muset provést úpravu volání skriptu, jelikož momentálně nepracujeme jakkoliv s výstupem funkce. Abychom mohli s jejím výstupem pracovat, musíme přidat zpětné volání a funkci, které budou data předána. K tomu nám poslouží metoda withSuccessHandler() před tím, než zavoláme samotnou funkci processInputData(). Následně přidáme do šablony div, ve kterém budeme vypisovat výstupní data a vytvoříme JavaScript funkci showResult(), které výše zmíněným zpětným voláním předáme výstupní data. Ta pak už jen připojí získaný obsah jako vnitřní HTML div elementu.

Tělo šablony index.html se všemi podpůrnými funkcemi by mělo finálně vypadat takto:

```
<label>Křestní jméno: </label>
<input id="name" type="text">
<label>Datum narození</label>
<input id="date" type="date">
<button id="btn">Odeslat</button>
<br>
<div id="result"></div>

<script>
  document.getElementById('btn')
    .addEventListener('click', function() {
      google.script.run.withSuccessHandler(showResult)
        .processInputData(
          document.getElementById('name').value,
          document.getElementById('date').valueAsNumber
        );
    });

  function showResult(resultHTML) {
    document.getElementById('result').innerHTML = resultHTML;
  }
</script>
```

6.3 Logování a debugging

Jelikož nyní budeme pracovat s daty, které může vkládat uživatel, je třeba si představit integrovaný ladící nástroj a možnost logování proměnných či vlastních textů pro lepší orientaci v kódu v případě pádu aplikace z důvodu prozatím neznámé chyby.

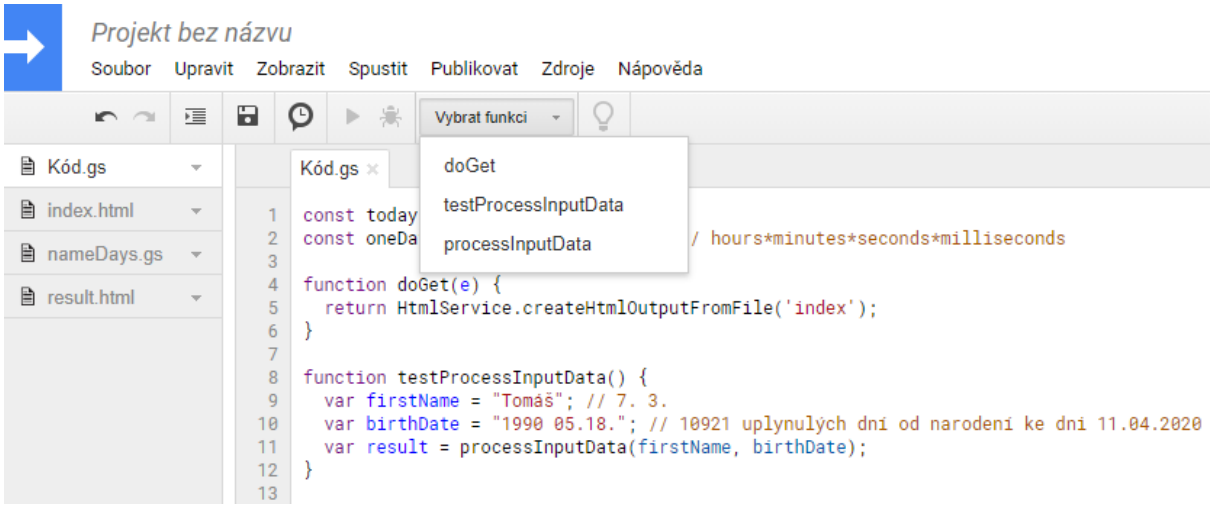
Stejně jako v případě množství jiných editorů zde můžeme využít tzv. breakpointů, neboli zarážek, které nám umožní program za běhu v daném místě pozastavit a provést inspekci běhového prostředí. Toto využijeme převážně za účelem prohlédnutí si hodnot v nejrůznějších proměnných v rámci lokální funkce, celého skriptu i v globálním kontextu.

Pojďme si ukázat využití debuggeru za pomoci testovací funkce, jež otestuje funkčnost námi vytvořeného kódu. Stanovme si hodnoty dvou vstupních proměnných, jež jsou funkcí processInputData() požadovány a rovnou si za pomoci dostupných online nástrojů zjistěme očekávané výstupní hodnoty. Zvolíme-li například jméno Tomáš, kdy víme, že svátek slaví 7.3. a pakliže stanovíme datum narození testovací osoby na 18. 5. 1990, tak využitím různých online počitadel zjistíme počet uplynulých dní k dnešnímu dni.

Máme-li stanovené hodnoty testovacích dat, vytvoříme si testovací funkci testProcessInputData(), jejíž obsah bude následující:

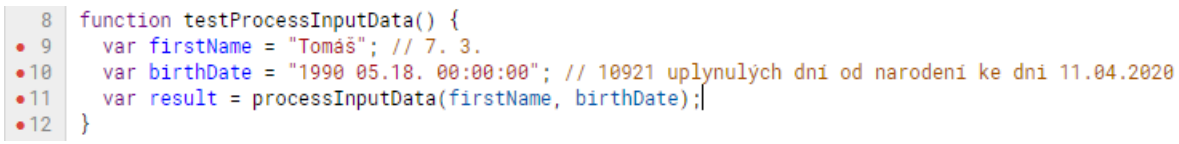
```
function testProcessInputData() {
  var firstName = "Tomáš"; // svátek slaví 7. 3.
  var birthDate = "1990 05.18.";
  // 10921 uplynulých dní od narození ke dni 11.04.2020
  var result = processInputData(firstName, birthDate);
}
```

Chceme-li nyní využít ladícího nástroje, musíme nejprve vybrat funkci, kterou chceme ladit. Za tímto účelem najdeme vedle ikony brouka seznam, ve kterém by se měly nacházet všechny dostupné funkce v rámci aktivního souboru.



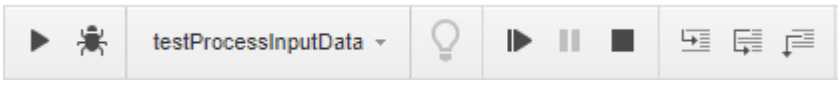
Obr. 3 - Výběr funkce k ladění

Následně kliknutím na číslo řádky vytvoříme několik breakpointů, na kterých se chceme zastavit a prohlédnout si hodnoty proměnných, které nás zajímají.



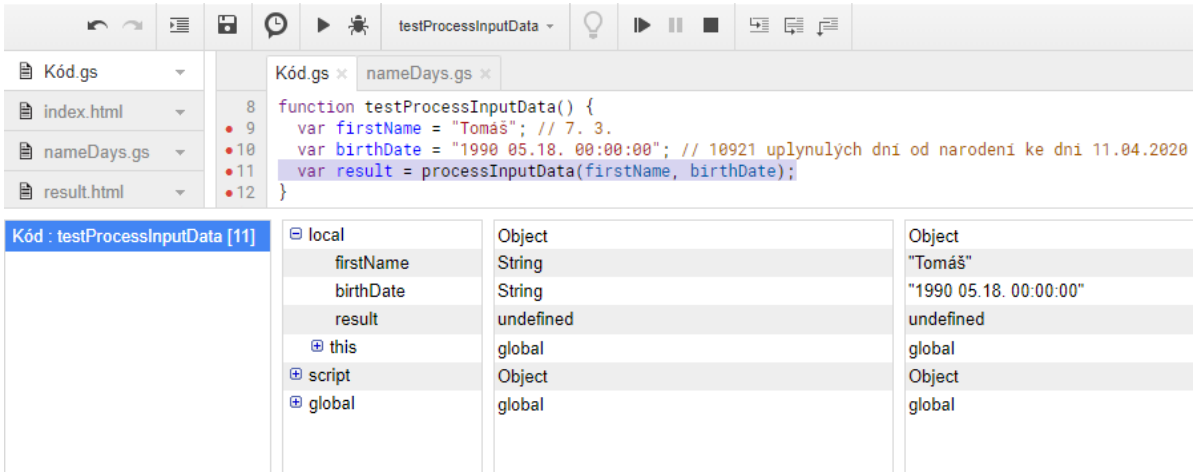
Obr. 4 - Ukázkové nastavení breakpointů

Máme-li zvolenou funkci pro ladění, která obsahuje alespoň jeden dosažitelný breakpoint, můžeme kliknout na ikonu brouka a začít tak ladění samotné, přičemž se rovnou zastavíme na prvním dosaženém breakpointu. Provedením této akce se nám v liště nástrojů zobrazí nové ikony, které nám při ladění nabídnou další možnosti jako například způsob následujícího krokování (chceme-li se například kouknout do těla vnořené funkce a zkoumat jí), přechod na následující breakpoint či předčasné ukončení ladění.



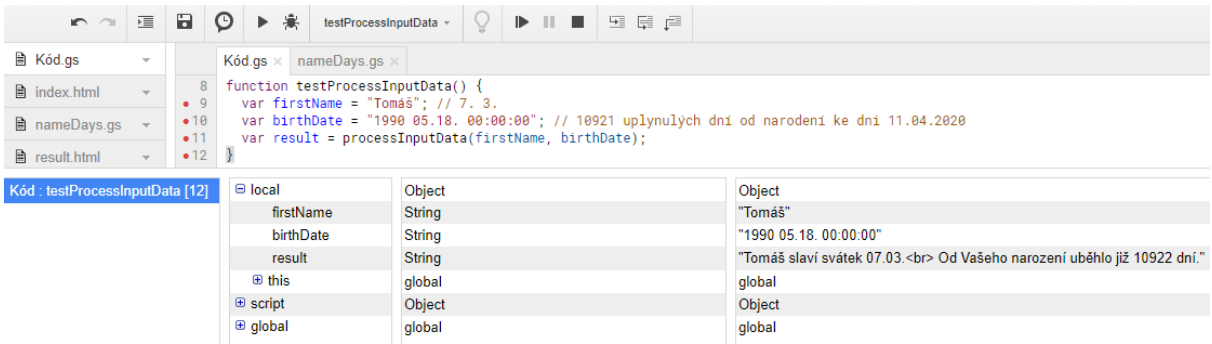
Obr. 5 - Debugovací lišta s možnostmi při ladění

Kliknutím na ikonu přehrávání v ladícím sekci budeme pokračovat na další breakpoint. Pokračujme tedy dále, dokud nedosáhneme na řádek 11.



Obr. 6 - Obsah proměnných při ladění

Zde si můžeme všimnout, že v případě, kdy stojíme na řádku, kde nás zajímá obsah proměnné při inicializaci její hodnoty, tak požadovanou informaci nezjistíme. Breakpoint se zarazí na řádce vždy před spuštěním vlastního kódu, jež se na řádce nachází. V tomto případě ještě tedy vůbec nedošlo k vyvolání funkce processInputData() a chceme-li tak znát obsah proměnné result, musíme provést ještě jeden krok vpřed na řádek 12.



Obr. 7 - Obsah proměnných při ladění v dalším kroku

Nyní již hodnotu proměnné result vidíme a můžeme si tak ověřit výsledné hodnoty. Ačkoliv datum svátku pro zadané jméno sedí, počet uplynulých dní od narození se nám rozchází o jednu. To bude nejspíš zapříčiněno tím, že mezi uplynulé dny počítáme i aktuální den, kdežto použité online nástroje ho do výsledného počtu nezahrnují. Teď už záleží vždy pouze na samotném autorovi kódu, jakým způsobem chce, aby počítadlo fungovalo a jaké veškeré dny by mělo zahrnout.

Z osobní zkušenosti jsem však ladící nástroj použil jen zřídka kdy a cennějším nástrojem pro mě byl třída Logger, která zapisuje do spouštěcích protokolů. V momentě, kdy se pracuje se vstupními daty uživatele za běhu, debugger zde nepomůže. Chceme-li tedy zapsat aktuální hodnotu proměnné či vlastní personalizovanou zprávu do protokolu spuštění, stačí vyvolat funkci Logger.log() a na vstup jí předat požadovaná data. Můžeme si tak například označit začátek specifického segmentu kódu, či využít vlastní chybové zprávy skrze log zachycením výjimky pomocí try/catch. Na příkladu předchozí testovací funkce bychom mohli za použití Loggeru zalogovat data následujícím způsobem:

```
function testProcessInputData() {
  Logger.log("Začátek funkce testProcessInputData()");
  var firstName = "Tomáš"; // 7. 3.
  var birthDate = "1990 05.18. 00:00:00";
  // 10921 uplynulých dní od narodení ke dni 11.04.2020
  Logger.log(
    "processInputData(" + firstName + ", " + birthDate + ");"
  );
  var result = processInputData(firstName, birthDate);
  Logger.log(
    "Výsledek volání funkce processInputData() => " + result
  );
  Logger.log("Konec funkce testProcessInputData()");
}
```

Pokud nyní chceme funkci otestovat, tak namísto použití debuggeru zvolenou funkci jen spustíme kliknutím na sousední ikonu přehrání. Pak už stačí jen nahlédnout do spuštění projektů a najít si odpovídající záznam, kde můžeme nahlédnout do protokolů v rámci spuštění dané funkce.

Projekt	Funkce	Typ	Čas zahájení	Délka	Stav
Projekt bez názvu	testProcessInputData	Editor	11. 4. 2020 15:16:48	0.278 s	Dokončeno

Protokoly služby Stackdriver

```
11. 4. 2020 15:16:48    Informace  Začátek funkce testProcessInputData()
11. 4. 2020 15:16:48    Informace  processInputData(Tomáš, 1990 05.18.
                                   00:00:00);
11. 4. 2020 15:16:48    Informace  Výsledek volání funkce processInputData() =>
                                   Tomáš slaví svátek 07.03.<br>
                                   Od Vašeho narození uběhlo již 10922 dní.
11. 4. 2020 15:16:48    Informace  Konec funkce testProcessInputData()
```

OBNOVIT

Obr. 8 - Záznam spuštění a jeho protokoly

Občas může dojít k takovým chybám, o kterých bychom chtěli dostat informaci nejlépe ihned, abychom kód mohli co nejdříve opravit. Můžeme si tak vytvořit vlastní obslužné funkce, jež budeme volat v try/catch blocích v případě odchycení výjimky. Logger nám umožňuje získat všechny logy dané relace, které spolu se samotnou odchycenou výjimku odešleme na určenou emailovou adresu. V našem případě, kdy je skript privátní a spouštíme ho jen my, bychom mohli vytvořit následující error handler:

```
function someFunction() {
  try {
    ...
  } catch(e) {
    errorHandler(e);
  }
}

function errorHandler(e) {
  var recipient = Session.getActiveUser().getEmail();
  var subject = 'Nastala neočekávaná chyba: ' + e;
  var body = Logger.getLog();
  MailApp.sendEmail(recipient, subject, body);
}
```

7 Objekty pro manipulaci s nástroji G Suite

V posledním odstavci poslední kapitoly jsme si v poslední ukázce kódu představili možnost personalizované práce s výjimkami, kde jsme využili vestavěného objektu MailApp za účelem odeslání emailu s aktuálními protokoly relace. Těchto objektů existuje řada, přičemž každý reprezentuje daný nástroj, jako jsou například dokumenty, tabulky, mail a další. Již v předchozích kapitolách jsem zmínil, že chceme-li pomocí těchto objektů manipulovat s některým z těchto nástrojů, bude takový skript ke své funkcionalitě vyžadovat odpovídající oprávnění. V této kapitole si tedy na praktické ukázce vysvětlíme práci s některými z těchto objektů a všech dalších souvislostí s nimi spjatých.

Vytvořme si tedy nový projekt, jež bude obsahovat webové rozhraní, kde bude moct uživatel opět vyplnit několik vstupních polí, které povedou k vytvoření kalendářové události. Mimo jiné budeme veškeré takto vytvořené události zaznamenávat pro přehled do tabulky. Dále si vytvoříme univerzální šablonu, z níž vygenerujeme personalizovaný obsah emailu, který pozvané osobě odešleme. Ačkoliv je účel této ukázky čistě demonstrativní, určitě bychom nalezli nějaké praktické využití, jako například rozesílání svatebních poznámek apod.

Pro zjednodušení budeme uvažovat, že veškeré události budou pouze v rámci jednoho dne. V předchozí ukázce jsme jako vstupních parametrů využívali křestního jména a datumu, které zde zachováme a přidáme k nim čas od, čas do, název události a email příjemce.

Známe-li veškeré vstupní proměnné, můžeme přejít k návrhu šablony. Pokud bychom přistupovali k logice získávání vstupních proměnných stejně, jako v předchozím návrhu, výsledná šablona by mohla mít následující podobu:

```
<!DOCTYPE html>
<html>
  <head>
    <base target="_top">
  </head>
  <body>

    <label>Jméno: </label><input id="name" type="text">
    <label>Gmail: </label><input id="mail" type="email">
    <label>Event: </label><input id="event" type="text">
    <label>Datum: </label><input id="date" type="date">
    <label>Od: </label><input id="time_from" type="time">
    <label>Do: </label><input id="time_to" type="time">
    <button id="btn">Uložit událost</button>

    <script>
      document.getElementById('btn')
        .addEventListener('click',function() {
          var date = document.getElementById('date');

          google.script.run.createEnhancedEvent(
            document.getElementById('name').value,
            document.getElementById('mail').value,
            document.getElementById('event').value,
            date.value + " "
            + document.getElementById('time_from').value,
            date.value + " "
            + document.getElementById('time_to').value
          );
        });
    </script>

  </body>
</html>
```

Pracujeme-li s jednou či dvěma vstupními proměnnými, může být toto řešení efektivní. Zvýší-li se nám však počet vstupních proměnných, přestává být toto řešení elegantním a nabízí se nám zde možnost užití formuláře. Ten nám umožní přistupovat k hodnotám vstupních polí objektově bez nutnosti vyhledávání elementu dle jeho id. Využijeme-li takového přístupu, šablona bude vypadat následovně:

```
<!DOCTYPE html>
<html>
  <head>
    <base target="_top">
  </head>
  <body>
    <form onsubmit="event.preventDefault(); onFormSubmit(this);">
      <label>Jméno: </label><input name="name" type="text">
      <label>Gmail: </label><input name="mail" type="email">
      <label>Event: </label><input name="event" type="text">
      <label>Datum: </label><input name="date" type="date">
      <label>Od: </label><input name="timeFrom" type="time">
      <label>Do: </label><input name="timeTo" type="time">
      <input type="submit" value="Uložit událost" />
    </form>
    <script>
      function onFormSubmit(form) {
        google.script.run.processForm(form);
      }
    </script>
  </body>
</html>
```

Samozřejmě je zde třeba zabránit samotnému odeslání formuláře a data předat odpovídající funkci na straně App Scriptu. Opět lze toto řešení aplikovat různými způsoby dle zvyklostí programátora, jako například užitím event listeneru, či voláním vlastní funkce nad google.script.run přímo ve vlastnosti onsubmit formuláře. Já jsem zvolil takovýto postup už jen z toho důvodu, že nám nabízí možnost následně nějakým způsobem pracovat se šablonou po vykonání skriptu. Můžeme tak například vyprázdnit vyplněné vstupní pole formuláře a vypsat informační zprávu o úspěšném, či neúspěšném provedení akce v závislosti na vrácené hodnotě. Nesmíme ale zapomenout na fakt, že chceme-li nějakým způsobem pracovat s výstupem tohoto skriptu, musíme opět při volání skriptu přidat funkci withSuccessHandler().

Jelikož jsme si již na začátku vymezili rozsah a funkcionalitu projektu, pojďme si rozdělit logiku do základních dílčích celků. Víme, že budeme chtít událost zaznamenávat do seznamu v tabulce. Budeme tedy potřebovat vytvořit novou tabulku a získat její identifikátor, abychom mohli tabulku inicializovat v App Scriptu a předat ji obslužnému objektu SpreadsheetApp, jež nám s ní umožní manipulovat. Do jednotlivých sloupců pak budeme ukládat jméno osoby, email, název události, datum a čas od / do. Odstraníme-li zbylé sloupce a všechny řádky, měli bychom mít takto vypadající tabulku:

	A	B	C	D	E
1	Osoba	Email	Událost	Od	Do
2					

Obr. 9 - Struktura tabulky pro záznamy událostí

V aktivním okně prohlížeče, kde máme tabulku otevřenou, můžeme její identifikátor získat jednoduše z url, ve kterém se nachází. Jedná se o část odkazu, za lomítkem následujícím po písmeni d až do dalšího lomítka. Pro upřesnění zde uvedu url adresu mé tabulky se zvýrazněnou částí, kde se její identifikátor nachází.

https://docs.google.com/spreadsheets/d/
1hHQiiOmRhdNg8jNRBvor2584-QVyDXkTAyHvPbcNFcw
/edit

Podobným způsobem si připravíme i další prerekvizitu, kterou je soubor šablony se zástupnými proměnnými pro pozvánku. Strukturu zástupných proměnných si můžete určit sami dle svých vlastních požadavků. Jak už může být z předchozích ukázek kódu zřejmé, osobně se držím anglických názvů, přičemž veškeré názvy koncipuji na základě camelCase, neboli tzv. velbloudí notace. Pro zástupné proměnné jsem naopak zvolil snake_case, neboli hadí notaci, o velkých písmenech a jako speciální poznávací znaky jsem zvolil dvě složené závorky na obou stranách. Styl zápisu je tedy následující:

{{ZASTUPNA_PROMENNA}}

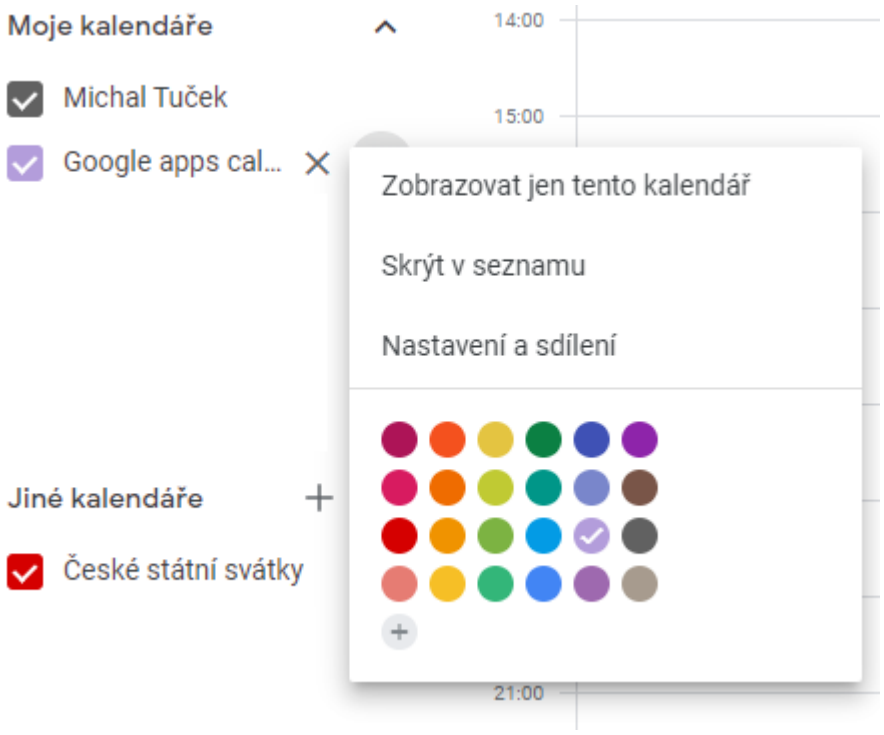
Vytvoříme si tedy nový soubor, jehož obsah bude například takovýto:

“Vážená paní / Vážený pane {{NAME}},

dovolujeme si Vás srdečně pozvat na událost “{{EVENT}}”, která se bude konat v termínu {{FROM}} - {{TO}}”

Vlastní text a konvenci pojmenování můžete samozřejmě dle svých osobních preferencí změnit. Stejným způsobem jako v případě tabulky získáme i zde identifikátor souboru, abychom dokument mohli následně inicializovat v App Scriptu a předat ji obslužnému objektu DocumentApp, který nám s ním umožní manipulovat. Jelikož zde však nebudeme manipulovat se samotnou šablonou, ale s její kopií, bude zapotřebí použít objekt DriveApp, který dle identifikátoru soubor získá, vytvoří jeho kopii a s tou budeme následně pracovat.

Posledním krokem v rámci přípravy je kalendář. Zde můžeme využít našeho osobního, či vytvořit nový pouze pro tento projekt. Z hlediska testování, kdy budeme přidávat během vývoje několik ukázkových událostí bych doporučoval vytvořit nový testovací kalendář a až tehdy, kdy bude aplikace odladěna, můžeme pouze zaměnit identifikátor. Pro získání identifikátoru si vybereme kalendář, který budeme chtít využít a přes ikonu tří vertikálních teček vybereme možnost Nastavení a sdílení.



Obr. 10 - Proklik na nastavení zvoleného kalendáře

Na stránce nastavení kalendáře se nachází mimo jiné i sekci Integrovat kalendář, kde nalezneme ID kalendáře, pomocí kterého kalendář následně budeme moci inicializovat v App Scriptu a předat ho obslužnému objektu CalendarApp, který nám s ním umožní manipulovat.

Tabulku, dokument i kalendář máme připravené a jejich identifikátory známe. Můžeme se tedy pustit do tvorby vlastního skriptu. Nejprve vytvoříme funkci processForm(), které předáváme vstupní hodnoty z formuláře. Jejím obsahem bude výpis vstupních polí s hodnotami ve formátu JSON do protokolů relace, ať máme vždy ponětí o tom, jaká data nám přišla na vstup. Hodnoty vstupních polí získáme z objektu formuláře jednoduše a to vyvoláním proměnné objektu nesoucí stejný název, jako vlastnost name vstupního pole. Následně předáme hodnoty v požadovaném podobě funkci createEnhancedEvent(), která již bude mít na starost samotnou logiku vytváření události a všech věcí k tomu spjatých, dle zadání. Zobrazení HTML na výstupu zde zůstává stejné, jako v předchozí ukázce pomocí doGet() funkce.

```
function processForm(formObject) {
  Logger.log(JSON.stringify(formObject));
  createEnhancedEvent(
    formObject.name,
    formObject.mail,
    formObject.event,
    new Date(formObject.date + " " + formObject.timeFrom),
    new Date(formObject.date + " " + formObject.timeTo)
  );
}
```

Nyní přejdeme k návrhu funkce createEnhancedEvent(). Její logiku si oddělíme do jednotlivých segmentů v závislosti na využívaném nástroji G Suite. Nejprve přidáme nový záznam do tabulky, následně vytvoříme pozvánku ze šablony a vyplníme ji pomocí vstupních dat. Soubor dále ve formátu PDF pošleme v příloze na email hosta a v posledním kroku vytvoříme samotnou událost v kalendáři, na kterou odešleme též hosta přizveme.

I v této ukázce budeme chtít formátovat datum, budeme k tomu opět potřebovat časové pásmo. V předchozí ukázce jsme ho získali pomocí objektu Intl. Jelikož zde máme však tentokrát k dispozici kalendář, který má časové pásmo uživatele zanesené ve vlastním nastavení, můžeme tuto hodnotu zapsat do další proměnné a jednoduše s ní pracovat. Ačkoliv bychom dle návrhu měli vytvořit událost jako poslední, inicializujeme si kalendář pro tyto účely hned na začátku. Začátek hlavních bloků si budeme pomocí Loggeru zapisovat pro přehled do protokolů pro účely snadnějšího ladění v případě chyby.

```
Logger.log('Inicializuji kalendář...');
var calendar = CalendarApp.getCalendarById(
  'dm4lim7srbh0de37vd2ikd028g@group.calendar.google.com'
);
```

Vestavěný objekt CalendarApp disponuje metodou getCalendarById(), který se nám pokusí na základě vstupního identifikátoru vrátit daný kalendář v podobě objektu Calendar. Důvod, proč inicializujeme kalendář v prvním kroku, je právě metoda getTimeZone(), kterou objekt Calendar disponuje.

V prvním kroku je třeba zapsat záznam o události do tabulky. Využijeme objektu SpreadsheetApp a pomocí metody openById() a identifikátoru získáme objektovou reprezentaci tabulky - objekt třídy Spreadsheet. Stejně jako v případě normální práce s tabulkou musíme vybrat aktivní list. Název jsme ponechali výchozí, tedy na objektu Spreadsheet zavoláme metodu getSheetByName() a předáme jí název “List 1”. Výstupem této metody je objekt třídy Sheet, na kterém už můžeme provádět úpravy obsahu. V našem případě budeme chtít zapsat nový řádek, k čemuž využijeme metody appendRow(). Na vstup této metody přijde pole s položkami seřazenými stejně, jako jsme rozvrhli sloupce tabulky.

```
Logger.log('Vkládám data do tabulky...');
var spreadsheet = SpreadsheetApp.openById(
  '1hHQiiOmRhdNg8jNRBvor2584-QVyDXkTAyHvPbcNFcw'
);
var sheet = spreadsheet.getSheetByName('List 1');
sheet.appendRow([
  name,
  mail,
  eventName,
  Utilities.formatDate(
    from, calendar.getTimeZone(), "dd.MM.yyyy HH:mm"
  ),
  Utilities.formatDate(
    to, calendar.getTimeZone(), "dd.MM.yyyy HH:mm"
  )
]);
Logger.log('Data úspěšně vložena do tabulky!');
```

Dalším krokem bude práce s dokumentem. Přístup zde však nebude tak jednoduchý jako v případě tabulky, jelikož než-li budeme moci se souborem pracovat, budeme muset nejprve vytvořit jeho kopii. Chceme-li manipulovat se soubory bez ohledu na jejich typ či vlastní obsah, jedná se o úkony na úrovni Google Disku, tudíš pro práci s tímto nástrojem využijeme zástupného objektu DriveApp. Nejprve pomocí metody getFileById() a identifikátoru získáme požadovaný soubor reprezentovaný objektem File. Vytvoření kopie nám umožní metoda makeCopy() tohoto objektu, která v případě úspěšného vykonání předá na svém výstupu nový objekt stejné třídy reprezentující nově vytvořenou kopii souboru.

Dalším logickým krokem by bylo s tímto objektem dále pracovat a nahradit v dokumentu zástupné znaky na základě předaných vstupních hodnot formuláře. Zde však nastává problém, jelikož jak jsem již zmínil, DriveApp pracuje se soubory bez ohledu na jejich typ a obsah a ztrácíme tím tak možnost využít specifických metod, jimiž zástupné objekty jednotlivých typů služeb disponují. Identifikátor souboru však zůstává ve všech případech vždy stejný a tak na objekt File zavoláme metodu getId() pro jeho získání.

Pracujeme-li s dokumenty využijeme služby DocumentApp. Obdobně jako v případě SpreadsheetApp zde pro získání souboru dle identifikátoru existuje metoda openById(), díky které získáme objekt Document, se kterým již můžeme pracovat za účelem úpravy obsahu. Způsobů, jakým uchopit část této logiky může opět existovat více.

Využil jsem zde prázdného objektu, přičemž jména jeho vlastních proměnných nepřímo odpovídají zástupným proměnným v šabloně. Těmto proměnným jsou připsány adekvátní hodnoty a následně je všechny projdeme pomocí for cyklu. V každém průběhu cyklu nejprve upravíme název tak, aby odpovídal formátu zápisu zástupných proměnných. Připojíme tedy z obou stran dvě složené závorky a název pomocí univerzální metody toUpperCase() převedeme do kapitálek. Tato hodnota bude sloužit jako první vstup funkce replaceText(), která se pokusí v těle souboru nalézt odpovídající záznam a přepsat ho hodnotou specifikovanou na druhém vstupu.

Názvy jednotlivých proměnných objektu bychom rovnou mohli uvádět v kapitálkách, ale z důvodu udržení konzistentní konvence psaní kódu jsem se rozhodl pro toto řešení. V posledním kroku ještě zavoláme na objektu Document metodu saveAndClose(). Ta není nutně vyžadována, ale bez jejího užití může docházet k chybě, kdy při následném vypsání obsahu dokumentu získáme text nevyplněné šablony, protože ještě nedošlo k zapsání provedených změn.

```
Logger.log('Vytvářím kopii dokumentu se šablonou...');
var copyID = DriveApp.getFileById(
  '16tOGJdgdk2EdqAQK9P2WVVUU3Fywn869idMHZ3xVoLc'
).makeCopy('Pozvánka').getId();
var document = DocumentApp.openById(copyID);
Logger.log('Kopii dokumentu úspěšně vytvořena s ID: ' + copyID);
var object = {
  name: name,
  event: eventName,
  from: Utilities
    .formatDate(from, calendar.getTimeZone(), "dd.MM.yyyy HH:mm"),
  to: Utilities
    .formatDate(to, calendar.getTimeZone(), "dd.MM.yyyy HH:mm")
};
Logger.log('Vyplňuji zástupné proměnné v kopii...');
for (var property in object) {
  Logger.log(
    '{{' + property.toUpperCase() + '}} => '+ object[property]
  );
  document.getBody().replaceText(
    '{{' + property.toUpperCase() + '}}',
    object[property]
  );
}
Logger.log('Ukládám vyplněný dokument...');
document.saveAndClose();
Logger.log('Dokument úspěšně uložen!');
```

Předposledním krokem bude odeslání emailu s přílohou obsahující vyplněný dokument převedený do univerzálního formátu PDF. Oproti ostatním službám zde nemusíme předávat žádný vstup pro získání emailu, jelikož se automaticky inicializuje email přihlášeného uživatele spouštějící skript. Abychom mohli email odeslat, potřebujeme specifikovat emailovou adresu příjemce, předmět, obsah a námi požadovanou přílohu. Pro účely demonstrace nám jako obsah poslouží též vlastní text dokumentu, abychom si ukázali i tento způsob řešení. Jelikož jsme však v předchozím úseku zavolali nad dokumentem metodu saveAndClose(), budeme nejprve potřebovat dokument znovu otevřít. Abychom získali jeho podobu ve formě PDF, budeme muset jít o úroveň výše a opět využít možností objektu DriveApp. Disponuje totiž metodou getAs(), jež po zadání korektního vstupu dokument předá na výstup v jiném, než jeho původním formátu. Chceme-li tedy dokument na výstup získat ve formátu PDF, předáme na vstup této funkce "application/pdf".

MailApp i jiné třídy disponují několika stejnými metodami, které se liší svým vstupem a v závislosti na stylu použití se tak programátor může rozhodnout, kterou z nich využije. Jelikož budeme pracovat s přílohou, museli bychom stejně využít objektu, a proto jsem volil řešení, kdy se metodě předají všechny vstupní hodnoty rovnou ve formě objektu strukturovaného dle oficiální dokumentace.

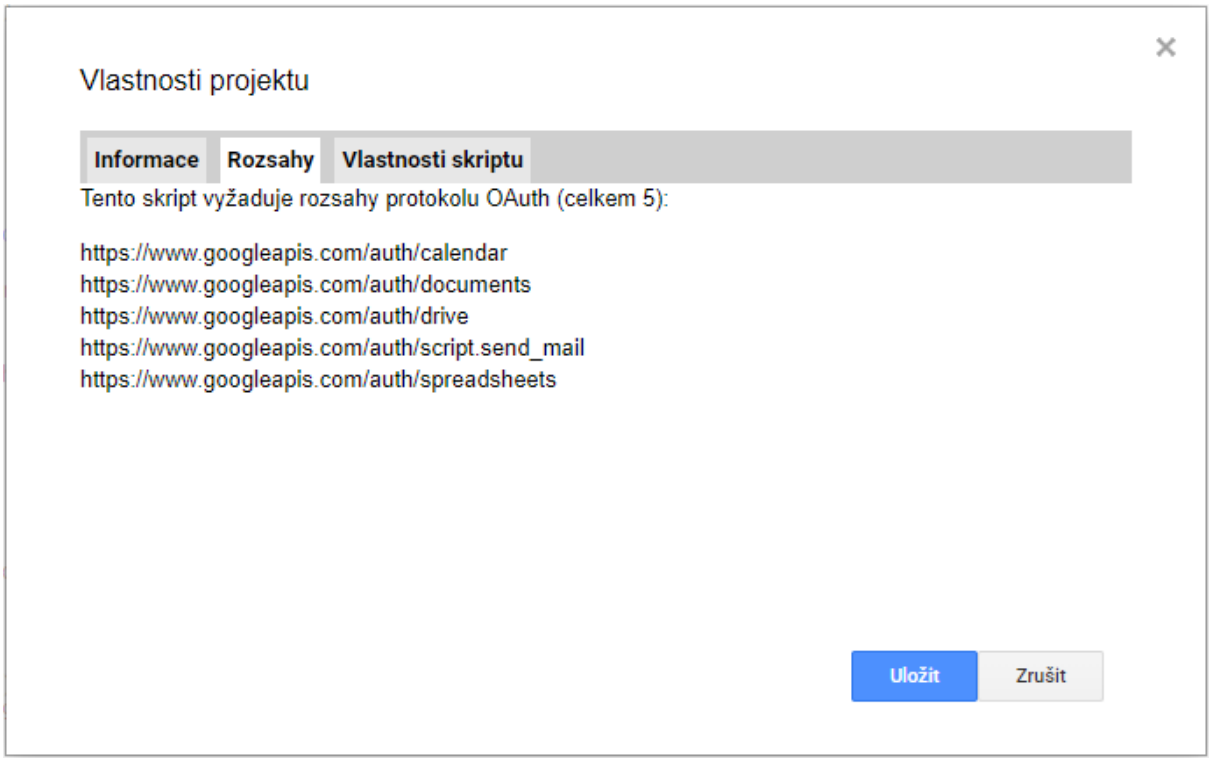
```
Logger.log('Odesílám email...');
MailApp.sendEmail({
  to: mail,
  subject: "Pozvánka",
  htmlBody: DocumentApp.openById(copyID).getBody().getText(),
  attachments:
[DriveApp.getFileById(copyID).getAs('application/pdf')]
});
Logger.log('Email odeslán!');
```

Pozn.: MailApp nepracuje se službou Gmail, ale je určena pouze k využívání za účelem odeslání emailu. Chtěli-li bychom pracovat se službou Gmail, využili bychom třídy GmailApp.

Nyní již zbývá finální část, ve které vytvoříme událost v kalendáři. Jelikož jsme kalendář inicializovali již na začátku, tento krok již znovu provádět nemusíme a rovnou zavoláme nad tímto objektem metodu createEvent(). Jelikož na událost budeme chtít přizvat další účastníky a zároveň jim odeslat upozornění na email, využijeme funkce s rozšířeným vstupem o parametr options. Následně jen předáme na vstup funkce všechny parametry a dle dokumentace přidáme objekt options s emailovou adresou hosta a povolíme parametr sendInvites() pro odesílání pozvánek.

```
Logger.log('Vytvářím kalendářovou událost...');
var event = calendar.createEvent(
  eventName,
  from,
  to,
  {
    guests: mail,
    sendInvites: true
  }
);
Logger.log(
  'Kalendářová událost úspěšně vytvořena s ID: ' + event.getId()
);
```

Máme-li celou logiku navrženou, můžeme projekt opět zprovoznit jako webovou aplikace. Ještě předtím, než tak učiníme, nahlédneme do vlastností projektu. V menu zvolíme záložku Soubor a následně vybereme položku Vlastnosti projektu. V novém přehledovém okně nás zajímá karta Rozsahy. Zde nalezneme všechna oprávnění, jež bude projekt vyžadovat ke své činnosti. Z návrhu víme, s jakými službami budeme pracovat a můžeme si tak jednoduše ověřit, zda-li všechny záznamy sedí. Navíc budeme mít službu DriveApp. Při odesílání emailu nevyužijeme služby GmailApp, jak by někteří mohli očekávat, ale MailApp, která je vytvořena pouze za účelem odesílání emailů, přičemž nevyžadujeme jiných služeb v rámci Gmailu.



Obr. 11 - Přehled jednotlivých oprávnění projektu

Rozhodneme-li se tedy takovýto projekt publikovat jako webovou aplikaci, budeme narozdíl od předchozí ukázky vyzvání ke kontrole oprávnění, kterých projekt ke své práci vyžaduje. V prvním kroku zvolíme účet, pod kterým budeme aplikaci spouštět. V tento moment se jedná o dosud neověřenou aplikaci společností Google, vyskočí na nás výstraha o této skutečnosti. Jelikož jsme však tvůrci aplikace a víme, jakým způsobem operuje a co vše dělá, klikneme na Rozšířené možnosti a následně na položku Přejít na web Eventor (nebezpečné). Google tímto způsobem chrání své neznalé uživatele, kteří například našli aplikaci třetí strany, která může kromě své hlavní nabízené funkcionality disponovat skrytými funkcemi, jež mohou sbírat uživatelská data bez jejich vědomí, či nějakým jiným způsobem uškodit. Zde se dostáváme na přehled, vypisující všechna oprávnění, o jejichž udělení aplikace žádá, přičemž si zde můžeme povšimnout analogie k výše zmíněným rozsahům. Kliknutím na tlačítko Povolit tato oprávnění projektu povolíme a ten je tak nyní nasazen ve formě webové aplikace.

Abychom si ověřili funkčnost aplikace, vyplníme testovací data a zkusíme událost uložit. Celý proces můžeme sledovat v protokolech spuštěné relace.

Projekt	Funkce	Typ	Čas zahájení	Délka	Stav
Eventor	processForm	Webová aplikace	15. 4. 2020 15:22:47	8.774 s	Dokončeno

Protokoly služby Stackdriver

```
15. 4. 2020 15:22:47    Informace  {"mail":"joh.doe@gmail.com","timeTo":"18:00","timeFrom":"08:00","name":"John Doe","event":"Testovací událost","date":"2020-04-16"}
15. 4. 2020 15:22:47    Informace  Inicializuji kalendář...
15. 4. 2020 15:22:47    Informace  Vkládám data do tabulky...
15. 4. 2020 15:22:48    Informace  Data úspěšně vložena do tabulky!
15. 4. 2020 15:22:48    Informace  Vytvářím kopii dokumentu se šablonou...
15. 4. 2020 15:22:53    Informace  Kopii dokumentu úspěšně vytvořena s ID: 1-iHpNFNVNZsAHZAnCqPpWKCdreOX-Y_qFdeZKiMA-G0
15. 4. 2020 15:22:53    Informace  Vyplňuji zástupné proměnné v kopii...
15. 4. 2020 15:22:53    Informace  Nahrazuji {{NAME}} za John Doe
15. 4. 2020 15:22:53    Informace  Nahrazuji {{EVENT}} za Testovací událost
15. 4. 2020 15:22:53    Informace  Nahrazuji {{FROM}} za 16.04.2020 08:00
15. 4. 2020 15:22:53    Informace  Nahrazuji {{TO}} za 16.04.2020 18:00
15. 4. 2020 15:22:53    Informace  Ukládám vyplněný dokument...
15. 4. 2020 15:22:53    Informace  Dokument úspěšně uložen!
15. 4. 2020 15:22:53    Informace  Odesílám email...
15. 4. 2020 15:22:55    Informace  Email odeslán!
15. 4. 2020 15:22:55    Informace  Vytvářím kalendářovou událost...
15. 4. 2020 15:22:56    Informace  Kalendářová událost úspěšně vytvořena s ID: d90qin5an0bmgifj4cvkihpm4o@google.com
```

Obr. 12 - Protokoly testovací relace

Vidíme zde, že veškerá vstupní data byla funkcím předána, inicializace dílčích souborů a kalendáře proběhly v pořádku, zástupné znaky šablony byly v kopii nahrazeny za odpovídající hodnoty, email odeslán a událost vytvořena. Pro kontrolu si otevřeme tabulku se záznamy a vidíme, že zde přibyl nový řádek s adekvátními daty. Dále můžeme přejít na Gmail, kde v odeslané poště nalezneme email odeslaný na momentálně neexistující adresu john.doe@gmail.com, jež obsahuje vyplněný text ze šablony a v příloze nalezneme vygenerovaný PDF soubor. Otevřeme-li kalendář a přejdeme-li na zadané datum, nalezneme zde i událost ve zvoleném čase, kde po prokliku na ní vidíme v seznamu hostů nepotvrzenou účast uživatele se stejnou emailovou adresou. Přejdeme-li navíc do složky na disku, kde jsme vytvořili šablonu, vidíme, že zde nově přibyl soubor s názvem Pozvánka, s vyplněnými daty.

Jelikož máme veškerá data zaznamenáváme do tabulky, nemusíme nutně požadovat, aby soubory jednotlivých pozvánek na disku zůstávaly. Za tímto účelem můžeme skript obohatit o další část kódu, který se o tuto problematiku postará. Jak je nám již známo, při práci se soubory jako takových využijeme služby DriveApp. Ačkoliv bychom metodu na odstranění souboru očekávali zde, není tomu tak, jelikož mazání souborů může provést objektová reprezentace složky, která je těmto souborům nadřazena. Pomocí metody getParents() nad souborem získáme všechny jeho nadřazené uzly a následným zřetězením metody next získáme nejbližší nadřazenou složku, nad kterou pak vyvoláním metody removeFile() soubor odstraníme.[13]

```
var file = DriveApp.getFileById(copyID);
var folder = DriveApp.getRootFolder();
if (file.getParents().hasNext()) {
  folder = file.getParents().next();
}
folder.removeFile(file);
```

Správně bychom si nejprve pomocí metody hasNext() měli ověřit, zda-li vůbec existuje nadřazený uzel, jelikož nacházíme-li se na úrovni kořenu, dále už jít nejde. Navíc jsme si na konci předchozí kapitoly uvedli příklad vlastního error handleru, který bychom mohli nasadit ve funkci processForm(), jež by místo voidu mohla nyní vracet boolean indikující, zda-li proces v pořádku proběhl až dokonce, či někde nastala chyba. Přidáme tedy výše uvedenou funkci errorHandler() a funkci upravíme do následující podoby.

```
function processForm(formObject) {
  Logger.log(JSON.stringify(formObject));
  try {
    createEnhancedEvent(
      formObject.name,
      formObject.mail,
      formObject.event,
      new Date(formObject.date + " " + formObject.timeFrom),
      new Date(formObject.date + " " + formObject.timeTo)
    );
    return true;
  } catch(e) {
    errorHandler(e);
    return false;
  }
}
```

Pouze v rámci tohoto projektu se nám nabízí hned několik dalších možností na úpravy, zdokonalení a obohacení o další funkcionalitu. Takovým příkladem by dále mohly být validační funkce na straně šablony zabraňující zadání prázdného či nevalidního vstupu, uchopení výstupu funkce processForm() a následné vypsání informace uživateli o výsledném stavu spuštěného skriptu a určitě by se našlo mnoho dalších.

Hlavním účelem této kapitoly však bylo představení některých z dostupných objektů pro práci s nástroji platformy G Suite. Doposud jsme se však věnovali tvorbě vlastních webových aplikací, místo obohacení funkcionality některé ze služeb existujících. Mimo jiné jsme také prozatím nevyužili ani žádné ze služeb nějaké třetí strany. V následující kapitole se tedy zaměříme na jeden specifický nástroj, nad kterým vytvoříme vázaný skript, abychom tento nástroj obohatili o nové možnosti práce s ním.

8 Tabulka, vázaný skript a služby třetích stran

Chceme-li některý z existujících nástrojů platformy G Suite obohatit o specifickou funkcionalitu za účelem vytvoření jakési hybridní aplikace, jednou z nejlepších možností je využití tabulkového kalkulátoru, který již sám o sobě disponuje širokou nabídkou vestavěných funkcí.

Výstupem projektu tentokrát nebude rozhraní webové aplikace, ale vytvoření logiky nad tabulkou za účelem generování faktur na základě spjatých záznamů v tabulce. Využijeme zde služeb Administrativního registru ekonomických subjektů (dále jen jako ARES), což je informační systém Ministerstva financí České republiky umožňující vyhledávání informací o subjektu dle jeho identifikačního čísla. Pomocí této služby tedy získáme fakturační údaje subjektu, pro který budeme fakturu vystavovat. Jelikož žijeme v moderní době, kde již převažují platby elektronické, vygenerujeme si i QR kód s údaji o platbě pomocí nástroje QR Platba. Stejně jako v předchozím projektu si i zde vytvoříme šablonu, do které budeme finálně parsovat předané hodnoty a získáme tak soubor faktury. Po vygenerování faktury vytvoříme koncept mailu pod naším Google účtem k odeslání zprávy s fakturou zákazníkovi a následně vytvoříme v kalendáři událost pro ověření přijetí platby v datum splatnosti. Náš skript se bude volat vždy po provedení úprav v tabulce, kdy zkontrolujeme rozsah provedených změn a v případě, kdy bude odpovídat vymezeným sloupcům a budou splněny všechny spjaté podmínky, vyvoláme požadovanou obslužnou funkci. Za tímto účelem využijeme možností spouštěčů, o kterých jsme již hovořili, ale názorně jsme si je v praktické ukázce ještě neměli možnost představit.

Pro návrh samotné tabulky je třeba si nejprve logicky rozdělit fakturu do jednotlivých částí. Máme zde naše fakturační údaje, které zůstávají statické a mohou tak být v šabloně již předvyplněny. Následují fakturační údaje odběratele, které se pro každý subjekt budou měnit a je tedy třeba je zakomponovat mezi vstupní data tabulky. K těmto údajům ještě následně přibývají parametry, jako identifikační číslo faktury (možnost využití ve formě variabilního symbolu), datum vystavení a splatnosti. Vlastní obsah pak už tvoří jednotlivé záznamy položek s množstvím, cenou za měrnou jednotku a cenou za uvedený počet jednotek.

Z toho nám vyplývá, že máme-li tři oddělené logické celky na sobě závisejících, budeme je mít oddělené též v tabulce jakožto tři samostatné listy. Každý list bude reprezentovat jednu tuto logickou část, tedy bude existovat list klientů, list fakturačních položek a hlavní list faktur.

Celou problematiku si můžeme představit v podobě databáze, jelikož budeme potřebovat tyto listy mezi sebou propojit odpovídajícími vztahy. Jedna faktura má vždy právě jednoho odběratele, tudíž odběratel bude cizím klíčem v tabulce faktur. Tento vztah reprezentujeme v tabulce seznamem, jehož obsah bude brán z listu klientů, například v podobě názvu subjektu, dle kterého následně propojíme fakturu se záznamem na listu klientů za účelem získání fakturačních údajů. Naopak fakturačních položek bude pod jednou fakturou zpravidla více a proto zde volíme opačný postup, kdy záznam ponese cizí klíč odkazující na list faktur v podobě identifikátoru faktury. Obou těchto seznamů můžeme jednoduše dosáhnout využitím nástroje Ověření dat pro buňky daného sloupce.

Než-li se pustíme do tvorby skriptu, budeme potřebovat nejprve vytvořit tabulku a předpřipravit si její strukturu. Navíc si ještě rovnou vytvoříme i soubor se šablonou faktury, ať se k tomuto pak nemusíme zpětně vracet.

Jak jsme si již řekli, tabulka bude obsahovat tři základní listy – list klientů, faktur a jejich položek.

	A	B	C	D	E	F	G	H	I
1	Nahrát	Název	Mail	Ulice	Město	PSČ	Země	IČO	DIČ
2	☐								
3	☐								
4	☐								

Obr. 13 - List klientů

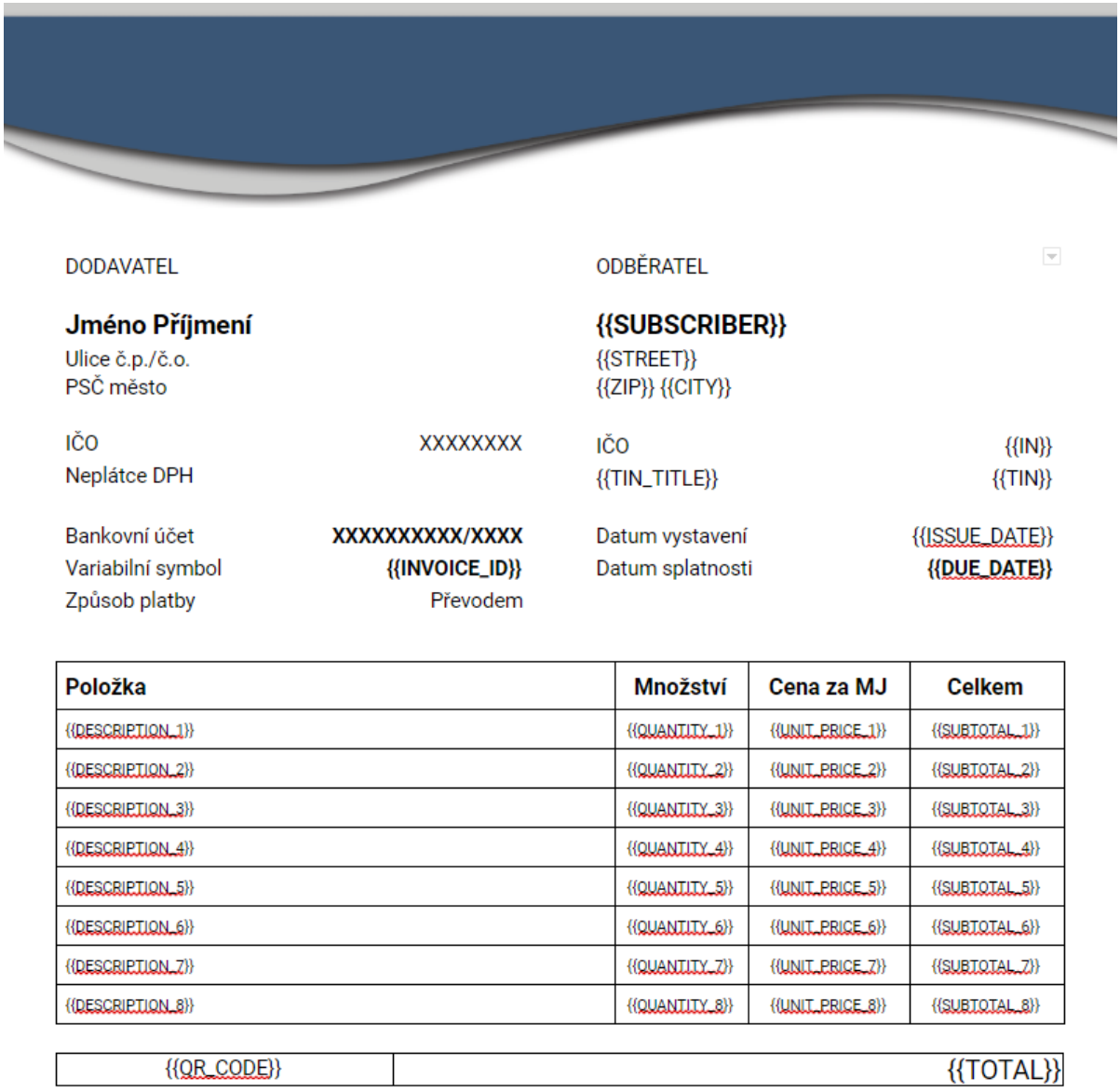
	A	B	C	D	E	F	G
1	Vystavit	Odběratel	Zakázka	ID	Datum vystavení	Datum splatnosti	Popis
2	☐						
3	☐						
4	☐						

Obr. 14 - List faktur

	A	B	C	D	E	F
1	ID faktury	Popis	Množství	Měrná jednotka	Cena za jednotku	Celkem
2						0 Kč
3						0 Kč
4						0 Kč

Obr. 15 - List položek

Vzhled a rozložení jednotlivých prvků šablony pak záleží na potřebách jednotlivců. Pro účely této demonstrace jsem vytvořil jednoduchou fakturu se standardním rozložením prvků a aby se nejednalo čistě jen o text, vložil jsem do dokumentu i jednoduchý design záhlaví.



DODAVATEL

Jméno Příjmení
Ulice č.p./č.o.
PSČ město

IČO XXXXXXXX
Neplátce DPH

Bankovní účet **XXXXXXXXXX/XXXX**
Variabilní symbol **{{INVOICE_ID}}**
Způsob platby Převodem

ODBĚRATEL

{{SUBSCRIBER}}
{{STREET}}
{{ZIP}} {{CITY}}

IČO {{IN}}
{{TIN_TITLE}} {{TIN}}

Datum vystavení {{ISSUE_DATE}}
Datum splatnosti **{{DUE_DATE}}**

Položka	Množství	Cena za MJ	Celkem
{{DESCRIPTION_1}}	{{QUANTITY_1}}	{{UNIT_PRICE_1}}	{{SUBTOTAL_1}}
{{DESCRIPTION_2}}	{{QUANTITY_2}}	{{UNIT_PRICE_2}}	{{SUBTOTAL_2}}
{{DESCRIPTION_3}}	{{QUANTITY_3}}	{{UNIT_PRICE_3}}	{{SUBTOTAL_3}}
{{DESCRIPTION_4}}	{{QUANTITY_4}}	{{UNIT_PRICE_4}}	{{SUBTOTAL_4}}
{{DESCRIPTION_5}}	{{QUANTITY_5}}	{{UNIT_PRICE_5}}	{{SUBTOTAL_5}}
{{DESCRIPTION_6}}	{{QUANTITY_6}}	{{UNIT_PRICE_6}}	{{SUBTOTAL_6}}
{{DESCRIPTION_7}}	{{QUANTITY_7}}	{{UNIT_PRICE_7}}	{{SUBTOTAL_7}}
{{DESCRIPTION_8}}	{{QUANTITY_8}}	{{UNIT_PRICE_8}}	{{SUBTOTAL_8}}

{{QR_CODE}}	{{TOTAL}}

Obr. 16 - Šablona faktury

Nyní můžeme již přejít k tvorbě vázaného skriptu k tabulce. Přes záložku v menu Nástroje se po kliknutí na položku Editor skriptu dostaneme do rozhraní editoru skriptu, kde můžeme začít vytvářet vlastní logiku celé aplikace.

Pracujeme-li s tabulkou, velmi často budeme pracovat s různými rozsahy řádků, ale převážně také i sloupců. Proto je i pro budoucí úpravy struktury tabulky lepší, pokud nebudeme přímo uvádět na vstupu takových funkcí vlastní pozice sloupců, ale vytvoříme si zástupné proměnné, ve kterých jejich pozice specifikujeme. Pokud bychom totiž v budoucnu přidali, odstranili či jen přesunuli některý ze sloupců, museli bychom tak na všech místech, pracujících s takovým rozsahem, aktualizovat pozici těchto sloupců. Máme-li však pozice reprezentovány určitou proměnnou, stačí pak pouze změnit její hodnotu a tato změna se propíše na všechna místa, kde se tato proměnná volá.

Vytvoříme si za tímto účelem nový soubor skriptu, který si nazveme constants. V tomto souboru definujeme veškeré konstantní proměnné, se kterými budeme pracovat. Mimo pozice jednotlivých sloupců si také vytvoříme proměnné zastupující názvy jednotlivých listů, rozhodneme-li se je též v budoucnu změnit.

Definice konstant pro list se záznamy faktur:

```
const invoicesSheet = 'Faktury';
const invoiceColumns = {
  generate: 1,
  subscriber: 2,
  project: 3,
  ID: 4,
  issueDate: 5,
  dueDate: 6,
  description: 7
};
```

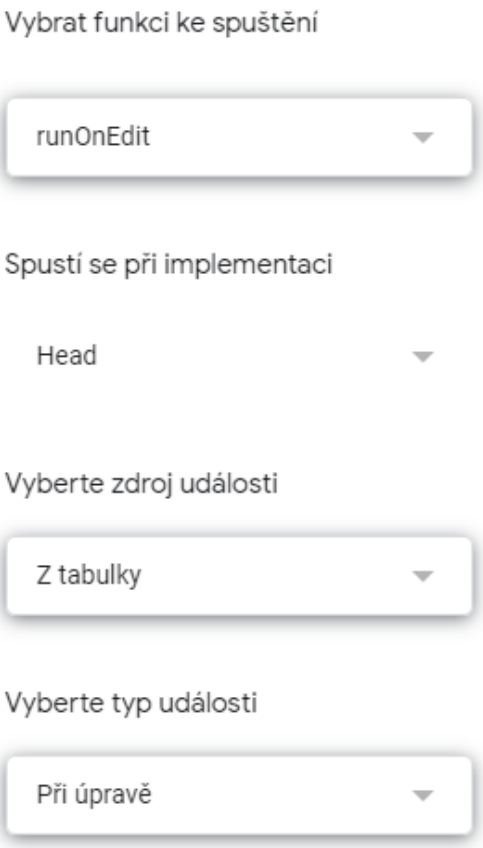
Definice konstant pro list se záznamy klientů:

```
const clientsSheet = 'Klienti';
const clientColumns = {
  load: 1,
  name: 2,
  mail: 3,
  street: 4,
  city: 5,
  ZIP: 6,
  country: 7,
  IN: 8,
  TIN: 9
};
```

Definice konstant pro list se záznamy položek faktury:

```
const itemsSheet = 'Položky';
const itemColumns = {
  invoiceID: 1,
  description: 2,
  quantity: 3,
  units: 4,
  unitPrice: 5,
  total: 6
};
```

Oproti předchozím ukázkám se zde skript sám automaticky spustí v závislosti na provedené změně v tabulce, kterou jednotlivé odposlouchávací funkce vyhodnotí. Odpovídá-li tato změna nastaveným parametrům, vyvolá se finální obslužná funkce. Za tímto účelem však nejprve musíme vytvořit spouštěč a funkci, jež bude v adekvátních případech vyvolána. Takový spouštěč bude mít následující nastavení:



Obr. 17 - Vlastní spouštěč vyvolávající funkci runOnEdit při úpravě obsahu tabulky

Funkce runOnEdit() bude obsahovat vstupní parametr e, jež obsahuje informace o provedené změně v tabulce. Tento parametr bude následně předáván takzvaným odposlouchávacím funkcím, jež budou dále vyhodnocovat, zda-li se daná změna týká jejich segmentu.

```
function runOnEdit(e) {
  loadInfoEventListener(e);
  generateInvoiceEventListener(e);
}
```

V tomto případě bude při změně vždy vyvolána pouze jedna takováto obslužná funkce a proto odposlouchávacím funkcím přiřadíme návratovou hodnotu v podobě booleanu. Je-li tato návratová hodnota pravdivá, ukončíme hlavní funkci, jelikož víme, že událost byla již úspěšně odchycena a není třeba vyvolávat další odposlouchávací funkci. Tato úprava značně ulehčí náročnosti vykonávání skriptu, jelikož se nebude muset provádět další proces ověřování, je-li jasné, že žádná další odposlouchávací funkce tuto událost pro svůj segment nevyhodnotí.

```
function runOnEdit(e) {
  if (loadInfoEventListener(e)) return;
  if (generateInvoiceEventListener(e)) return;
}
```

Začněme nejprve segmentem pro nahrávání dat z ARES a to odposlouchávací funkcí. Ze struktury tabulky vyplývá, že zaškrtnutím checkboxu budeme chtít tuto funkci vyvolat. Na úrovni rozsahu provedené změny pak musíme vyhodnotit, zda-li byla změna provedena na listu Klienti, přesněji zda-li byl zaškrtnut checkbox ve sloupci Nahrát. Funkce však pro svou činnost potřebuje i identifikační číslo subjektu a tak bude třeba ověřit, zda-li je v aktivním řádku vyplněn sloupec IČO. Jsou-li všechna tato kritéria splněna, vyvoláme obslužnou funkci loadInfo s předáním potřebných parametrů.

```
function loadInfoEventListener(e) {
  var sheet = e.source.getActiveSheet();
  var watchColumn = clientColumns.load;
  var watchSheet = clientsSheet;
  var activeRow = e.range.getRow();
  var icoCell = sheet.getRange(activeRow, clientColumns.IN);
  var checkboxCell = sheet.getRange(activeRow, watchColumn);

  if (
    sheet.getName() == watchSheet
    && e.range.columnStart == watchColumn
    && checkboxCell.isChecked()
    && !icoCell.isBlank()
  ) {
    loadInfo(sheet, activeRow, icoCell.getValue());
    return true;
  }

  return false;
}
```

Funkce loadInfo vyplní získaná data z ARES do tabulky, přičemž ještě změnou barvy řádku uživateli znázorníme, že funkce probíhá. K tomu, abychom však získali potřebných dat, si vytvoříme ještě doprovodnou funkci getInfoByIN(), která data z ARES získá. Budeme-li provádět jakékoliv volání na externí URL, ze které budeme chtít získat data, využijeme k této akci služby UrlFetchApp, která disponuje metodou fetch. Dle dokumentace ARES víme, že volané URL musí obsahovat GET parameter "ico". Výsledná podoba bude následující:

https://wwwinfo.mfcr.cz/cgi-bin/ares/darv_bas.cgi?ico=XXXXXXXX

Výstupem této funkce je HTTP odpověď reprezentovaná objektem HTTPResponse, přičemž pro získání dat z odpovědi je ještě třeba vyvolat na tomto objektu metodu getContentText().

Během studie dokumentace služby ARES jsem se snažil najít odpověď na otázku, zda-li by šlo odpověď reprezentovat ve formátu JSON namísto formátu XML, ve kterém odpověď přicházela. Na tuto otázku jsem záhy našel následující odpověď:

"Když člověk překonal všechny překážky, stál před poměrně komplexním XML souborem. Respektive milionem takových souborů. Pro omladinu, zhýčkanou JSON, CSV a jinými formáty, to byl často boj." [14]

Celý tento úsek byl o to těžší, jelikož s formátem XML v této podobě a takového rozsahu jsem se ještě nikdy nesetkal. Celý vývoj této aplikace se právě v tomto místě nejvíce pozastavil, jelikož jsem nebyl schopný data uchopit a pracovat s nimi, ačkoliv jsem postupoval dle všech dostupných dokumentací. Klíčem celého problému bylo nevyužití namespaces, neboli jmenných prostor, kterých bylo třeba u funkce getChild() nad XML dokumentem. Nakonec jsem tedy tuto chybu objevil a vytvořil jsem si objekt obsahující všechny jmenné prostory obsažené v dokumentu a odpovídající jmenný prostor jsem předal na vstup příslušné funkce getChild().

Abych si však přece jen ulehčil práci a vzdálil se od tohoto formátu, vytvořil jsem si pomocnou funkci jsonize(), která XML převede do obyčejného JSON objektu, se kterým se mi pracuje lépe. V tento moment tak vytvářím nový soubor skriptu “helpers.gs”, který obsahuje všechny vlastní pomocné funkce.

```
function jsonize(el) {
  var json = {};
  for (var child in el) {
    json[el[child].getName()] = el[child];
  }
  return json;
}
```

V prvním kroku inicializuji strukturu výsledného objektu na výstupu funkce. Následně se dotáži služby ARES o data daného subjektu specifikovaného pomocí identifikačního čísla. Z odpovědi získám strukturované XML a inicializuji si též všechny jeho jmenné prostory. Následně získám potomka, s jehož obsahem chci pracovat a tato data převedu do formátu JSON a předám je jako hodnotu dočasné proměnné temp.

```
Logger.log("Fetching content via API...");
var result = {
  name: '', street: '', city: '', zip: '', country: '', TIN: ''
};
var response = UrlFetchApp.fetch(
  "https://wwwinfo.mfcr.cz/cgi-bin/ares/darv_bas.cgi?ico=" + ico
);
var content = response.getContentText();
var aresNS = "http://wwwinfo.mfcr.cz/ares/xml_doc/schemas/";
  var namespaces = {
    are: XmlService.getNamespace(
      aresNS + "ares/ares_answer_basic/v_1.0.3"
    ),
    D: XmlService.getNamespace(
      aresNS + "ares/ares_datatypes/v_1.0.3"
    ),
    U: XmlService.getNamespace(
      aresNS + "uvis_datatypes/v_1.0.3"
    ),
    xsi: XmlService.getNamespace(
      "http://www.w3.org/2001/XMLSchema-instance"
    )
}
var document = XmlService.parse(content);
var root = document.getRootElement();
var data = root.getChildren()[0].getChild('VBAS',
namespaces.D).getChildren();
var temp = jsonize(data);
Logger.log(temp);
```

```
result.name = temp.OF.getValue();
if (temp.DIC != null) {
  result.TIN = temp.DIC.getValue();
}
var temp = jsonize(temp.AA.getChildren());
Logger.log(temp);
result.country = temp.NS.getValue();
if (temp.NU == null) {
  temp.NU = temp.NCO;
  temp.N = temp.NOK;
  temp.NCO = null
}
result.street = temp.NU.getValue() + " " + temp.CD.getValue();
if (temp.CO != null) {
  result.street += "/" + temp.CO.getValue();
}
result.city = temp.N.getValue();
if (temp.NCO != null
  && temp.N.getValue() != temp.NCO.getValue()
) {
  result.city += " - " + temp.NCO.getValue();
}
result.zip = temp.PSC.getValue();
return result;
```

Máme-li data z ARES inicializovaná, stačí odpovídající hodnoty v požadovaném formátu přiřadit parametrům výstupního objektu, ze kterého je následně budeme zapisovat do jednotlivých buněk daného řádku. Veškeré podmínky mají za účel formátovat dané hodnoty, je-li některý z dalších parametrů přítomný, či naopak není. Vyjma samotného zápisu dat navíc ještě zbarvíme pozadí řádku, abychom indikovali stav. Žlutá barva značí zpracovávání, červená chybu a zelená úspěšné vložení dat.

```
function loadInfo(sheet, row, ico) {
  Logger.log("loadInfoEventListener triggered!");
  sheet.getRange(row,1,1,sheet.getLastColumn())
    .setBackgroundRGB(224, 224, 0);
  SpreadsheetApp.flush();

  try {
    var data = getInfoByIN(ico);

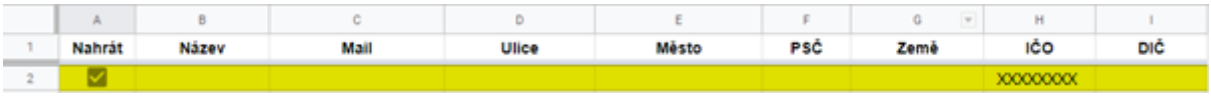
    sheet.getRange(row, clientColumns.name).setValue(data.name);
    sheet.getRange(row, clientColumns.street)
      .setValue(data.street);
    sheet.getRange(row, clientColumns.city).setValue(data.city);
    sheet.getRange(row, clientColumns.ZIP).setValue(data.zip);
    sheet.getRange(row, clientColumns.country)
      .setValue(data.country);
    sheet.getRange(row, clientColumns.TIN).setValue(data.TIN);

    sheet.getRange(row,1,1,sheet.getLastColumn())
      .setBackgroundRGB(102, 224, 102)
    sheet.getRange(row,1,1,sheet.getLastColumn())
      .setBackground(null);

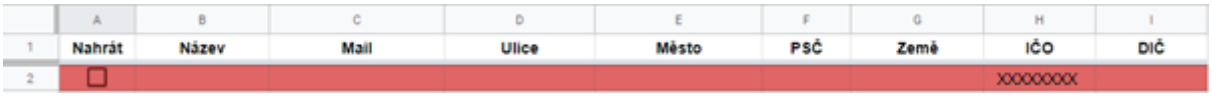
    SpreadsheetApp.flush();
  } catch (e) {
    sheet.getRange(row, clientColumns.load) setValue(false);
    sheet.getRange(row, 1, 1, sheet.getLastColumn())
      .setBackgroundRGB(224, 102, 102);
    SpreadsheetApp.flush();

    throw e;
  }
}
```

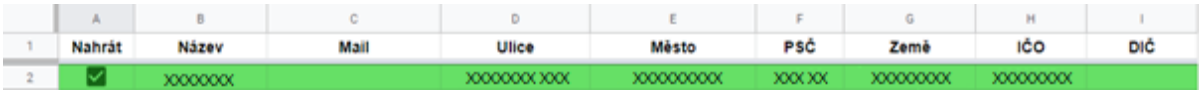
Abychom nově implementovanou funkcionalitu otestovali, zkusíme získat data k námi vybranému subjektu. Vložíme jeho IČO do odpovídající buňky a zaškrtneme na tom samém řádku checkbox Nahrát. Během pár vteřin by měl řádek změnit barvu pozadí na žlutou, což indikuje odchycení události a vyvolání odpovídající funkce.



Obr. 18 - Indikace spuštění funkce loadInfo



Obr. 19 - Indikace nastalé chyby



Obr. 20 - Ukázka zápisu úspěšně zinicializovaných dat z ARES

Máme-li údaje o subjektu, kterému budeme fakturu vystavovat, můžeme pokračovat s vytvořením logiky pro generování faktur. Vyjma samotného parsování dat do dokumentu, které bude stejné jako v předchozí ukázce, nyní budeme muset samotná data posbírat z jednotlivých buněk a správně k faktuře spárovat její položky a odběratele. Začněme tedy opět tvorbou odposlouchávací funkce, která ověří, zda-li uživatel v provedené změně zaškrtnul checkbox v prvním sloupci na listu Faktury.

```
function generateInvoiceEventListener(e) {
  var sheet = e.source.getActiveSheet();
  var watchColumn = invoiceColumns.generate;
  var watchSheet = invoicesSheet;
  var activeRow = e.range.getRow();
  var checkboxCell = sheet.getRange(activeRow, watchColumn);

  if (
    sheet.getName() == watchSheet
    && e.range.columnStart == watchColumn
    && checkboxCell.isChecked()
  ) {

    ...

    generateInvoice(sheet, activeRow, records);
    return true;
  }

  return false;
}
```

Takto bychom mohli jednoduše definovat, zda-li byla provedena požadovaná akce a následně vyvolat obslužnou funkci generateInvoice(). Oproti předchozímu kroku jsme zde však závislí na uživatelském vstupu ve formě vyplněním určitých buněk tabulky. Musíme tedy nejprve ověřit, zda-li jsou všechny požadované sloupce v aktivním řádku vyplněné a v případě, že některý z nich není, označíme takový sloupec červeným pozadím a obslužnou funkci nevyvoláme, dokud do těchto sloupců nedojde k zadání dat.

```
Logger.log("Checking invoice data cells...");
var checkCells = [
  invoiceColumns.subscriber,
  invoiceColumns.project,
  invoiceColumns.ID,
  invoiceColumns.issueDate,
  invoiceColumns.dueDate
];

checkCells = checkCells.filter(function(colulm, index) {
  if (sheet.getRange(activeRow, colulm).isBlank()) {
    sheet.getRange(activeRow, colulm)
      .setBackgroundRGB(224, 50, 50);
    return true;
  } else {
    sheet.getRange(activeRow, colulm).setBackground(null);
    return false;
  }
});

if (checkCells.length !== 0) {
  sheet.getRange(activeRow, invoiceColumns.generate)
    .setValue(false);
  SpreadsheetApp.flush();
  return true;
}
```

Ověření provedeme takovým způsobem, kdy vytvoříme pole checkCells obsahující pozice požadovaných sloupců a pomocí funkce filter nad tímto polem následně ověříme, zda-li odpovídající rozsah není prázdný. Výstupem bude pole, obsahující pozice všech sloupců, jež tyto požadavky nesplňují. Finálně tak můžeme jednoduše ověřit, zda-li velikost tohoto pole není nulová, kdy by takovém případě odškrtneme checkbox a vyvoláním metody SpreadsheetApp.flush() zajistíme, aby došlo ke všem provedeným grafickým úpravám tabulky, které mohou být občas z hlediska zlepšení výkonu zařazeny do společné fronty, než aby se prováděly jednotlivě.

Vyjma vlastních sloupců na listu Faktury je však třeba provést ještě další kontroly. Dle návrhu se k faktuře totiž dále vážou její položky pomocí klíče ID. Je tedy třeba zkontrolovat, zda-li existuje alespoň jeden záznam přiřazený k dané faktuře, jelikož nechceme generovat prázdnou fakturu bez záznamů. Krom toho je třeba stejně jako v případě sloupců na listu Faktury zkontrolovat, zda-li jsou u odpovídajících záznamů na listu Položky též vyplněné všechny požadované sloupce. Abychom získali propojené řádky, vytvoříme si nejprve pomocné funkce getRowsByCellValue() a getRowByCellValue() v případě, že bychom požadovali pouze jeden takový záznam.

```
function getRowsByCellValue(
  sheet,
  cellValue,
  searchedCellPosition,
  limit = -1
) {
  var dataRange = sheet.getDataRange();
  var values = dataRange.getValues();
  var rows = [];
  for (var i = 0; i < values.length; i++) {
    if (values[i][searchedCellPosition] == cellValue) {
      rows.push(
        sheet.getRange(i + 1, 1, 1, sheet.getLastColumn())
      );
      if (rows.length === limit) return rows;
    }
  }
  return rows;
}
```

Budeme-li chtít získat řádky z jiného listu v závislosti na hodnotě sloupce z aktivního listu, budeme muset vyhledávací funkci getRowsByCellValue() předat informace o listu, ve kterém má řádky hledat, pozici sloupce, jež má obsahovat danou hodnotu a následně onu zmíněnou hodnotu samotnou. V případě, kdybychom chtěli získat pouze první řádek rozšíříme vstup této funkce ještě o parametr limit, který defaultně nastavíme do hodnoty -1, přičemž velikost pole nikdy nemůže takovéto logicky dosáhnout. Nalezené řádky budeme následně vracet v podobě rozsahu, což nám v určitých případech umožní rozšířené možnosti manipulace se sloupci, než-li v případě, kdybychom natvrdo vraceli pouze pole hodnot seřazených dle sledu sloupců v tabulce.

V případě, kdy bychom požadovali na výstupu této funkce pouze jeden řádek, bychom jednoduše mohli na vstup předat limit o hodnotě jedna, ale jelikož takováto metoda může být užita při relacích 1:1 velmi často, vytvoříme si pro tyto účely dodatečně funkci getRowByCellValue(), která na svém vstupu nebude parametr limit vyžadovat, jelikož vždy předá na místo jeho vstupu u funkce getRowsByCellValue() hodnotu jedna a na výstupu u vráceného pole vyžádá jeho první položku a vrátí tak na svém výstupu rovnou rozsah.

```
function getRowByCellValue(
  sheet,
  cellValue,
  searchedCellPosition
) {
  return getRowsByCellValue(
    sheet,
    cellValue,
    searchedCellPosition,
    1
  )[0];
}
```

Máme-li pomocné funkce vytvořené, můžeme se vrátit zpět k ověřování dat u spjatých záznamů na cizím listu.

Nejprve ověříme, zda-li existuje alespoň jeden záznam spjatý k ID faktury na aktivním řádku:

```
Logger.log("Checking invoice records...");
var records = getRowsByCellValue(
  SpreadsheetApp.getActive().getSheetByName(itemsSheet),
  sheet.getRange(activeRow, invoiceColumns.ID).getValue(),
  itemColumns.invoiceID - 1
);
if (records.length === 0) {
  sheet.getRange(activeRow, invoiceColumns.generate)
    .setValue(false)
    .setBackgroundRGB(224, 50, 50);
  SpreadsheetApp.flush();
  return true;
}
```

Jelikož ověření, zda-li je obsah buňky prázdný, budeme využívat i na jiných místech, vytvoříme si pomocnou funkci isEmpty:

```
function isEmpty(checkVar) {
  return (
    checkVar === undefined
    || checkVar == null
    || checkVar.length <= 0
  );
}
```

Existuje-li jeden či více spjatých záznamů, ověříme jejich obsah a ujistíme se, že žádný z vyžadovaných sloupců není prázdný:

```
Logger.log("Checking invoice records data cells...");
var errors = records.filter(function(range) {
  var errors = [];
  errors = range.getValues()[0].filter(function(column, index) {
    var range = this.getSheet().getRange(
      this.getRow(),
      index + 1
    );
    range.setBackground(null);
    if (isEmpty(column)) {
      range.setBackgroundRGB(224, 50, 50);
      SpreadsheetApp.flush();
      return true;
    }
  }, range);
  if (errors.length !== 0) return true;
});
if (errors.length !== 0) {
  sheet.getRange(activeRow, invoiceColumns.generate)
    .setValue(false)
    .setBackgroundRGB(224, 50, 50);
  SpreadsheetApp.flush();
  return true;
}
```

Opět zde využijeme funkce filter, ale jelikož pracujeme s vícero řádky, kdy pomocí námi vytvořené funkce getRowsByCellValue() získáme jejich rozsahy, je třeba jednotlivě provést další volání funkce filter na hodnotách získaných z každého rozsahu. Abychom mohli uvnitř jednotlivých iterací hodnot sloupců užít jejich nadřazeného rozsahu, předáme si do kontextu funkce tento rozsah jako parametr this. V každé filtrovací smyčce pak na výstupu kontrolujeme, zda-li pole obsahuje nějaké záznamy. Pokud je toto pole prázdné, žádný ze sloupců nebyl prázdný a můžeme tak vyvolat samotnou generovací logiku s jistotou, že všechna potřebná data jsou připravena.

Vyjma zbarvení do žluta, čímž indikujeme, že obslužná funkce byla vyvolána, zapouzdříme její vlastní logiku do bloku try/catch a v případě zachycení výjimky zbarvíme řádek do červena, aby uživatel dostal o nastalé chybě informaci.

```
function generateInvoice(sheet, row, records) {
  Logger.log("generateInvoiceEventListener triggered!");
  sheet.getRange(row,1,1,sheet.getLastColumn())
    .setBackgroundRGB(224, 224, 0);
  SpreadsheetApp.flush();

  try {

    ...

  } catch (e) {
    sheet.getRange(row,1,1,sheet.getLastColumn())

      .setBackgroundRGB(224, 102, 102);
    SpreadsheetApp.flush();
    throw e;
  }
}
```

Nejprve inicializujeme proměnné pro sloupce, jejichž hodnoty využijeme na více místech a dle uvedeného odběratele na aktivním řádku listu Faktury vyhledáme daný záznam z listu Klienti.

```
var calendar = CalendarApp.getCalendarById(
  'dm4lim7srbh0de37vd2ikd028g@group.calendar.google.com'
);
var invoiceID = sheet.getRange(row, invoiceColumns.ID).getValue();
var dueDate = sheet.getRange(row, invoiceColumns.dueDate)
  .getValue();

Logger.log("Getting client data...");
var client = getRowByCellValue(
  SpreadsheetApp.getActive().getSheetByName(clientsSheet),
  sheet.getRange(row, invoiceColumns.subscriber).getValue(),
  clientColumns.name - 1
).getValues()[0];
```

Data následně uložíme do strukturovaného objektu, který použijeme při parsování.

```
var object = {
  subscriber: client[clientColumns.name-1],
  street: client[clientColumns.street-1],
  ZIP: client[clientColumns.ZIP-1],
  city: client[clientColumns.city-1],
  IN: client[clientColumns.IN-1],
  tin_title: 'DIČ',
  TIN: client[clientColumns.TIN-1],
  invoice_id: invoiceID,
  issue_date: Utilities.formatDate(
    sheet.getRange(row, invoiceColumns.issueDate).getValue(),
    calendar.getTimeZone(),
    "dd. MM. yyyy"
  ),
  due_date: Utilities.formatDate(
    dueDate,
    calendar.getTimeZone(),
    "dd. MM. yyyy"
  )
  total: 0
};
```

V případě, že subjekt není plátcem DPH, zapíšeme tuto informaci též do záhlaví faktury.

```
if (object.TIN == '') {
  object.tin_title = 'Neplátce DPH';
}
```

Dále projdeme veškeré položky faktury, které též vložíme do parsovacího objektu a s každým záznamem aktualizujeme celkovou cenu. Ceny by bylo dobré též formátovat do standardizované podoby, a proto si nejprve vytvoříme pomocnou funkci formatPrice, jež toto formátování obstará.

```
function formatPrice(value) {
  return new Intl.NumberFormat('cs-CZ', {
    style: 'currency',
    currency: 'CZK',
  }).format(value);
}
```

Máme-li pomocnou funkci pro formátování cen vytvořenou, zapíšeme do objektu jednotlivé záznamy. Normálně bychom použili cyklus forEach, ale jelikož jsme omezeni na počtem záznamů faktury, použijeme obyčejný for cyklus.

```
for (var i = 1; i < 9; i++) {
  object['description_' + i] = '';
  object['quantity_' + i] = '';
  object['unit_price_' + i] = '';
  object['subtotal_' + i] = '';
  if (records[i - 1] != null) {
    object['description_' + i] = records[i - 1].getValues()
      [0][itemColumns.description - 1];
    object['quantity_' + i] = records[i - 1].getValues()
      [0][itemColumns.quantity - 1];
    object['unit_price_' + i] = formatPrice(
      records[i - 1].getValues()[0][itemColumns.unitPrice - 1]
    );
    object['subtotal_' + i] = formatPrice(
      records[i - 1].getValues()[0][itemColumns.total - 1]
    );
    object.total += records[i - 1].getValues()
      [0][itemColumns.total - 1];
  }
}
```

Musíme zde brát v potaz, že naše konstanty obsahují jako hodnoty pozice sloupců, nikoliv jejich indexy. Proto je nutné vždy od pozice odečíst 1, abychom dostali hodnotu indexu. Než zformátujeme i celkovou cenu všech položek, využijeme tuto hodnotu pro generování QR kódu s informacemi o platbě. Za tímto účelem vytvoříme pomocnou funkci generateQR. Požadované obecné hodnoty si můžeme nastavit jako globální konstanty.

```
function generateQR(invoiceID, total) {
  var url =
    "https://api.paylibo.com/paylibo/generator/czech/image?"
    + "accountNumber=" + accountNumber
    + "&bankCode=" + bankCode
    + "&amount=" + total
    + "&currency=CZK"
    + "&vs=" + invoiceID
    + "&message=Platba za fakturu " + invoiceID
    + "&size=200";
  return UrlFetchApp.fetch(url).getBlob();
}
```

Pomocnou funkci tedy využijeme za účelem získání QR kódu a celkovou cenu zformátujeme. V této fázi je parsovací objekt zcela vyplněn, přičemž si ještě necháme jeho obsah pro účely ladění zalogovat mezi protokoly relace.

```
object.qr_code = generateQR(invoiceID, object.total);
object.total = formatPrice(object.total);
Logger.log(object);
```

Nyní můžeme přejít k samotnému parsování. Logika zde zůstává stejná, mění se pouze v případě parsování QR kódu. Ten totiž musíme vložit v obrazové podobě do odstavce nadřazeného zástupné proměnné.

```
Logger.log("Parsing data to invoice file...");
var copyID = DriveApp.getFileById(
  '1mvSjgxkzTShmx5K66a7pcUG3rftNxwcr54DCUSYQalc'
).makeCopy('Invoice_' + invoiceID).getId();
var document = DocumentApp.openById(copyID);
var body = document.getBody();
for (var property in object) {
  if(property.toUpperCase() == 'QR_CODE') {
    var templateVar = body.findText(
      "{{" + property.toUpperCase() + "}}"
    );
    var element = templateVar.getElement();
    element.asText().setText("");
    var img = element.getParent().asParagraph().insertInlineImage(
      0,
      object[property]
    );
    continue;
  }
  body.replaceText(
    "{{" + property.toUpperCase() + "}}",
    object[property]
  );
}
document.saveAndClose();
```

Tímto způsobem vparsujeme data do souboru, včetně QR kódu. Máme-li vytvořenou fakturu, zapíšeme její datum splatnosti do kalendáře.

```
Logger.log("Creating calendar event for invoice due date...");
   var event = calendar.createAllDayEvent(
  "Due date of invoice " + invoiceID,
  dueDate
);
```

Pokud jsme u záznamu klienta specifikovali jeho emailovou adresu, vytvoříme novou zprávu, kterou uložíme jako koncept ve službě Gmail. K vytvoření zprávy budeme též potřebovat pomocnou funkci createDraftEmail(), kterou nejprve vytvoříme.[15][16]

Nejprve inicializujeme veškeré proměnné potřebné k tvorbě zprávy:

```
var mimetype = 'application/pdf';
var suffix = '.pdf';
var file = DriveApp.getFileById(attachementFileID);
var forScope = GmailApp.getInboxUnreadCount();
var signature = Gmail.Users.Settings.SendAs.list("me").sendAs
  .filter(function(account) {
    if(account.isDefault){
      return true
    }
  })[0].signature;
var htmlBody = signature;
```

Dále vytvoříme samotný obsah zprávy:

```
var raw =
  'Subject: ' + subject + '\r\n'
  + 'To: ' + to + '\r\n'
  + 'Content-Type: multipart/mixed; boundary=##########\r\n\r\n'
  + '--##########\r\n'
  + 'Content-Type: text/html; charset=UTF-8\r\n\r\n' + htmlBody
  + '\r\n'
  +'--##########\r\n'
  + 'Content-Type: ' + mimetype
  + '; charset=UTF-8; name="' + file.getName() + '"\r\n'
  + 'Content-Disposition: attachment; filename="'
  + file.getName() + suffix + '"\r\n'
  + 'Content-Transfer-Encoding: base64\r\n\r\n'
  + Utilities.base64Encode(file.getAs(mimetype).getBytes())
  + '\r\n--##########\r\n';
```

A v posledním kroku zprávu uložíme jako koncept:

```
var draftBody = Utilities.base64EncodeWebSafe(
  raw,
  Utilities.Charset.UTF_8
);
var params = {
  method:"post",
  contentType: "application/json",
  headers: {
    "Authorization": "Bearer " + ScriptApp.getOAuthToken()
  },
  muteHttpExceptions: true,
  payload: JSON.stringify({"message": {"raw": draftBody}})
};
var resp = UrlFetchApp.fetch(
  "https://www.googleapis.com/gmail/v1/users/me/drafts",
  params
);
Logger.log(resp);
```

Odpověď od služby Gmail si necháme zapsat do protokolů relace, abychom ji mohli v případě chyby zanalyzovat.

Odeslání emailu pak vyvoláme následovně:

```
Logger.log("Creating email draft if address specified...");
if (!isEmpty(client[clientColumns.mail - 1])) {
  createDraftEmail(
    client[clientColumns.mail - 1],
    'Fakturace',
    copyID
  );
}
```

V tomto místě již funkce úspěšně vykonala svou činnost a tak uživateli opět pomocí zbarvení pozadí aktivního řádku zelenou barvou tuto informaci předáme.

Vyplníme-li nyní tabulku testovacími daty a dáme vygenerovat fakturu, projdou-li veškeré validace, měli bychom mít vygenerovaný nový soubor ze šablony, zápis v kalendáři v podobě celodenní události v datu splatnosti a je-li vyplněný email u klienta, nalezneme v Gmailu nový koncept s fakturou ve formátu PDF v příloze.

9 Zobecnění problematiky a tvorba doplňku

Až doposud jsme skoro v každé ukázce využili parsovací logiky. Vezmeme-li v potaz, kolikrát by mohlo být třeba v různých případech na výstupu nějaké aplikační logiky převést surová data do reprezentativnější podoby, jako například vyplněním do nějakého dokumentu, vzniká zde myšlenka tuto problematiku zobecnit a vytvořit univerzální nástroj, jež tuto práci zautomatizuje.

I zde opět využijeme možností tabulkového kalkulátoru, kdy jedinou akcí ze strany uživatele bude vyplňování požadovaných dat a jejich následné generování dle vlastního nastavení. Abychom co nejvíce minimalizovali faktor lidské chyby, je třeba, aby doplněk provedl veškerou inicializaci za uživatele. Bude tedy třeba, aby tabulku převedl do takového formátu, který pro svou činnost vyžaduje a uživatel tak jen vyplňoval data do určitých sloupců, či využil vestavěných funkcí doplňku zjednodušující rutinní procesy.

Zadejme si nejprve nároky na funkcionalitu, kterou budeme od doplňku vyžadovat:

- inicializace pracovního prostředí v tabulce
- zadání parsovací šablony pomocí integrovaného dialogu s výběrem souboru na disku
- specifikace typu zástupných proměnných za účelem podmíněného ověření dat
- možnost rozšíření počtu dostupných zástupných proměnných
- generování parsovacího listu tabulky pro specifickou šablonu
- výběr formátu výstupního souboru mezi Google Dokumentem a PDF
- dávkové zpracování vícero záznamů najednou
- možnost sjednocení vícero záznamů do jednoho výstupního souboru

Při tvorbě doplňku či vázaného skriptu s možností vyvolávání funkcí uživatelem je třeba vytvořit menu, jehož položky toto volání umožní. Možností, jak přistupovat k tvorbě menu je hned několik. Můžeme využít existujícího menu v uživatelském rozhraní tabulkového kalkulátoru, do kterého přidáme novou položku, či vytvoříme doplňkové menu, které se pod názvem doplňku umístí pod záložku Doplňky. Další možností by bylo vytvoření postranního panelu, přičemž máme více možností a větší volnost v rozložení prvků, jejich stylování apod.

Rozhodl jsem se jít cestou vytvořením doplňkového menu, jelikož jsem se domníval, že bych možností postranního panelu plně nevyužil, ba naopak by tvorba šablony a jednotlivých prvků byla zbytečně komplikovaná.

Jedním z mých požadavků byla možnost perzistentního nastavení, kdy zaškrtnutím určitých položek menu mohl uživatel definovat některé z vlastností ovlivňující funkcionalitu skriptu. Abych této perzistentnosti docílil, využil jsem vestavěné služby PropertiesService, která je schopna, kdy jednotlivé záznamy nastavení ukládá v podobě objektu, tedy hodnota přiřazená klíči. Takovou vlastností bude nastavení formátu výstupního souboru. Jelikož budeme vždy pracovat na úrovni Google Dokumentů, zvolil jsem tento formát jako defaultní. Uživatel bude mít však možnost nastavit výstupní formát na univerzálnější PDF v případě potřeby.

```
const PROPERTY_OUTPUT_FORMAT = 'output';
const OUTPUT_FORMAT_GDOC = '0';
const OUTPUT_FORMAT_PDF = '1';

function initProperties() {
  var properties = PropertiesService.getUserProperties();
  if (properties.getProperty(PROPERTY_OUTPUT_FORMAT) === null) {
    properties.setProperty(
      PROPERTY_OUTPUT_FORMAT,
      OUTPUT_FORMAT_GDOC
    );
    properties = PropertiesService.getUserProperties();
  }
  return properties;
}
```

Nemáme-li údaj o nastavení, tak položku inicializujeme s defaultní hodnotou. Existuje-li však tato položka, nebudeme její hodnotu měnit.

Dalším krokem bude vytvořit v uživatelském rozhraní nové menu. Pokud bychom chtěli menu na úrovni hlavních záložek, využili bychom funkce createMenu() nab objektem Ui, reprezentující uživatelské rozhraní. Rozhodli jsme se však pro doplňkové menu, které vytvoříme pomocí obdobné funkce createAddonMenu(). Nad vráceným objektem Menu pak můžeme volat metody pro přidání položky, oddělovače či podnabídky.

Zde však nastal problém s položkami nastavení. V rámci vestavěného menu nástrojů platformy G Suite se často setkáme s položkami nastavení, které vizuálně označují svůj stav pomocí zaškrtávacího znaku. Očekával jsem tedy, že bude logicky možnost tohoto řešení využít při tvorbě vlastního menu. Objekt Menu však touto funkcionalitou v základu nedisponuje a bylo tedy třeba využít alternativního řešení pomocí funkce buildItemName().[17]

```
function buildItemName( name, check ) {
  return ( check ? "✓ " : "\u2003" ) + name;
}
```

Vykreslování menu bude obstarávat funkce initMenu(), které na vstup předáme objekt vlastností s uživatelským nastavením.

```
SpreadsheetApp.getUi().createAddonMenu()
  .addItem('Init Addon', 'initAddon')
  .addSeparator()
  .addItem('Add Parsing File', 'showPicker')
  .addItem('Add 10 more variables', 'addMoreVariables')
  .addSeparator()
  .addSubMenu(parsingSubmenu)
  .addSeparator()
  .addSubMenu(outputFormatSubmenu)
  .addToUi();
```

Přičemž jednotlivé podnabídky jsou následující:

```
var parsingSubmenu = SpreadsheetApp.getUi()
  .createMenu('Parsing')
  .addItem('Single', 'parseDataSingle')
  .addItem('Batch', 'parseDataBatch');
```

```
var outputFormatSubmenu = SpreadsheetApp.getUi()
  .createMenu('Output format')
  .addItem(
    buildItemName(
      'G Document',
      properties.getProperty(PROPERTY_OUTPUT_FORMAT)
      ==
      OUTPUT_FORMAT_GDOC
    ),
    'setOutputFormatToGDOC'
  )
  .addItem(
    buildItemName(
      'PDF',
      properties.getProperty(PROPERTY_OUTPUT_FORMAT)
      ==
      OUTPUT_FORMAT_PDF
    ),
   'setOutputFormatToPDF'
  );
```

Rovnou zde můžeme definovat i funkce pro nastavení výstupního formátu:

```
function setOutputFormat(outputType) {
  var properties = PropertiesService.getUserProperties();
  properties.setProperty(PROPERTY_OUTPUT_FORMAT, outputType);
  properties = PropertiesService.getUserProperties();
  initMenu(properties);
}

function setOutputFormatToPDF() {
  setOutputFormat(OUTPUT_FORMAT_PDF);
}

function setOutputFormatToGDOC() {
  setOutputFormat(OUTPUT_FORMAT_GDOC);
}
```

K ostatním funkcím se dostaneme postupně v jednotlivých sekcích.

Budeme-li chtít doplněk využívat, bude třeba ho nejprve inicializovat. Jelikož tato akce odstraní veškerá dat v tabulce, ať už nějaká obsahuje či ne, musí být uživatel na tuto skutečnost upozorněn. Využijeme tak další vestavěné metody objektu Ui, kterou je alert(). Ten obstará zobrazení vyskakovacího dialogu s definovaným textem a možnostmi voleb. Dle získaného vstupu od uživatele pak můžeme ověřit, zda s touto skutečností souhlasí a můžeme vytvořit konfigurační list s vlastním spouštěčem nad tabulkou.

```
function initAddon() {
  var ui = SpreadsheetApp.getUi();
  var result = ui.alert(
     'Warning',
     'This action will result in deleting all of the data.\n'
     + 'Do you still wish to continue?',
      ui.ButtonSet.YES_NO);

  if (result == ui.Button.YES) {
    createConfigSheet();

    if (ScriptApp.getProjectTriggers() == 0) {
      var spreadsheet = SpreadsheetApp.getActive();
      ScriptApp.newTrigger('runOnEdit')
      .forSpreadsheet(spreadsheet)
      .onEdit()
      .create();
    }
  }
}

function runOnEdit(e){
  Logger.log("onEdit event fired");
  configEventListener(e);
}
```

Normálně bychom spouštěč definovali ručně, ale jelikož má skript sloužit jako univerzální doplněk, je třeba, aby i tato inicializace proběhla na straně skriptu automaticky. Mimo jiné také musíme ověřit, zda-li tabulka spouštěčem již nedisponuje, v případě, rozhodl-li by se uživatel z jakéhokoliv důvodu doplněk znovu inicializovat. Zamezíme tak nechtěné duplikaci spouštěčů, které by logiku vyvolávali vícekrát, než je požadováno. K metodě configEventListener() se dostaneme později.

Stále je však třeba metodu createConfigSheet() definovat. Jejím účelem bude odstranění všech stávajících listů tabulky a následné vytvoření konfiguračního listu, do kterého budou ukládány záznamy o jednotlivých šablonách a jejich vlastním nastavení. Obecně se bude jednat o formátování tabulky na straně kódu, kdy většina metod svým názvem jasně definuje svou funkcionalitu.

Nejprve definujeme potřebné konstanty:

```
const HEADERS = [
  'Delete', 'Generate list', 'Template name', 'Template ID'
];
const DEFAULT_ROWS = 1000;
const DEFAULT_COLUMNS = 26;
const VARIABLE_TYPES = ['String', 'Date'];
```

Ujistíme se, že v tabulce neexistuje list s názvem Config:

```
var spreadsheet = SpreadsheetApp.getActiveSpreadsheet();
if (spreadsheet.getSheetByName('Config') !== null) {
  spreadsheet.insertSheet();
  spreadsheet.deleteSheet(spreadsheet.getSheetByName('Config'));
}
```

Odstraníme všechny stávající listy tabulky a vytvoříme konfigurační list:

```
var sheets = spreadsheet.getSheets();
var configSheet = spreadsheet.insertSheet().setName('Config');
for (var index in sheets) {
  Logger.log(sheets[index]);
  spreadsheet.deleteSheet(sheets[index]);
}
```

Provedeme základní úpravy formátování konfiguračního listu, vyplníme záhlaví, přidáme sloupce s zástupnými proměnnými, do prvních dvou sloupců vložíme checkboxy a ukotvíme záhlaví:

```
configSheet.deleteColumns(1, DEFAULT_COLUMNS - 4);
configSheet.deleteRows(1, DEFAULT_ROWS - 3);

configSheet.getRange('A1:D1')
  .setValues([HEADERS])
  .setHorizontalAlignment("center")
  .setVerticalAlignment("middle")
  .setFontWeight("bold");

var ranges = ['A1:A2', 'B1:B2', 'C1:C2', 'D1:D2'];
ranges.forEach(function(range) {
  configSheet.getRange(range).merge();
});

enforceCheckbox(configSheet.getRange('A3:B'));
configSheet.getRange('A3:A').setBackground('#f4cccc');
configSheet.hideRows(3);

addConfigTemplateVariables(10, configSheet);

configSheet.getRange('C:D')
  .setHorizontalAlignment("center");

configSheet.setColumnWidth(3, 200);
configSheet.setColumnWidth(4, 400);
configSheet.setFrozenRows(2);
```

Jelikož chceme dát uživateli možnost počet zástupných proměnných zvýšit dle vlastních požadavků, oddělíme tuto logiku do vlastní funkce, kterou pak budeme moct volat samostatně za tímto účelem.

```
function addConfigTemplateVariables(howMany, sheet = null) {
  if(sheet === null) {
    sheet = SpreadsheetApp.getActiveSpreadsheet()
      .getSheetByName('Config');
  }
  for(var i = 0; i < howMany; i++) {
    var pointer = sheet.getLastColumn();
    var firstCol = pointer + 1;
    var secondCol = pointer +  2;
    sheet.insertColumnsAfter(pointer, 2);
    sheet.getRange(1, firstCol, 1, 2).merge()
      .setValue('Variable');
    sheet.getRange(2, firstCol)
      .setValue('Type').setFontStyle("italic");
    sheet.getRange(2, secondCol)
      .setValue('Name');
    sheet.getRange(1, firstCol, 3, 2)
      .setHorizontalAlignment("center")
      .setFontWeight("bold");
    enforceList(sheet.getRange(3, firstCol), VARIABLE_TYPES)
      .setFontWeight("normal").setFontStyle("italic");
  }
  SpreadsheetApp.flush();
}
```

Ve funkcích dochází mimo jiné i k vyvolávání metod enforceCheckbox() a enforceList(), přičemž do skupiny těchto metod patří ještě další metoda enforceDate(). Jedná se o pomocné metody, které nad vstupním rozsahem nastaví odpovídající podmíněné formátování.

```
function enforceCheckbox(range) {
  var enforceCheckbox = SpreadsheetApp.newDataValidation();
  enforceCheckbox.requireCheckbox();
  enforceCheckbox.setAllowInvalid(false);
  enforceCheckbox.build();
  return range.setDataValidation(enforceCheckbox);
}

function enforceList(range, list) {
  var enforceList = SpreadsheetApp.newDataValidation();
  enforceList.requireValueInList(list)
  enforceList.setAllowInvalid(false);
  enforceList.build();
  return range.setDataValidation(enforceList);
}

function enforceDate(range) {
  var enforceDate = SpreadsheetApp.newDataValidation();
  enforceDate.requireDate();
  enforceDate.setAllowInvalid(false);
  enforceDate.build();
  range.setDataValidation(enforceDate);
}
```

Existuje-li již funkce addConfigTemplateVariables(), můžeme rovnou definovat i jednoduchou funkci z menu addMoreVariables(), která rozšíří počet dostupných zástupných proměnných vždy o dalších deset:

```
function addMoreVariables() {
  addConfigTemplateVariables(10);
}
```

Než-li nastavíme odposlouchávací funkci nad konfiguračním listem, přidáme logiku pro vložení šablony za pomoci výběrového dialogu nad Google Diskem. Výhodou u nástrojů oblíbených větším počtem uživatelů a vývojářů je podpora ze strany této komunity a možnost implementace spousty existujících řešení často s přehlednou dokumentací pro snadné pochopení jejich funkcionality.

Využijeme zde tedy existujícího řešení implementace Google Pickeru, které upravíme a rozšíříme dle našich potřeb.[18]

Ve funkci createPicker() nahradíme MIME typ 'application/vnd.google-apps.folder' za 'application/vnd.google-apps.document', jelikož chceme, aby uživatel mohl vybrat Google Dokument.

Místo zobrazení prostého zobrazení identifikátoru vybrané položky ve zpětném volání předáme tento identifikátor na vstup vlastní obslužné funkce addFile(), kterou definujeme:

```
var config = SpreadsheetApp.getActive().getSheetByName('Config');
const DEFAULT_TYPE = 'String';

function addFile(id) {
  var document = DocumentApp.openById(id);
  var body = document.getBody();

  var data = [false, false, document.getName(), document.getId()];

  var variables = body.getText().match(/\{\{.([A-Z\_])*\}\}/gm);

  variables.forEach(element => {
    var variable = element.substring(2, element.length - 2)
      .replace("_", " ").toLowerCase();

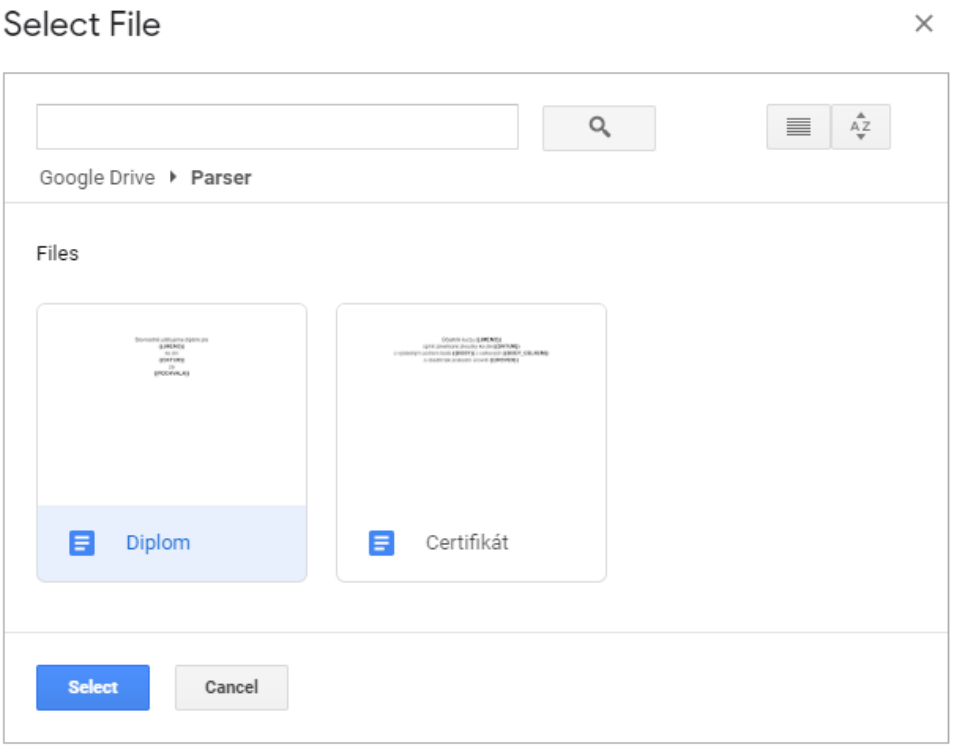
    variable = variable.charAt(0).toUpperCase()

      + variable.slice(1);

    data.push(DEFAULT_TYPE , variable);
  });

  config.appendRow(data);
}
```

Obslužná funkce addFile() prohledá soubor a získá všechny zástupné proměnné, které následně převede do formátu normálního zápisu názvu proměnné. Získaná data následně zapíše do nového řádku v konfiguračním listu.



Obr. 21 - Výběr šablony pomocí integrovaného dialogu

Máme-li nyní možnost přidávat záznamy na konfigurační list, je na čase definovat odposlouchávací funkci configEventListener(). U každého záznamu máme na výběr ze dvou možností - odstranit záznam a přidružené listy, nebo vytvořit nový list ze záznamu. Odposlouchávací funkce tedy zkontroluje, který z checkboxů byl zaškrtnut a dle toho vyvolá odpovídající logiku. Při odstraňování záznamu ověří, zda-li je checkbox pro generování zaškrtnutý, na základě čehož se rozhodne, zda-li má smazat přidružený list.

```
function configEventListener(e) {
  var spreadsheet = e.source;
  var sheet = spreadsheet.getActiveSheet();
  var watchTab = 'Config';

  if (sheet.getName() != watchTab) { return false; }

  var activeRow = e.range.getRow();
  var deleteCell = sheet.getRange(activeRow, 1);
  var generateCell = sheet.getRange(activeRow, 2);

  if (e.range.columnStart == 1 && deleteCell.isChecked()) {
    var parseSheet = spreadsheet.getSheetByName(
      sheet.getRange(activeRow, 3).getValue()
    );
    if (parseSheet != null && generateCell.isChecked()) {
      spreadsheet.deleteSheet(parseSheet);
    }
    sheet.deleteRow(activeRow);
  } else if (e.range.columnStart == 2
    && generateCell.isChecked()
  ) {
    generateSheet(
      generateCell,
      sheet.getRange(activeRow, 3).getValue(),
      chunkArray(
        sheet.getRange(
          activeRow, 5, activeRow, sheet.getLastColumn()
        ).getValues()[0],
        2
      )
    );
  }
}
```

Před definicí funkce generateSheet() je třeba ještě definovat pomocnou funkci chunkArray(), kterou předáváme na její vstup. Tato funkce rozdělí vstupní pole na rovné díly o definované velikosti. Výstupem tedy bude multidimenzionální pole se zástupnými proměnnými zapsaných ve formátu dvojic typ a název.[19]

```
function chunkArray(myArray, chunk_size){
  var results = [];
  while (myArray.length) {
    results.push(myArray.splice(0, chunk_size));
  }
  return results;
}
```

Vstupní hodnoty funkce generateSheet() jsou buňka obsahující generovací checkbox, název šablony a informace o jednotlivých sloupcích. Před vygenerováním nového listu k šabloně je třeba nejprve ověřit, zda-li v tabulce již neexistuje list se stejným jménem. V takovém případě funkci přerušíme a uživatele o této skutečnosti upozorníme pomocí vyskakovacího dialogu.

```
function generateSheet(checkboxCell, name, columns) {
  var activeSpreadsheet = SpreadsheetApp.getActiveSpreadsheet();
  var newSheet = activeSpreadsheet.getSheetByName(name);

  if (newSheet != null) {
    checkboxCell.setValue(false);
    var ui = SpreadsheetApp.getUi();
    ui.alert(
      'Same list already exists!',
      'It seems like you\'re trying to generate new list'
      + ' with the same name as one of your existing list.\n'
      + 'Try to change the template name to have a unique'
      + ' name so there\'s no conflict.',
      ui.ButtonSet.OK
    );
    return;
  }

 …
}
```

Pokud takovýto list však v tabulce neexistuje, tak jej vytvoříme a dle typu zástupné proměnné nastavíme případné ověření dat.

```
newSheet = activeSpreadsheet.insertSheet().setName(name);
newSheet.getRange(1, 1).setValue('Generate');
newSheet.getRange(1, 2).setValue('URL');

columns = columns.filter(function(columnData, index) {
  if (isEmpty(columnData[0])) { return false; }

  var columnName = columnData[1];
  newSheet.getRange(1, index + 3).setValue(columnName);
  var columnType = columnData[0];
  switch (columnType) {
    case 'String':
      break;

    case 'Date':
      enforceDate(
        newSheet.getRange(2, index + 3, DEFAULT_ROWS, 1)
      );
      break;

    default:
      break; // THROW EXCEPTION
  }
  return true;
});

newSheet.deleteColumns(
  columns.length + 3,
  DEFAULT_COLUMNS - columns.length - 2
);

newSheet.setFrozenRows(1);
newSheet.getRange(1, 1, DEFAULT_ROWS, newSheet.getLastColumn())
  .setHorizontalAlignment("center");
newSheet.getRange(1, 1, 1, newSheet.getLastColumn())
  .setFontWeight("bold");
enforceCheckbox(newSheet.getRange("A2:A"));
```

Jelikož bude často potřeba ověřit, zda-li je obsah proměnné prázdný, vytvoříme si pomocnou funkci isEmpty(), abychom neustále neopakovali stejný zápis podmínek.

```
function isEmpty(variable) {
  return variable == ''
    || variable == undefined
    || variable == null
    || variable.length <= 0;
}
```

Momentálně máme v listu typů proměnných jen textový řetězec a datum. Pro textový řetězec není přidávat ověření dat, pro datum však ano. Zároveň, nastala-li by situace, kdy by nám na vstup funkce switch přišel neznámí typ proměnné, můžeme například zapracovat vyhození nové výjimky. Dále je zde také prostor jednoduchou implementaci dalších typů v případě potřeby.

Zaškrtneme-li nyní checkbox generování listu pro specifický záznam šablony, vytvoří se nám tak nový list na základě vstupních proměnných a jejich uživatelem definovaných typů. Dostáváme se tak k poslední části, kterou je samotná parsovací logika, tvořící jádro celého doplňku. Definujme tedy nejprve potřebné proměnné, konstanty a funkce vyvolávající samotné parsování.

```
const LINK_PLACEHOLDER = '#@#@#@#@#@';

const OUTPUT_MODE_SINGLE = '0';
const OUTPUT_MODE_BATCH = '1';

function parseDataSingle() {
  parseData(OUTPUT_MODE_SINGLE);
}

function parseDataBatch() {
  parseData(OUTPUT_MODE_BATCH);
}
```

Dle parsovacího módu můžeme v kódu rozhodnout, zda-li mají být soubory parsovány samostatně odděleně, či hromadně se sjednoceným výstupem do jednoho souboru. V případě dávkového parsování nám pak poslouží hodnota konstanty LINK_PLACEHOLDER jako zástupný symbol za neznámý odkaz v průběhu parsování.

Přejděme nyní k funkci parseData() se vstupním parametrem určujícím parsovací mód. Nejprve je třeba získat uživatelské nastavení, které nám umožní určit typ výstupu, na základě čehož budeme s daty pracovat rozdílným způsobem. Následně získáme data z aktivního listu aktivního listu, ke kterému přiřadíme záznam konfigurace. Vytvoříme si proměnné, do kterých si uložíme identifikátor šablony a informace o zástupných proměnných.

```
var properties = PropertiesService.getUserProperties();

var dataSheet = SpreadsheetApp.getActiveSheet();
var data = dataSheet.getDataRange().getValues().splice(1);

var configSheet =
SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Config');
var configRow = getRowByCellValue(configSheet,
dataSheet.getName(), 3).getValues()[0];

var templateID = configRow.splice(0, 4)[3];
var templateColumns = chunkArray(configRow, 2);
```

Pomocná funkce getRowByCellValue() a následně pak i funkce getRowsByCellValue() zůstávají stejné, jako v předchozí ukázce.

Pokud budeme provádět dávkové parsování, vytvoříme si soubor, do kterého budeme vyplněné šablony vkládat na novou stránku, přičemž abychom měli tento soubor na stejném místě jako tabulku, budeme muset získat její aktuální umístění a soubor přesunout. Musíme však provést kontrolu, zda-li se nenacházíme v kořenové složce, jelikož v takovém případě není třeba soubor přemisťovat.

```
if(outputMode === OUTPUT_MODE_BATCH) {
  var batchDocument = DocumentApp.create('Batch export');
  var batchDocumentID = batchDocument.getId();

  if (DriveApp.getFileById(templateID).getParents().hasNext()) {
    var targetFolder = DriveApp.getFileById(templateID)
        .getParents().next();
    var file = DriveApp.getFileById(batchDocumentID);
    file.getParents().next().removeFile(file);
    targetFolder.addFile(file);
  }
}
```

Nyní můžeme pomocí forEach cyklu projít data aktivního listu a zpracovat vybrané položky.

```
data.forEach(function(row, index, data) {
  if(isEmpty(row[0]) || !row[0]) { return; }
  var documentID = parse(
    templateID,
    templateColumns,
    row.splice(2),
    index
  );
  var document = DocumentApp.openById(documentID);
  var fileURL = document.getUrl();
  switch(outputMode) {
    case OUTPUT_MODE_SINGLE:
      if(
        properties.getProperty(PROPERTY_OUTPUT_FORMAT)
        ===
        OUTPUT_FORMAT_PDF
      ) {
        fileURL = convertDocToPDF(documentID, document).getUrl();
      }
      dataSheet.getRange(index + 2, 2)
        .setValue('=hyperlink("' + fileURL + '";"FILE")');
      break;
    case OUTPUT_MODE_BATCH:
      mergeDocs(batchDocumentID, documentID);
      dataSheet.getRange(index + 2, 2).setValue(LINK_PLACEHOLDER);
      break;
    default:
      // THROW EXCEPTION
      break;
  }
  dataSheet.getRange(index + 2, 1).removeCheckboxes();
});
```

Nejprve ověříme, zda-li je záznam na daném řádku označen zaškrtnutím checkboxu ke zpracování. Nehledě na parsovací mód či výstupní formát vždy vytvoříme kopii šablony s vyplněnými daty ve formátu Google Dokumentu a vložíme ho do proměnné, ke které ještě vytvoříme pomocnou proměnnou obsahující odkaz na tento soubor.

Další zpracování se liší dle parsovacího módu a proto pomocí funkce switch vybereme odpovídající mód, jehož logika se vykoná.

V případě jednotného zpracování pouze zkontrolujeme, zda-li je výstupní formát nastaven do PDF a pokud ano, tak Google Dokument zkonvertujeme a přepíšeme odkaz na nový soubor, který následně vložíme do druhého sloupce aktivního řádku.

Chceme-li provést zpracování dávkového vložíme obsah nově vytvořeného souboru na novou stranu jednotného výstupního souboru a místo odkazu vložíme do sloupce zástupný symbol, jež jsme si na začátku definovali.

Přišla-li by na vstup této funkce jiná hodnota, můžeme například vyhodit novou výjimku, přičemž zde máme i prostor na případnou implementaci dalších parsovacích módů.

V každém případě po úspěšném parsování odstraníme checkbox, jelikož není třeba znovu parsovat daný záznam.

Po ukončení cyklu ještě musí dojít ke kontrole, zda-li je parsovacím módem dávkové zpracování, kdy uživatel nastavil výstupní formát jako PDF. Nastane-li tato situace, je třeba podniknout krok navíc, kdy jednotný výstupní soubor převedeme do formátu PDF. Nehledě na formát je pak ještě třeba zástupný symbol nahradit reálným odkazem.

```
if(outputMode === OUTPUT_MODE_BATCH) {
  if(
    properties.getProperty(PROPERTY_OUTPUT_FORMAT)
    ===
    OUTPUT_FORMAT_PDF
  ) {
      batchDocument = convertDocToPDF(
        batchDocumentID,
        DocumentApp.openById(batchDocumentID)
      );
  }
  var dataRange = dataSheet.getRange(2, 2, dataSheet
    .getDataRange().getLastRow(), 2);
  var linkPlaceholders = dataRange.getValues();
  replaceInSheet(
    linkPlaceholders,
    LINK_PLACEHOLDER,
    '=hyperlink("' + batchDocument.getUrl() + '";"FILE")'
  );
  dataRange.setValues(linkPlaceholders);
}
```

Pojďme si teď přiblížit všechny nové funkce, jež se v této metodě volají. Začneme vůbec hlavní funkcí parse(), která obstarává samotné parsování dat do šablony. Jejími vstupními daty jsou identifikátor šablony, nastavení zástupných proměnných a data z řádku aktivního listu. Inicializujeme pole pro vstupní data nahrazující zástupné proměnné, které nejprve dle typu zformátujeme.

```
var BreakException = {};
var data = [];
try {
  templateColumns.forEach((templateColumn, index) => {
    if(isEmpty(templateColumn[1])) { throw BreakException; }
    var value = rowData[index];

    switch (templateColumn[0]) {
      case 'String':
        value = value.toString();
        break;
      case 'Date':
        value = Utilities.formatDate(
          new Date(value),
          SpreadsheetApp.getActive().getSpreadsheetTimeZone(),
          "dd. MM. yyyy"
        );
        break;

      default:
        // THROW EXCEPTION
        break;
    }

    data.push({
      'name': templateColumn[1],
      'type': templateColumn[0],
      'value': value
    });
  });
} catch(e) {
  if (e !== BreakException) throw e;
}
```

Chceme-li forEach cyklus předčasně ukončit, můžeme využít řešení, kdy si vytvoříme vlastní výjimku, kterou v požadovaném okamžiku vyhodíme v bloku try/catch. V případě odchycení této výjimky nezastavujeme běh aplikace, ale výjimku ignorujeme a pokračujeme v konání funkce.[20]

Opět zde pracujeme s jednotlivými datovými typy a stejně jako v předchozích případech i zde můžeme definovat logiku, pro neznámý typ či jednoduše rozšířit funkcionalitu v případě přidání typu nového.

Následně načteme šablonu, vytvoříme kopii, inicializujeme její obsah a ve smyčce forEach nahradíme zástupné znaky odpovídajícími hodnotami, jako v předchozích ukázkách. Dokument pak uložíme a na výstupu této funkce vrátíme jeho identifikátor.

```
var template = DriveApp.getFileById(templateID);
var documentID = template.makeCopy(
  template.getName() + '_copy'
).getId();
var document = DocumentApp.openById(documentID);
var body = document.getBody();

data.forEach((column) =>  {
  body.replaceText(
    "{{" + column.name.replace(" ", "_").toUpperCase() + "}}",
    column.value
  );
});

document.saveAndClose();

return documentID;
```

V případě konverze formátu souboru z Google Dokumentu do PDF voláme funkci convertDocToPDF(). Ta získá umístění souboru předaného na vstupu, vytvoří nový soubor v požadovaném formátu a starý odstraní, neurčíme-li na vstupu posledním parametrem jinak.

```
function convertDocToPDF(fileID, document, remove = true) {
  var file = DriveApp.getFileById(fileID);
  var folder = DriveApp.getRootFolder();
  if (file.getParents().hasNext()) {
    folder = file.getParents().next();
  }
  var blob = document.getAs('application/pdf');
  blob.setName(document.getName() + ".pdf");
  var pdf = folder.createFile(blob);

  if(remove) { folder.removeFile(file); }

  return pdf;
}
```

V případě dávkového zpracování nově sjednocujeme soubory jednotlivých záznamů do jednotného souboru. Za tímto účelem využíváme funkce mergeDocs(), která vezme identifikátory dvou Google Dokumentů ze vstupu, vyvolá pomocí nich soubory a vloží data z druhého souboru do prvního. Opět zde můžeme volitelným parametrem specifikovat, zda-li má být druhý soubor po vykonání sjednocení odstraněn, či nikoliv. Dalším volitelným parametrem je appendPageBreak, který určuje, zda-li se mají záznamy vkládat přímo za sebe, či mají-li být umístěné na novou stranu.[21]

Nejprve je ve třeba inicializovat oba dokumenty s obsahem a počtem prvků.

```
function mergeDocs(
  firstDocID,
  secondDocID,
  appendPageBreak = true,
  removeSecondDoc = true
) {
  var firstDoc = DocumentApp.openById(firstDocID);
  var body = firstDoc.getBody();
  var otherBody = DocumentApp.openById(secondDocID).getBody();
  var totalElements = otherBody.getNumChildren();
  …
}
```

Dále pomocí cyklu projdeme všechny elementy druhého souboru a dle typu zapíšeme odpovídající obsah do souboru prvního. V případě nespecifikovaného elementu vyhodíme výjimku:

```
for( var j = 0; j < totalElements; ++j ) {
  var element = otherBody.getChild(j).copy();
  var type = element.getType();
  switch(type) {
    case DocumentApp.ElementType.PARAGRAPH:
      body.appendParagraph(element);
      break;

    case DocumentApp.ElementType.TABLE:
      body.appendTable(element);
      break;

    case DocumentApp.ElementType.LIST_ITEM:
      body.appendListItem(element);
      break;

    default:
      throw new Error(
        "Passed type couldn't appear in the body: " + type
      );
      break;
  }
}
```

Následně určíme, zda-li má být obsah odsazen na novou stranu, či má-li dojít k odstranění druhého souboru. Nakonec soubor uložíme a zavřeme, na výstup není třeba předávat žádnou hodnotu.

```
if(appendPageBreak) { firstDoc.getBody().appendPageBreak(); }

if(removeSecondDoc) {
  var file = DriveApp.getFileById(secondDocID);
  var folder = file.getParents().next();
  folder.removeFile(file);
}

firstDoc.saveAndClose();
```

Poslední doposud neznámou funkcí je funkce replaceInSheet(), kterou využíváme v případě, kdy nahrazujeme zástupný symbol za odkaz na výstupní soubor parsovacího procesu. Jedná se o jednoduchou pomocnou funkci, která projde vstupní rozsah daného sloupce a v případě shody se specifikovanou hodnotou zamění obsah sloupce za další určenou hodnotu.[22]

```
function replaceInSheet(values, to_replace, replace_with) {
  for(var row in values){
    var replaced_values = values[row]
      .map(function(original_value) {
        return original_value.toString()
          .replace(to_replace,replace_with);
      });
    values[row] = replaced_values;
  }
}
```

Nyní máme veškeré potřebné funkce specifikované a logika doplňku je nyní dle požadovaných kritérií kompletní a plně funkční. Uživatel může nahrát záznam o souboru šablony, navýšit počet dostupných zástupných proměnných, vygenerovat list pro danou šablonu, specifikovat formát výstupních souborů a hromadně zpracovat neomezené množství záznamů jednotlivě, či dávkově.

Doplněk bychom v tomto okamžiku mohli publikovat, ale jelikož Google výrazně specifikuje, aby byl doplněk odladěný a zcela funkční, tomuto kroku se vyhneme, jelikož je zde hodně míst ke zlepšení, chybějící logiky či možných scénářů, pro které není definováno chování.

10 Závěr

Stejně jako každý jazyk a technologie i v rámci Google App Script je zde mnoho zákoutí, kterých je v mém případě třeba ještě rozkrýt. Podrobná dokumentace, rozsáhlá a nadšená komunita a záštita jedním z největších technologických gigantů jsou jednoznačnými pozitivními kritérii pro výběr této technologie. Možnost provázat vestavěné nástroje na základě vlastních specifických potřeb je pro Google skrytým esem v rukávu v oblasti sad kancelářských aplikací. Není proto divu, že má ve světě takové zastoupení ze strany uživatelů a pro svou jednoduchost je tak velice oblíbená.

Při zpracování své bakalářské práce jsem narazil na mnoho nových informací, i na jistá úskalí, obojí mi však bylo velkým přínosem. Nejvíce mě obohatila nově získaná znalost práce s různými vestavěnými objekty, pomocí kterých mohu jednotlivé služby mezi sebou propojovat a vytvořit tak aplikace úplně jiným způsobem, než jsem byl doposud zvyklý. Z nějakého důvodu bylo mé první setkání s JavaScriptem negativní, a proto jsem na něj i tímto způsobem nahlížel. Během zpracování jednotlivých ukázek a z důvodu úzkého propojení s App Scriptem jsem si však uvědomil, že to není špatná technologie a postupně jsem si jí nakonec oblíbil. Možnost komunikace se službami třetích stran přes rozhraní API zde otevírá možnosti napojení služeb Google k vlastním aplikacím, či naopak.

Mezi úskalí, na která jsem narazil, pro mě byla převážně práce s komplexním XML souborem v rámci tvorby ukázky využívající systému ARES, jelikož s tímto formátem běžně nepracuji. Dále byla menším problémem především moje nutkavá touha po neustálé inovaci, která mě dokázala u určité části zdržet i několik dní. Způsobů, jakými lze dojít z bodu A do bodu B je totiž enormní množství, a ne vždy bylo první mnou vybrané řešení zcela optimální. Často jsem tedy vyhledával inspiraci v již existujících implementacích jinými a snažil jsem se si z každé odnést co nejvíc. Stejně jako další dovednosti, i tato se rozvíjí s praxí a studiem. Ne nadarmo se říká, že chybami se člověk učí, jelikož spoustu zahozených návrhů řešení mi i tak bylo cennou oporou při tvorbě finální podoby doplňku.

Celkově se mi toto téma velmi líbilo a obohatilo mě, proto v něm určitě budu pokračovat i nadále v osobním životě mimo tuto práci a rozhodně věřím, že mi veškeré získané znalosti budou jako programátorovi obecně v rámci vývoje velikým přínosem.

Seznam použitých informačních zdrojů

[1] What Is the Cloud?: Cloud Definition. Cloudflare [online]. 2020 [cit. 2020-04-15]. Dostupné z: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/

[2] Cloud computing. IBM [online]. [cit. 2020-04-15]. Dostupné z: https://www.ibm.com/cloud/learn/cloud-computing

[3] Cloud Computing Definition. Investopedia [online]. Frankenfield, 2019 [cit. 2020-04-15]. Dostupné z: https://www.investopedia.com/terms/c/cloud-computing.asp

[4] Pochopte co je cloud, cloud computing a další pojmy. Quadro Net [online]. 2020 [cit. 2020-04-15]. Dostupné z: http://www.quadronet.cz/pochopte-co-je-cloud-cloud-computing-a-dalsi-pojmy/

[5] What is Cloud Storage and How Does It Work? TechTarget [online]. Rouse, 2020 [cit. 2020-04-15]. Dostupné z: https://searchstorage.techtarget.com/definition/cloud-storage

[6] OneDrive, Google Disk, Dropbox. Který je pro firmu nejlepší? Bestonline.cz [online]. Sýkora, 2019 [cit. 2020-04-15]. Dostupné z: https://www.bestonline.cz/blog/onedrive-vs-gdrive-vs-dropbox/

[7] Cloud Storage vs. Cloud Computing: What's the Difference? Cloud Carib [online]. 2018 [cit. 2020-04-15]. Dostupné z: https://info.cloudcarib.com/blog/cloud-storage-vs.-cloud-computing-whats-the-difference

[8] Cloud Computing VS Cloud Storage: Back to Cloud Basics. NewCloud Networks [online]. NewCloud Networks, 2017 [cit. 2020-04-15]. Dostupné z: https://blog.newcloudnetworks.com/cloud-computing-vs-cloud-storage-back-to-cloud-basics

[9] G Suite. KIV/ZIM - Znalostní a informační management [online]. Šrámek, 2018 [cit. 2020-04-15]. Dostupné z: https://students.kiv.zcu.cz:3443/projects/kiv-zim/wiki/G_Suite

[10] Write Google Scripts using Modern JavaScript ES6. Digital Inspiration [online]. Agarwal, 2020 [cit. 2020-04-15]. Dostupné z: https://www.labnol.org/es6-google-apps-script-v8-200206

[11] Google Apps Script: A Beginner's Guide. Ben Collins [online]. Collins, 2019 [cit. 2020-04-15]. Dostupné z: https://www.benlcollins.com/apps-script/google-apps-script-beginner-guide/

[12] Web Apps: Apps Script. Google Developers [online]. Google, 2019 [cit. 2020-04-15]. Dostupné z: https://developers.google.com/apps-script/guides/web

[13] Moving Files In Google Drive Using Google Script. Stack Overflow [online]. Woodhouse, 2016 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/38808875/moving-files-in-google-drive-using-google-script

[14] ARES jako otevřená data? Ministerstvo financí šlo cestou nejmenšího odporu. Lupa.cz [online]. Kokeš, 2020 [cit. 2020-04-15]. Dostupné z: https://www.lupa.cz/clanky/ares-jako-otevrena-data-ministerstvo-financi-slo-cestou-nejmensiho-odporu/

[15] Adding attachment to Gmail draft with Google Apps Script and Gmail API. Stack Overflow [online]. Rutte, 2017 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/45976952/adding-attachment-to-gmail-draft-with-google-apps-script-and-gmail-api

[16] Apps Script to get the users signature. Stack Overflow [online]. Davids, 2017 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/41727640/apps-script-to-get-the-users-signature/41729412#41729412

[17] Can you make menu items in Google spreadsheet custom menus tickable? Stack Overflow [online]. SamofWise, 2012 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/13451501/can-you-make-menu-items-in-google-spreadsheet-custom-menus-tickable/13452486#13452486

[18] Using Google Picker with Google Apps Script. Digital Inspiration [online]. Agarwal, 2017 [cit. 2020-04-15]. Dostupné z: https://www.labnol.org/code/20039-google-picker-with-apps-script

[19] How to split an array into chunks of the same size easily in Javascript. Our Code World [online]. Delgado, 2016 [cit. 2020-04-15]. Dostupné z: https://ourcodeworld.com/articles/read/278/how-to-split-an-array-into-chunks-of-the-same-size-easily-in-javascript

[20] Short circuit Array.forEach like calling break. Stack Overflow [online]. Klarenbach, 2010 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/2641347/short-circuit-array-foreach-like-calling-break

[21] How can I generate a multipage text document from a single page template in google-apps-script? Stack Overflow [online]. Insas, 2012 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/10692669/how-can-i-generate-a-multipage-text-document-from-a-single-page-template-in-goog

[22] Google Apps Script for Multiple Find and Replace in Google Sheets. Stack Overflow [online]. Stephen, 2017 [cit. 2020-04-15]. Dostupné z: https://stackoverflow.com/questions/42150450/google-apps-script-for-multiple-find-and-replace-in-google-sheets

Seznam příloh

Příloha 1 – Vypracované ukázky

Seznam obrázků

Obr. 1 - Rozložení prvků online editoru 14
Obr. 2 - Ukázkové nastavení spouštěče 15
Obr. 3 - Výběr funkce k ladění 28
Obr. 4 - Ukázkové nastavení breakpointů 28
Obr. 5 - Debugovací lišta s možnostmi při ladění 28
Obr. 6 - Obsah proměnných při ladění 29
Obr. 7 - Obsah proměnných při ladění v dalším kroku 29
Obr. 8 - Záznam spuštění a jeho protokoly 31
Obr. 9 - Struktura tabulky pro záznamy událostí 36
Obr. 10 - Proklik na nastavení zvoleného kalendáře 38
Obr. 11 - Přehled jednotlivých oprávnění projektu 46
Obr. 12 - Protokoly testovací relace 47
Obr. 13 - List klientů 52
Obr. 14 - List faktur 52
Obr. 15 - List položek 52
Obr. 16 - Šablona faktury 53
Obr. 17 - Vlastní spouštěč vyvolávající funkci runOnEdit při úpravě obsahu tabulky 56
Obr. 18 - Indikace spuštění funkce loadInfo 65
Obr. 19 - Indikace nastalé chyby 65
Obr. 20 - Ukázka zápisu úspěšně zinicializovaných dat z ARES 65
Obr. 21 - Výběr šablony pomocí integrovaného dialogu 97