Slovenská technická univerzita

Fakulta informatiky a informačných technológií

Ilkovičova 3, 842 16 Bratislava 4

OWNET ANDROID
PROJEKTOVÁ DOKUMENTÁCIA

Jozef Arpáš, Marek Lóderer, Jaroslav Rais, Pavol Ružička
Michal Roško, Vladimír Sudor, Juraj Volentier

Tím č. 6: Marmots

Predmet: Tímový projekt

Vedúci projektu: Ing. Michal Barla, PhD.

Ak. rok: 2012/13

Kontakt: fiit.tim.06@gmail.com

Obsah

1 Úvod

Po pridelení témy *OFFLINE WEB*, sme sa spolu s naším mentorom Michalom Barlom dohodli, že budeme v rámci predmetu tímový projekt pracovať na mobilnej aplikácii pre platformu Android[1], ktorá bude poskytovať funkcionalitu aplikácie *OWNET*.[2]

*OWNET*aplikácia má umožňovať používateľom s nestabilným internetovým pripojením surfovať offline, teda aj v prípade, že momentálne nemajú prístup k internetu. Zabezpečiť to má spôsobom, že obsah používateľom navštívených stránokmá ukladať na SD kartu zariadenia(mobilný telefón, tablet). V prípade výpadku internetového pripojenia, sa tak načíta obsah jednotlivých stránok práve z údajov uložených na SD karte.

Okrem tejto hlavnej požiadavky, by mala aplikácia poskytovať používateľovi aj portál, kde by mohol editovať obsah uložených stránok na SD karte, prípadne tagovať si preňho zaujímavé stránky.

Aplikácia by mala rovnako zabezpečovať inteligentné pred načítavanie stránok v čase, keď nie je internetové pripojenie využívané. Pri výbere, ktoré stránky pred načítavať, by sa mali zohľadňovať zaznamenaná používateľovaaktivita, teda aké stránky najčastejšie navštevuje a rovnako aj odporúčania alebo aktivita jeho priateľov, prípadne ľudí pripojených v rovnakej sieti(napr. učiteľ v africkej škole môže odporučiť svojim žiakom zaujímavé video z oblasti biológie).

Nakoľko sme sa rozhodli realizovať naše riešenie aplikácie OWNET agilným spôsobom vývoja, konkrétne metodikou SCRUM, budú jednotlivé kapitoly tohto dokumentu opisovať jednotlivé šprinty.

[1] http://www.android.com/

[2] http://ownet.fiit.stuba.sk/?l=sk

2 Obzerance

Na začiatku úvodného šprintu bolo potrebné hlavne zanalyzovať možnosti platformy Android, teda či a aké nám poskytuje nástroje pre realizáciu pôvodnej funkcionality projektu *OWNET*.Na začiatku bol teda treba v prvom rade vyriešiť:

1. Realizáciu proxy na Androide, pre odchytávanie požiadaviek webového prehliadača.
2. Zistiť ako je možné ukladať obsah sťahovaných stránok na SD kartu.
3. Vytvoriť databázu, ktorá by reprezentovala obsah stránok na SD karte.

Tieto identifikované User Stories, sme následne realizovali pomocou ďalej opísaných úloh.

2.1 Princípy fungovanie*Service* na androide

Service(služba) je aplikačným komponentom, ktorého cieľom je zabezpečiť poskytovanie dlhších operácií v čase mimo používateľovej interakcie alebo poskytnúť funkcionalitu iným aplikáciám. Vlastnosti *služby*:

- zabezpečuje beh aplikácií na pozadí,
- zabezpečuje beh aplikácií bez používateľského rozhrania,
- pre vytvorenie služby musíme implementovať triedu, ktorá dedí od triedy *Service*,
- následne treba definovať metódy *onCreate()*, *onBind()* a*onDestroy()*,
- službu je tiež potrebnédeklarovať v súbore AndroidManifest.xml nasledovným spôsobom:

```
<manifest ... >
  ...
<application ... >
<service android:name=".ExampleService" />
    ...
</application>
</manifest>
```

- pre správne vytvorenie službyje potrebné ho umiestniť do vlastného vlákna, aby nebežal v hlavnom.

Rozoznávame dva typy:

1. *Started*- služba beží nepretržite a neurčitúdobu, až kým je neprerušíme.
2. *Bounded*- služba je viazaná na nejakú aplikáciu a beží vtedy, keď je vyvolaná iným procesom.

Metódy, ktoré je potrebné implementovať pre každú službu:

1. *onStartComand()*- systém vola túto metódu keď žiadame o začatie vykonávania služby. To sa uskutoční volaním metódy *startService*().
2. *onBind()* - metóda je volaná v prípade,že iný prvok chce byť naviazaný na službu. Tato metóda vracia komunikačný kanál sa danú službu.
3. *onCreate()* - metóda sa vykonáva predtým, ako je služba spustená, čiže predtým ako je volaná metóda *onStartCommand()* alebo *onBind()*. V prípade, že služba už beží, táto metóda nieje vyvolaná.
4. *onDestroy()* - metóda je volaná, keď sa už služba nepoužíva a je ukončená.

2.2 Ako nastaviť proxy na Androide

Medzi jednu z elementárnych požiadaviek, ktorú je potrebné zabezpečiť kvôli funkčnosti aplikácie, patrí nastavenie proxy v prehliadači alebo v telefóne.

Nastavenie proxy pre operačný systém Android(verzia 2.3)

1. All apps
2. Settings
3. Wireless & networks
4. Wi-Fi Settings
5. Press the Menu button on your phone
6. Click on Advanced
7. Wi-Fi proxy – Wi-Fi proxy related settings
8. Click Enable Wi-Fi Proxy
9. Enter the Wi-Fi proxy host
10. Enter the Wi-Fi Proxy port

Chrome

- Ponúka možnosti nastavenia Proxy
- Ponúka možnosť offline webu pre Android 4.0

Firefox

1. zadáme„*about:config*“ do prehliadača
2. zadáme„*proxy.http*“ do okna v *about:config*
3. do pola*network.proxy.http*zadáme meno serveru
4. do pola*nerwork.proxy.http_port* zadáme číslo portu

Doplhin HD

- Nejde nastaviť samostatne na prehliadači

Boat browser

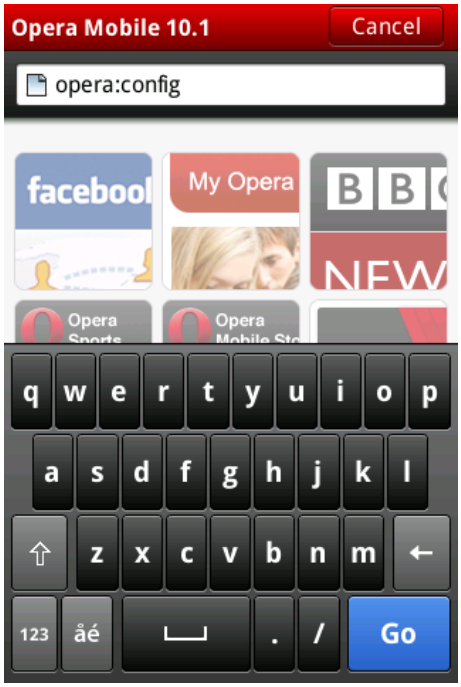
- Zatiaľ sa nedá nastaviť proxy.

Rozhodli sme sa používať prehliadač Opera, kde sa dá nastaviť proxy.

Opera

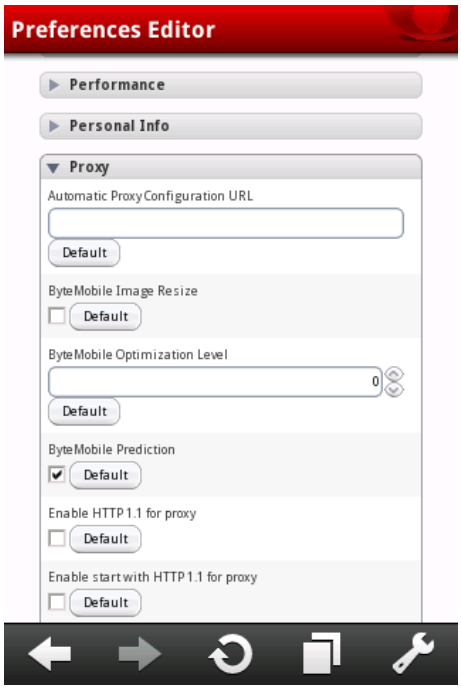
Pre prehliadač Opera Mobile je postup nasledovný:

1. Pre prístupu k nastaveniam prehliadača zadáme do url adresy *opera:config* a potvrdíme.



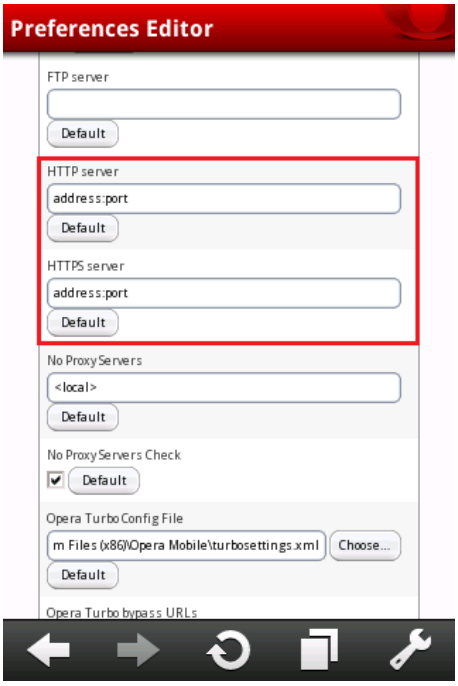
Obr. 2.1 opera:config

2. Na zobrazenej stránke vyhľadáme *Proxy* a zvolíme na túto možnosť. Na displeji sa nám zobrazí nasledovná obrazovka.



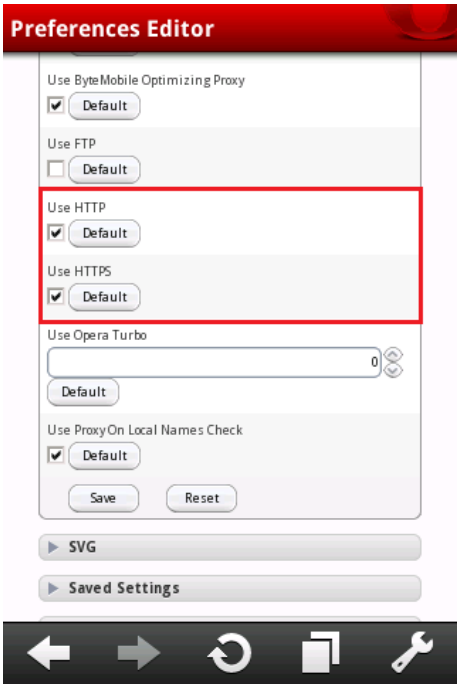
Obr. 2.2 opera:config - proxy

3. Vyhľadáme pole *HTTP server*, do ktorého zapíšeme nami vybranú adresu proxy v tvare *adresa:port*, napr. moja.proxy.com:3128.



Obr. 2.3 opera:config – proxy –HTTP server

4. Nižšie vyhľadáme pole *Use HTTP*a zaškrtneme ho.



Obr. 2.4 opera:config – proxy – use http

2.3 Vytvorenie proxy service

Analýza

Jednou z kľúčových úloh OwNetaplikácie je zachytávanie požiadaviek webových prehliadačov na mobilnom zariadení. Operačný systém Android poskytuje API obsahujúce balík *java.net*.V balíku sa nachádza trieda *java.net.ServerSocket*, teda server ktorý čaká na prichádzajúce spojenie (*connection*) od klientov.

ServerSocket sa otvára na konkrétnom porte, ktorý sa zadáva pri vytváraní objektu.Na mobilných zariadeniach je povolený rozsah portov na intervale <1025, 65536>.

Trieda *ServerSocket* poskytuje metódu *accept()*, ktorá zachytáva požiadavky.Metóda *accept()*obsahuje blokujúce čakanie.

Návrh

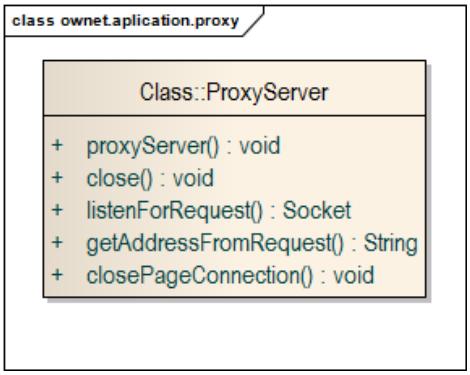
Princíp fungovania:

- Vytvorí sa objekt *ServerSocket*, ktorý počúva na zadanom porte.
- Vyvolá metódu *accept()*, ktorá čaká, až kým nezachytí požiadavku.
- Následne sa vytvorí objekt *Socket*, ktorý preberá zachytenú požiadavku a komunikuje s klientom.
- Objekt *SocketServer*môže prijímať ďalšie požiadavky.

Ukážka prichádzajúcej požiadavky z webového prehliadača:


```
GEThttp://www.youtube.com/HTTP/1.1
User-Agent: Opera/9.80 (Windows NT 6.1; U; sk) Presto/2.10.289 Version/12.02
Host: www.youtube.com
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png,
image/webp, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: sk-SK,sk;q=0.9,en;q=0.8
Accept-Encoding: gzip, deflate
Cookie: PREF=f1=10000000&fv=11.4.402; VISITOR_INFO1_LIVE=xw0S4E66vXY
Proxy-Connection: Keep-Alive
```

Implementácia



Obr. 2.5 balík ownet.aplication.proxy

ProxyServer

- *start()*–metóda vytvára objekt *ServerSocket*. Otvára ho na porte, ktorý bol zadaný v konštruktore triedy *ProxyServer*.
- *close()*- metóda slúži na ukončenie práce s objektom *ServerSocket*.
- *listenForRequest()*- metóda slúži na zachytenie požiadaviek z internetových prehliadačov. Vytvára objekt *Socket*, ktorý preberá celu komunikáciu s klientom (odpovedá na danú požiadavku).
- *getAddressFromRequest()* - metóda vracia URL adresu, ktorá je uložená v hlavičke požiadavky (*httpRequest*-u).
- *closePageConnection()* -metóda slúži na ukončenie/uzatvorenie spojenia s klientom.

2.4 Získavanie odkazov

Analýza

Táto časť zastrešuje spôsob získavania dát z internetu a ich modifikáciu pre uloženie a následné opätovné načítanie dát. Pokiaľ aplikácia zachytí vstupný tok údajov, ktorý prichádza z internetu, je potrebné tento tok uložiť na SD kartu zariadenia. Pri opätovnom navštívení tejto stránky aplikácia použije súbory z SD karty.

Nakoľko konkrétna stránka sama posiela požiadavky na potrebný obsah, nie je potrebné ju parsovať a dolovať z nej údaje ako obrázky, podporné HTML stránky a skripty. Pokiaľ by to takto nefungovalo, bolo by potrebné stánku parsovať. Takúto možnosť nám ponúka knižnica JSoup, pomocou ktorej by sme vytvorili DOM štruktúru. Na základe tejto štruktúry by sme vytvorili adresáre a súbory na SD karte. Iným, ale účinným riešením by bolo naprogramovanie vlastnej knižnice na parsovanie.

Po vytvorení stromovej štruktúry adresárov, je potrebné zapísať vstupný tok údajov danej požiadavky na SD kartu zariadenia. Následne je potrebné, aby prehliadač dokázal k požiadavke priradiť príslušný súbor uložený na SD karte. Túto funkcionalitu by mala zabezpečiť databáza.

Návrh

Nakoľko prehliadač sám posiela požiadavky na potrebný obsah, nie je potrebné obsah stránok parsovať. Požiadavka má formu URL adresy. Zo získanej URL adresy následne vytvoríme stromovú štruktúru pre adresáre. Do tejto štruktúry potom vložíme súbor, ktorý obsahuje obsah požiadavky(HTML stránka, skript, obrázok a iné.).

Pokiaľ sa už požiadavka nachádza na SD karte, je potrebné, aby aplikácia poslala ako odpoveď na požiadavku Input stream z uloženého súboru a nie z internetu.

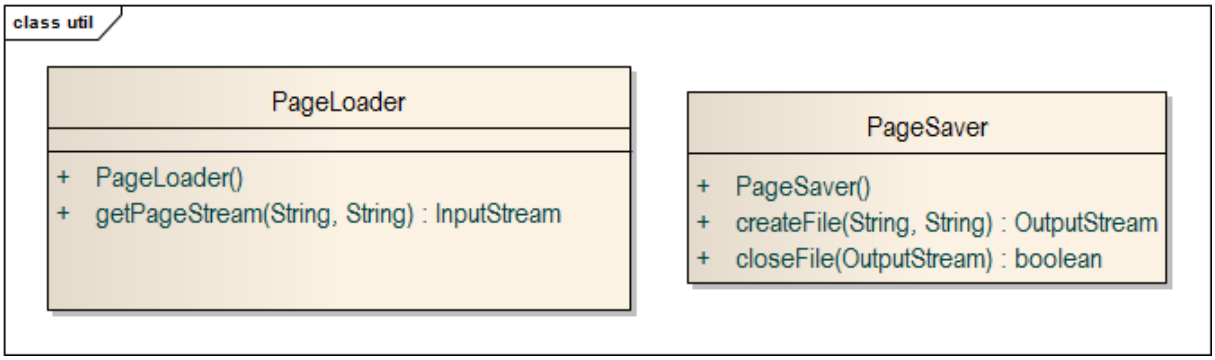
Implementácia

Aplikácia nepretržite zachytáva požiadavky nad Soketom.

```
Socket remote = proxyServer.listenForRequest();
```

Po zachytení požiadavky ju odošle ako *InputStream* na uloženie triede *PageSaver*. Táto trieda obsahuje metódu na uloženie *createFile()* a ukončenie ukladania *closeFile()*.

Na nastavenie súboru ako *InputStream*-u pre čítanie z SD karty, je vytvorené trieda *PageLoader*.



Obr. 2.6 balík ownet.aplication.util

PageSaver

- *createFile()* – metóda vytvorí adresáre podľa stromovej štruktúry a do súboru nastaví ukladanie *InputStream*-u.
- *closeFile()*– metóda, po prečítaní celého *InputStream*-u, ho vyprázdni a následne uzavrie.

PageLoader

- *getPageStream()* – metóda vyhľadá požadovaný súbor v stromovej štruktúre a vytvorí *InpuStream* nad koncovým súborom.

2.5 SD card Access a databáza na nej

Analýza

Hlavným kritériom pri výbere databázy bola skutočnosť, aby bola vybraná databáza podporovaná platformou Android. Táto požiadavka odstránila z našej palety často používané databázy ako napr. Oracle a MongoDB.Pri analýze dostupných databáz sme sa tak zamerali najmä na Couchbase a SQLite.

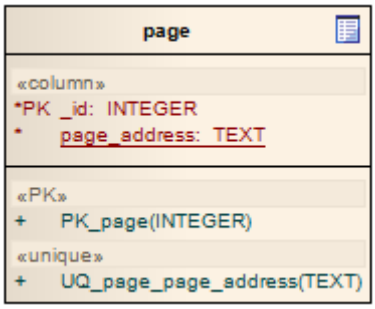
Couchbase je databáza, ktorá obsahuje CouchDB, pričom je obohatená o cache-ovanie. Ide o dokumentovo orientovanú databázu ajej hlavnou výhodou je absencia problémov spojených s ORM (angl. object relation mapping). Ďalšou podstatnou výhodou je jej RESTful API, ktoré je prívetivé pri komunikácii mobilného zariadenia so serverom. Okrem iných vlastností podporuje auto-sharding a má nulový prestoj pri údržbe.

SQLite je SQL databáza bez serveru a bez konfigurácií. Údaje ukladá priamo do pamäti zariadenia. Databázový formát súborov jenavyše platformovo nezávislí, čo umožňuje prenosnosť údajov medzi jednotlivými zariadeniami bez ohľadu operačné systémy. SQLite má nízke zaťaženie hardvéru, čo v prípade využitia pri mobilnej aplikácii považujeme za veľmi dôležité.

Návrh

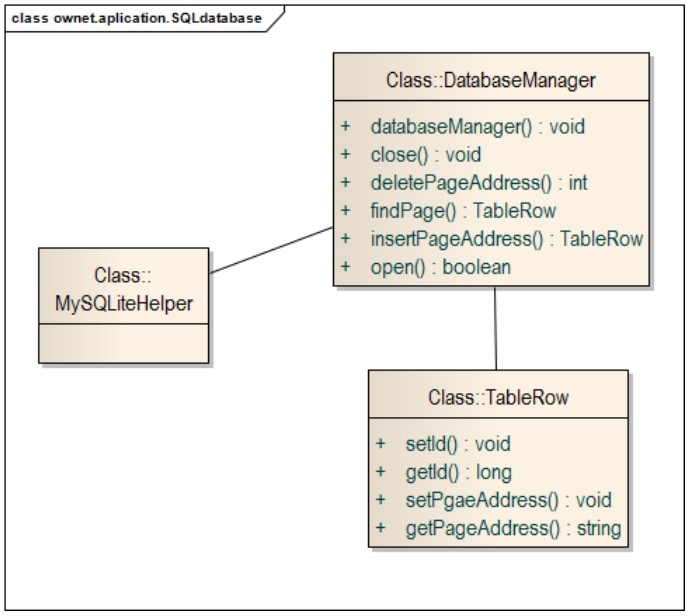
Obe vyššie popísané databázy umožňujú zapisovanie údajov do externej pamäti. Pre našu aplikáciu potrebujeme databázu, ktorá nám poskytne rýchle informácie o tom aké stránky máme uložené na SD karte. Z dôvodu nízkych pamäťových nárokov a jednoduchosti použitia sme sa rozhodli pre SQLite.

Vytvorili sme prvotnú podobu tabuľky pre súčasné potreby. Pre naše súčasné potreby sme potrebovali uložiť stiahnutú stránku. Po dohode v rámci tímu sme rozhodli neukladať informácie o tom, kde je stránka uložená, pretože ju budeme ukladať na kartu v stromovej štruktúre podľa štruktúry linku danej stránky.



Obr. 2.7 Tabuľka pre ukladanie stránok

Implementácia



Obr. 2.8 balík ownet.aplication.SQLdatabase

DatabaseManager

- *open()* - metóda nadviaže spojenie s databázou, ak neexistuje tabuľka *Page*tak ju vytvorí. Nemá žiadny vstupný parameter, ak sa podarilo nadviazať spojenie vracia true, v opačnom prípade false.
- *close()* – metóda uzavrie spojenie s databázou, nemá žiadny vstupný ani výstupný parameter.
- *insertPageAddress()* – metóda vloží adresu stránky do databázy, vráti objekt, ktorá predstavuje jeden riadok tabuľky. Tento objekt má štandardné set a get metódy pre každý stĺpec tabuľky. Ak sa nepodarí vložiť záznam vráti null.
- *deletePageAddress()* – metóda vymaže stránku, vstupný parameter je adresa stránky. Ak nájde rovnakú adresu v tabuľke vymaže záznam a vráti 1, inak -1.
- *findPage()* – metóda na základe adresy stránky(vstupného parametra), vráti objekt, ktorý predstavuje jeden riadok v tabuľke. Tento objekt má štandardné *set* a *get* metódy pre každý stĺpec tabuľky. Ak sa nepodarí vložiť záznam, vráti null.

3 Rajčiaková polievka

Po úspešnom ukončení prvého šprintu sa nám podarilo vytvoriť beta verziu aplikácie, ktorá bola schopná stiahnuť obsah stránky s očakávanou štruktúrou a neskôr ju načítať a zobraziť offline. Aplikácia však fungovala, len v emulátore a nie na mobilnom telefóne alebo tablete. Ďalším problémom bol aj slabý výkon(performance) aplikácie.

Cieľom druhého šprintu teda bolo:

1. Odhaliť, prečo aplikácia nefunguje na reálnych zariadeniach, a to pomocou logovania aplikácie.
2. Vylepšiť doterajší spôsob sťahovania obsahu stránok na SD kartu, tak, aby aplikácia dokázala stiahnuť údaje z ľubovoľnej stránky.
3. Začať s realizáciou portálu, ktorého prvým cieľom bude umožňovať spravovať obsah stránok na SD karte, teda umožniť používateľovi ich prehliadanie a mazanie.
4. Paralelizovať spôsob sťahovania obsahu stránok, kvôli urýchleniu behu aplikácie.

3.1 Vytvorenie portálu

Analýza

Jednou z funkcií, ktoré musí poskytovať OwNet, je ajportál, ktorý musí poskytovať spravovanie uložených webových stránok.

Návrh

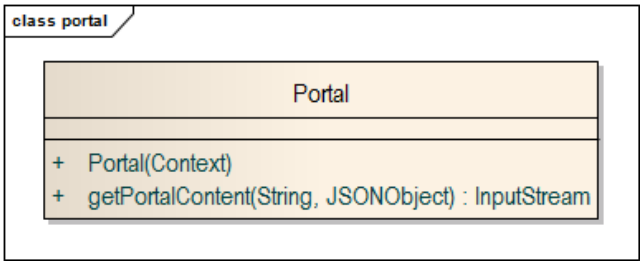
Funkcie portálu:

1. Zobrazenie zoznamu uložených stránok
2. Mazanie uložených stránok (odstránenie z databázy a z pamäte mobilného zariadenia)
3. Zobrazenie zoznamu najčastejšie navštevovaných stránok
4. Administrácia aplikácie
    - nastavenie veľkosti pamäte vyhradenej pre ukladanie stránok,
    - nastavenie spôsobu výberu obete pri prekročení kapacity vyhradenej pamäte

Implementácia

Ownet Portál sa skladá z dvoch častí:

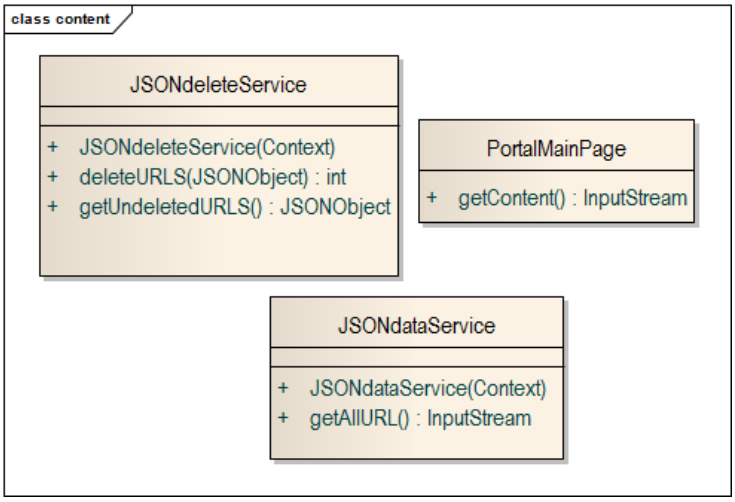
1) Java triedy nachádzajúce sa v balíku *ownet.aplication.portal.**
2) HTML stránky a JS skripty uložené v pamäti mobilného zariadenia



Obr. 3.1 balík ownet.aplication.portal

Portal

- *getPortalContent()* - metódazachytáva požiadavky smerované na portál. Ide o požiadavky,ktoré nejakým spôsobom pracujú z dátami z databázy.



Obr. 3.2 balík ownet.aplication.portal.content

JSONdataService

- *getAllURL()*- metóda vracia JSON súbor, obsahujúci zoznam URL adries uložených v databáze.

JSONdeleteService

- *deleteURLS()* – metóda vymaže URL adresu, ktoré boli vybrané používateľom na vymazanie z databázy.
- *getUndeletedURLS()*– metóda vracia zoznam URL adries, ktoré sa nepodarilo vymazať v metóde *deleteURLS()*.Tento zoznam spolus odpoveďou o výsledku mazania z databázy sa posiela späť na stránku portálu.

Formát JSON súboru:

```
{"posts": [
            { "url":"someUrlAddress.somethin"},
            { "url":"otherUrlAddress.somethin"},
```

```
            { "url":"thirdUrlAddress.somethin"},
            ]
}
```

JSON súbor neobsahuje URL adresy stránok, ktoré tvoria Portál (bezpečnostné opatrenie, aby ich používateľ omylom nevymazal).

Zoznam stránok a skriptov:

index.html predstavuje hlavnú stránku portálu.

urlList.html stránka zobrazuje zoznam URL adries uložených v databáze. Umožňuje výber a mazanie URL adries z databázy.

style.css základný css štýl stránok.

jquery.js obsahuje funkcie jQuery.

printData.js skriptobsahuje funkciu *printURLFromDB()*, ktorá zobrazujezískané URL adresy z databázy.

deleteURL.js obsahuje funkciu *deleteURL()*,ktorá vytvára JSON objekt obsahujúci URL adresy vybrané používateľom na zmazanie. Vytvorený JSON objekt je následne poslaný pomocou funkcie *jQuery.ajax*ako POST request naportál. Funkcia čaká na odpoveď portálu (mazanie bolo/nebolo úspešné), ktorá je zobrazená v podobe pop-up okna na stránke portálu. Pri neúspešnom mazaní posiela aplikácia JSON súbor obsahujúci zoznam adries, ktoré sa nepodarilo vymazať. Následne je tento JSON súbor rozložený a adresy, ktoré sa nepodarilo vymazať ostanú na stránke zaznačené.

json2.js skript slúži na transformáciu textu na JSON súbor.

Všetky html stránky a skripty sú uložené na SD karte mobilného zariadenia v adresári *ownet/pages/ownet.portal/*

3.1 Zachytávanie požiadaviek na portál

Na zachytávanie požiadaviek na portál slúži trieda s názvom *Portal.java* v balíku *ownet.aplication.portal*.Pomocou metódy*getPortalContent()* zachytáva určité URL adresy. Odpoveďou môže byť nový JSON súbor obsahujúci dáta z databázy, odpoveď o výsledku mazania údajov v databáze a iné, v závislosti od typu požiadavky.

Zachytávajú sa iba požiadavky, ktoré vyžadujú dáta z databázy, alebo s nimi priamo pracujú.Zachytávané požiadavky:

`http://ownet.portal/urlFromDB` klient žiada zoznam URL adries uložených v databáze

`http://ownet.portal/deleteURL` zasiela zoznam URL adries, ktoré majú byť z databázy vymazané.

3.2 Deduplikácia URL adries

Analýza

Táto časť projektu zabraňuje vzniku duplicitných adries a ich ukladaniu do databázy a na SD kartu zariadenia. Pokiaľ URL stránky obsahuje *www*, je možné tento prefix odstrániť a stránka je stále funkčná. Pokiaľ URL stránky prefix *www* neobsahuje, jej doplnenie spôsobí chybnú URL a jej obsah sa nenačíta. Teda prefix *www* nie je nutné používať, pretože si ho prehliadač a samotná požiadavka doplní sama.

Návrh

Deduplikáciu je potrebné zabezpečiť tak, že do databázy budeme ukladať všetky stránky bez prefixu www. Teda pred každou kontrolou či sa stránka v databáze nachádza alebo pri vkladaní do databázy bude prefix odstránený. Takto sa zabezpečí, že stránka nebude dvakrát vložená ani do databázy, ani na SD kartu.

Je dôležité, aby požiadavka obsahovala prefix, keď sa posiela na HttpClient-a, pretože požiadavku bez prefixu www nevie správne vyhodnotiť.

Implementácia

Implementácia tejto funkcionality je súčasťou triedy *Manager*. Zabezpečuje odstránenie www z reťazca, ktorý je použitý na kontrolu v databáze, vkladanie do databázy či na SD kartu. Pre odstránenie prefixu www používame štandardné Java funkcie *contents()* a *replace()*, ktorými zistíme, či sa prefix v reťazci nachádza a následne ich nahradíme prázdnym reťazcom.

3.3 Alias a mapovanie dlhých názvov súborov

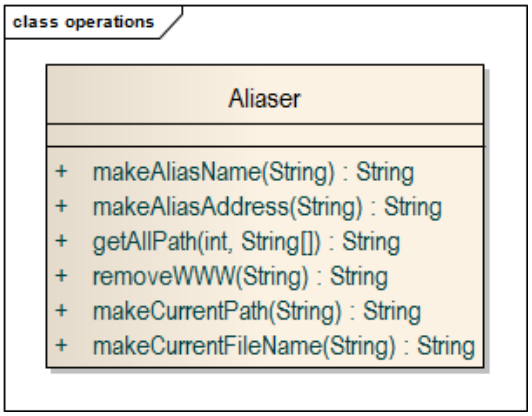
Analýza

Vieme, že dĺžka mena adresárov a súborov nie je neobmedzená. Taktiež mená súborov nemôžu obsahovať všetky znaky. Existujú špeciálne znaky ako | <> * ? / \ :, ktoré nemôžu byť súčasťou názvu. Musíme preto zabezpečiť, aby boli názvy adresárov a súborov korektné. Pokiaľ ide o názvy adresárov a súborov, tak u nich je určitá korekcia zabezpečená tým, že sú zaznamenané niekde na servery a pre ten platia presne tie isté obmedzenia, ktoré potrebujeme zabezpečiť. Problém môže vzniknúť pri pridávaní rôznych parametrov do URL adries, kde je štandardným znakom ? alebo meno súboru s viacerými bodkami. Takéto súbory nie sú korektné a preto je ich potrebné previesť do korektného tvaru.

Návrh

Zvoliť vhodný spôsob pre vytváranie názvov súborov a adresárov, ktoré nemajú korektný názov, tak, aby nevznikla chyba pri ich ukladaní na SD kartu.

Implementácia



Obr. 3.3 balík ownet.aplication.operations

Aliaser

- *makeAliasAddress()* – vstupným parametrom metódy je adresa stránky, ktorá neobsahuje prefix www. Následne z nej odstráni aj prefix http://. Z takto vytvorenej adresy pomocou metódy *makeAliasName()* vytvorí regulárny názov súboru a pomocou metódy *getAbsolutePath()* vytvorí celú stromovú štruktúru adresárov. Výstupom funkcie je vytvorená cesta spolu s názvom súboru do ktorého sa uloží obsah z *InputStream*-u.
- *makeAliasName()* – vstupným parametrom je pôvodný názov súboru. Výstupom je regulárny názov súboru, ktorý neobsahuje žiadny zo špeciálnych znakov menovaných v analýze a ani nie je neprimerane dlhý. Pokiaľ názov nie je regulárny, metóda vytvorí nový názov, ktorý je tvorená prefixom alias_ a systémovým časom v nanosekundách. Tento parameter je dostatočne presný na to, aby nedochádzalo k vytváraniu rovnakých mien. Na koniec názvu je potrebné pridať príponu.

```
alias = "alias_" + nanoStamp +".txt";
```

- *getAbsolutePath()* – vstupnými parametrami sú všetky názvy adresárov vytvorené z adresy vloženej do metódy*makeAliasAddress()* a počet týchto názvov. Metóda na výstupe vracia regulárnu cestu s pridanými oddeľovačmi /.
- *makeCurrentPath()* – vstupným parametrom je reťazec obsahujúci adresu bez prefixu www a http://. Na výstupe metóda vracia cestu ku súboru s jeho názvom.
- *makeCurrentFileName()* - vstupným parametrom je reťazec obsahujúci adresu bez prefixu www a http://. Na výstupe metóda vracia názov konečného súboru cesty.

3.4 Problém s index.hmtl

Analýza

Vieme, že pri zadávaní URL adresy bez názvu koncového súboru (napr. už www.azet.sk) nám prehliadač doplní ako koncový súbor najčastejšie niečo ako index.html, alebo index.php, alebo niečo úplne iné. To čo nám prehliadač navolí ako koncový súbor je vybrané podľa nastavení tvorcov konkrétnej webovej stránky. Bolo by teda vhodné, aby sme sa vyhli duplicitám v databáze a teda aj na SD karte zariadenia.

Avšak nevieme určiť, či adresa bez koncového súboru, bude mať ten istý obsah ako adresa s koncovým súborom. Preto riešenie duplicít nie je možné, pretože by sme takýmto prístupom prišli o údaje na stránkach.

Pokiaľ koncový súbor nebude existovať, nie je možné mu priradiť obsah. Preto musíme navrhnúť spôsob, ako vytvoriť meno koncového súboru, pod ktorým sa bude ukladať do databázy a na SD kartu.

Návrh

Nakoľko nevieme zabezpečiť, že obsah stránky bez koncového súboru a s ním, má totožný obsah, nie je možné ich rozdeľovať a hovoriť o duplicitách. Všetky adresy sa preto budú ukladať na SD kartu a databázy bez ohľadu na ich tvar, pokiaľ po úprave budú regulárne(trieda Aliaser).

Pre adresy bez koncového súboru bude názov koncového súboru vytvorený.

Implementácia

Duplicite adries v databáze sa predchádza zadefinovaním jej štruktúry. Stĺpec tabuľky obsahujúci adresy a aliasy (cesta k súboru na SD karte) musí byť unikátny. Pokiaľ tomu tak nebude, nie je uložený do databázy.

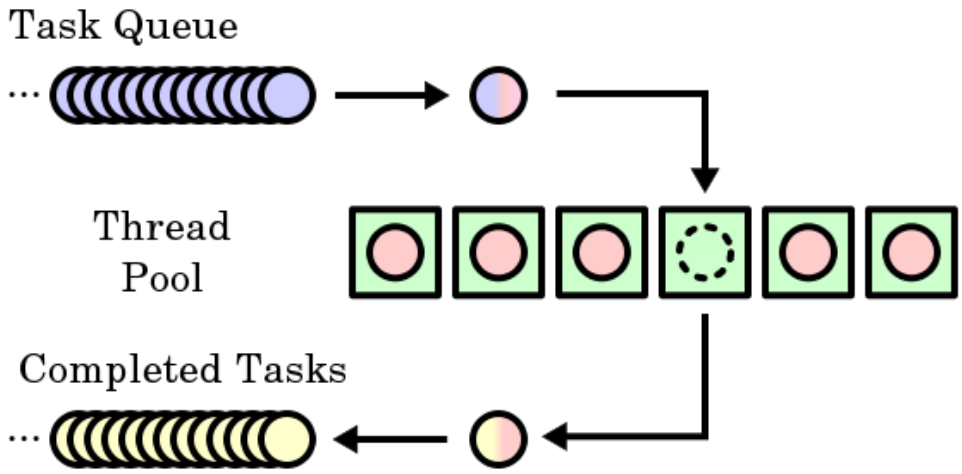
Trieda *Aliaser* obsahuje metódu makeAliasName(), ktorá zabezpečuje vytvorenie názvu koncového súboru ak neexistoval v ceste adresy. Meno súboru je vytvorené na základe pravidiel, ktoré sme popísali v časti dokumentu *3.5*.

3.5 Paralelizmus

Analýza a návrh

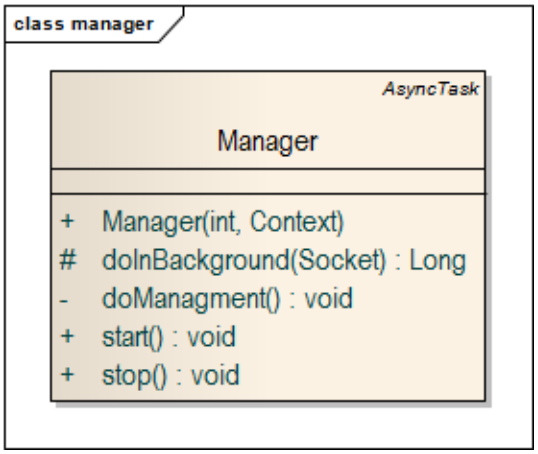
Z dôvodu pomalého načítavania stránok je nutné načítanie vykonávať paralelne. To znamená, že sa jednotlivé časti stránky budú načítavať naraz a nie postupne.

Navrhnuté riešenie je realizované prostredníctvom tzv. *thread pool patternu*. V tejto metóde sa vytvorí dopredu stanovený počet „robotníkov“ vlákien. Každý robotník si vypýta adresu ktorú, je potrebné vybaviť. Potom následne vykoná všetky úkony spojené s touto adresou. Po vytvorení všetkých robotníkov, hlavné vlákno čaká na dokončenie práce všetkých robotníkov. Keď všetci vykonajú svoju prácu cyklus sa opakuje prideľovaním ďalšej práce.



Obr. 3.4 Thread Pool Pattern

Implementácia



Obr. 3.5 balík ownet.aplication.manager

Pri implementácií bolo potrebné upraviť triedu *Manager*. Bola vytvorená metóda *doManagment()*. V tejto metóde sú všetky úkony, ktoré musí vykonať robotník s adresou. Túto metódu vykonáva každý robotník samostatne.

3.6 Preveriť správanie SD karty + logovanie aplikácie

Analýza a návrh

Prvým cieľom tejto úlohy bolo zistiť príčinu zlyhávania aplikácie(verzia Obzerance) pri práci s SD kartou. Keďže tento problém bolo treba čo najskôr analyzovať a odstrániť, tak sme pre získanie potrebných informácií zvolili možnosť využitia aplikácie CatLog. Táto aplikácia nám umožňuje spustiť zapisovanie všetkých logov do súboru. Toto riešenie aplikovali kvôli odstráneniu súčasného problému, avšak pre ďalší vývoj aplikácie bude potrebné navrhnúť a realizovať systém logovania zachytávajúci len logy aplikácie(a nie emulátora, prípadne iného zariadenia).

Druhým cieľom bolo navrhnúť a realizovať kompletné logovanie našej aplikácie.

- prvou možnosťou bolo použitie triedy *Log*, ktorá je určená na logovanie pre platformu Android. Táto možnosť však neumožňuje vytvoriť logovací súbor. Logy sme tak mohli sledovať len pomocou vyššie opísanej aplikácie *CatLog*.
- druhou možnosťou bolo použitie štandardnej triedy*java.util.logging.logger*.
- treťou možnosťou bolo použitie knižnice *log4j*.

Vybrali sme si štandardnú triedu *logger.java*, ktorá spĺňala naše požiadavky vytvorenia logovacieho súboru. Oproti log4j bola jednoduchá a ľahko konfigurovateľná.

Implementácia a testovanie

Pri riešení tejto úlohy sme dospeli k záveru, že naša aplikácia padala pri pokuse uložiť súbor, ktorého názov začínal „?“. Taktiež sme pri implementácii vytvorili triedu *MyLogger* a zalogovali sme jednotlivé triedy.

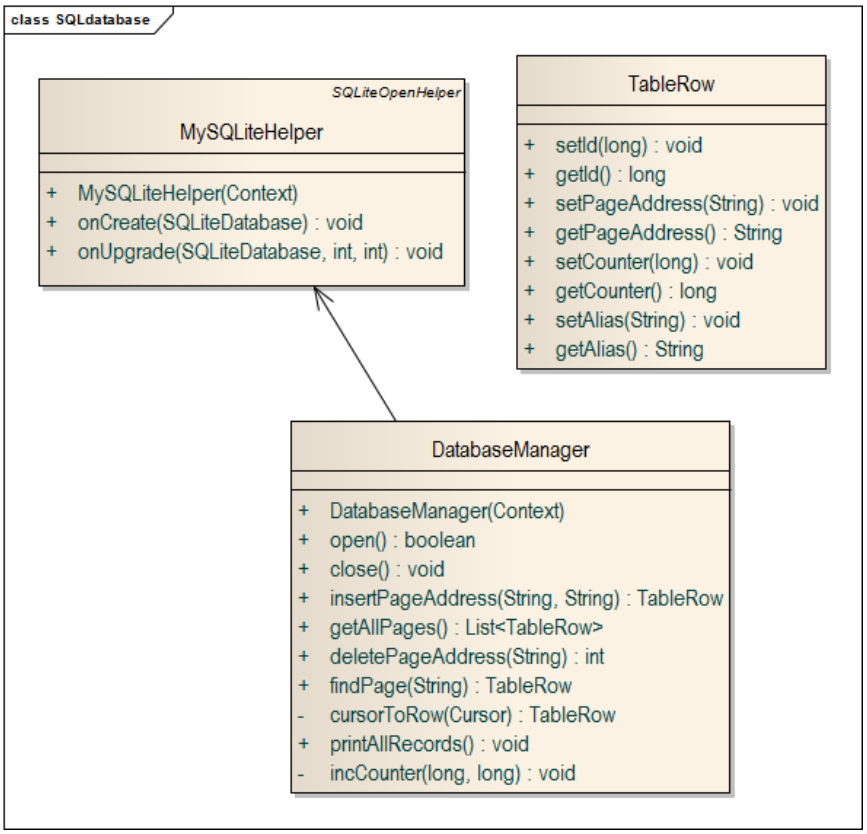
3.7 Access log

Analýza

Cieľom tejto úlohy bolo umožniť zaznamenávať počet prístupov na danú stránku, pre potreby budúceho inteligentného prednačítavania stránok, ale aj pre algoritmus výberu stránok na odstránenie. Po krátkej úvahe sa náš tím rozhodol pre zmenu v databáze. Ako najlepšie riešenie sme vybrali možnosť rozšírenia tabuľky, v ktorej sú uložené informácie o uložených stránkach. Rozhodli sme sa pre možnosť, kde bude pre každú stránku atribút, ktorý bude predstavovať počet prístupov.

Implementácia

Rozšírili sme tabuľku o dva stĺpce. Jeden stĺpec slúži práve pre rátanie prístupov na stránku. Nazvali sme ho *counter*. Počítanie prístupov sme vyriešili upravením metódy *findPage()*. Táto metóda slúžila práve pre vyhľadanie stránky v prípade dopytu. Do tela metódy sme pridali volanie metódy *incCounter()*, ktorá zabezpečí rátanie. Pri implementácii sme pridali aj druhý stĺpec, nakoľko sme počas šprintu identifikovali jeho potrebu. Tento stĺpec slúži ako kratší alternatívny názov k pôvodnému názvu súboru. Túto požiadavku sme identifikovali pri ukladaní súborov, kde sa vyskytol problém s maximálnou dĺžkou názvu súboru. Kvôli nasledovným zmenám sme museli upraviť aj metódu *insertPage()*.



Obr. 3.6 balík ownet.aplication.SQLdatabase

Pre prácu so súčasnou tabuľkou sme, vytvorili či zmenili metódy *findPage(), incCounter()*a *insertPage()*.

DatabaseManager

- *findPage()* – na základe adresy stránky, ktorá príde ako vstupný parameter, vráti objekt, ktorý predstavuje jeden riadok v tabuľke a zavolá metódu *incCounter()*. Tento objekt má štandardné set a get metódy pre každý stĺpec tabuľky. Ak sa nenájde záznam vráti null.
- *insertPageAddress()* – vloží adresu stránky do databázy, vráti objekt, ktorá predstavuje jeden riadok tabuľky. Tento objekt má štandardné set a get metódy pre každý stĺpec tabuľky. Ak sa nepodarí vložiť záznam vráti null. Vstupné parametre sú adresa stránky a jej alias. Hodnotu pre stĺpec *counter* vyplní implicitne hodnotou 0.

- *incCounter()* – metóda upraví záznam, ktorého *id* dostane ako vstupný parameter. Ďalším vstupným parametrom je hodnota v stĺpci *counter* pre daný záznam. Túto hodnotu inkrementuje a prepíše ňou aktuálnu hodnotu v databáze.

4 Fazuľový prívarok

V tomto šprinte sme sa na začiatku sústredili na odstránenie chýb, nakoľko jednotlivé časti aplikácie z predošlého šprintu sa nepodarilo úplne integrovať. Okrem toho sme sme mplementovali aj novú funkcionalitu ako doplnenie možnosti konfigurácie aplikácie, prostredníctvom gui rozhrania a zrýchlenie aplikácie pomocou implementácie thread poolu

4.1 HttpClient

Analýza

Aplikácia Ownet musí primeraným spôsobom vybaviť všetky prichádzajúce požiadavky. Ak sa požadovaná webová stránka alebo jej element nenachádza v databáze, musí aplikácia stiahnuť danú stránku z internetu. Java API a Android poskytujú dve hlavné triedy na získavanie webového obsahu:

- *URLConnetion* v balíku `java.net`
- *HttpClient* v balíku`org.apache.commons`

Obidve implementácie poskytujú približne rovnaké funkcie na získavanie obsahu webových stránok, pridávanie hlavičiek a nastavenia parametrov pripojenia.

Návrh

Na vývoj klienta bola vybraná implementácia HttpClient, pretože poskytuje prepracovanejšie API a väčšie množstvo funkcií. Klient vytvára prostredníka medzi internetovým prehliadačom a webovým serverom. Vytvorená požiadavka na webový server musí okrem typu metódy (GET, POST) obsahovať aj hlavičky z pôvodnej požiadavky, ktorú prijala trieda ProxyServer.

Vďaka pripájaniu parametrov http hlavičiek môžu webové servery lepšie odpovedať na požiadavky klienta (webový server získa informáciu o tom, s kým komunikuje – napr. mobilné zariadenie, tablet, atď.).

Ukážka hlavičky http požiadavky:

```
GET http://www.youtube.com/ HTTP/1.1
User-Agent: Opera/9.80 (Windows NT 6.1; U; sk) Presto/2.10.289 Version/12.02
Host: www.youtube.com
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png,
image/webp, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: sk-SK,sk;q=0.9,en;q=0.8
Accept-Encoding: gzip, deflate
Cookie: PREF=f1=10000000&fv=11.4.402; VISITOR_INFO1_LIVE=xw0S4E66vXY
Proxy-Connection: Keep-Alive
```

Http klient musí preposlať http hlavičku z prijatej odpovede, aby internetový prehliadač mohol správne interpretovať prijatú webovú stránku.

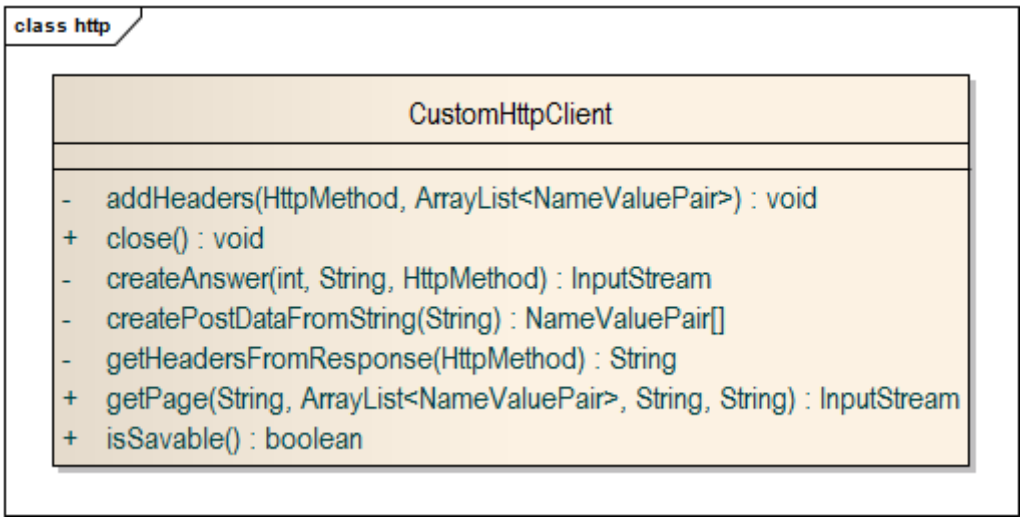
Ukážka hlavičky http odpovede:

```
HTTP/1.1 200 OK
Date: Mon, 03 Dec 2012 23:00:55 GMT
Server: Apache
X-Content-Type-Options: nosniff
Set-Cookie: NO_MOBILE=; path=/; domain=.youtube.com; expires=Thu, 01-Jan-1970
00:00:00 GMT
Set-Cookie: s_gl=af7554ba0c8e9bd5f37b21f5edfe8013cwIAAABDWg==; path=/;
domain=.youtube.com
Expires: Tue, 27 Apr 1971 19:44:06 EST
Cache-Control: no-cache
X-Frame-Options: SAMEORIGIN
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
X-OSSProxy: OSSProxy 1.3.335.312 (Build 335.312 Win32 en-us)(Oct 25 2012 23:16:24)
Connection: keep-alive
```

Vo výnimočných prípadoch (strata internetového spojenia, nedostupnosť webového servera, a iné) je potrebné vytvoriť fiktívnu http hlavičku obsahujúcu niektoré základne parametre.

Implementácia

Implementácia http klienta je obsiahnutá v triede `ownet.aplication.http.CustomHttpClient.java`.



Obr. 4.1 balík ownet.aplication.http

Metódy:

- *addHeaders()* – metóda pripája k vytvorenej http požiadavke (*HttpMethod*) hlavičky prijaté z internetového prehliadača. Hlavičky sú reprezentované ako dvojice *NameValuePair* v *ArrayListe*.

- *getPage()* – metóda slúži na získanie obsahu webovej stránky. Vstupnými parametrami sú url adresa stránky, hlavičky z pôvodnej požiadavky, typ metódy (POST, GET) a dáta, ak sa jedná o metódu typu POST. Výstupom je InputStream obsahujúci hlavičku a telo získanej webovej stránky.
- *getHeadersFromResponse()* – extrahuje hlavičky z http odpovede.
- *createAnswer()* – metóda pripája k získanej webovej stránke http hlavičky z odpovede.
- *createPostDataFromString()* – metóda slúži na naformátovanie dát, ktoré sa posielajú pri POST metóde. Dáta sú parsované a ukladané do tvaru: *parameter=value*
- *isSavable()* – vracia hodnotu *true* alebo *false*. Táto hodnota sa nastavuje v závislosti od toho, či sa má získaná stránka uložiť do pamäte alebo nie. Metóda vracia hodnotu *true*, keď je návratový kód z webového servera *200 OK*. V opačnom prípade je nastavená hodnota *false* a stránka by sa nemala uložiť (napr. návratový kód *404 Not Found*).
- *close()* – metóda slúži na uzatvorenie InputStream-u, ktorý obsahuje získanú webovú stránku.

4.2 Cache sweeper

Analýza

Vytváraním množstva záznamov a stiahnutím viacerých stránok sa ručné mazanie stalo nepraktickým. Práve preto sme potrebovali vytvoriť proces, ktorý vymaže väčšie množstvo nazbieraných stránok. Tento proces je dôležitý aj z hľadiska nastavenia hranice akú veľkú pamäť má k dispozícii naša aplikácia.

Návrh

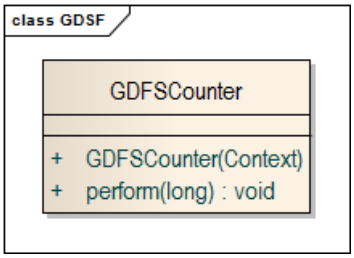
Po konzultácii sme sa rozhodli pre implementovanie GDFS(Greedy-Dual-Size-Frequency) algoritmus pre vymazávanie nahromadených stránok. Algoritmus využíva nasledovný vzorec pre výpočet váhy každej stránky uloženej našou aplikáciou.

Vaha = Clock + PocetPristupov x 100 / ln(Velkost + 1.1)

- *Clock - je váha posledne vymazaného záznamu.*

Implementácia

Pre implementáciu daného algoritmu sme vytvorili triedu *GDFSCounter*. Pred samotnou implementáciou sme museli upraviť databázu pridaním ďalšieho stĺpcu pre uloženie veľkosti súboru.



Obr. 4.2 GDFSCounter

GDFSCounter

- *perform()* - metóda, ktorá ma ako parameter veľkosť pamäte v byte-och koľko sa má vymazať. Táto metóda vypočíta pre každú uloženú stránku váhu a na základe vypočítaných váh vymaže potrebné množstvo uložených stránok.

GDFSCounter je spúšťaný priamo z aplikácie zadaním pamäte na vymazanie.

4.3 Fixnúť apku

Analýza

Po ukončení *Rajčinovej polievky*, sme konštatovali, že výsledná aplikácia nebola plne funkčná. Zapríčinil sa o to najmä fakt, že testovanie na emulátore dávalo odlišné výsledky ako testovanie na mobilnom telefóne. Z tohto dôvodu sme sa rozhodli zamerať viac na testovanie aplikácie a odhaliť tak chyby, ktoré nebolo možné identifikovať na emulátore, ale oj všetky iné.

Realizácia

Testovanie aplikácie mal na zodpovednosti Jozef Arpáš, ktorý zistené chyby študoval na základe logov, pomocou aplikácie *catlog.*Následne vytvoril bug v Jire, na základe presne definovanej štruktúry v dokumente riadeniaa formou SMS správy oznámil danému členovi tímu, že má priradený nový bug.

Programátor po preštudovaní zadania bugu v jire, buď hneď opravil chybu v zdrojovom kóde, alebo kontaktoval vedúceho tímu pre detailnejší opis chyby. Po opravení chyby, vrátil bug vedúcemu tímu na pretestovanie.

Zoznam identifikovaných bugov:

Bug: Nevytvorí sa service v telefóne, iba v emulátore

Riešenie: „Problém bol spôsobený vyťažením dostupných zdrojov mobilného zariadenia. Do repozitára sa dostala verzia projektu s prednastaveným počtom vlákien 20.Po znížení vlákien na počet 6, sa service opäť vytvorí a je možné používať aplikáciu.“

Bug: nevieme rozlíšiť napr. www.pokec.sk od pokec.sk, pričom toto by malo byť vyriešené v úlohe OWNET-38. Chyba sa vyskytuje v emulátore aj telefóne.

Riešenie: „V kóde to bolo implementovane, len to bolo v komentári, zrejme sme niečo testovali a zabudlo sa to dať do pôvodného stavu.“

Bug: aplikácia spadne na FileNotFound Exception pri outpustreame, kedy sa snažíme zapisovať do adresára. Chyba nastane napr. keď zadáme stránku dsl.sk, chyba sa vyskytuje aj na emulátore aj na telefóne.

Riešenie: „Bug bol odstránený...pri dopĺňaní funkcionality došlo k preklepu.“

Bug: nedokazeme vytvorit subor s regexp ".//."

Riešenie: „implementované“

Bug: po úspešnom načítaní stránky dsl.sk, som klikol na prvý článok, čo vyvolalo crash proxy service, v prílohe pripájam kompletný log padlo to na riadku:

```
if (!pageSaver.closeFile(fileOutputStream)){
```

Riešenie: „Problém spôsobil neexistujúci objekt pageSaver. Presunul som jeho vytvorenie do inej časti kódu. V repozitári je fixnutá verzia. Treba otestovať na mobile.“

Bug: Apka padla na NullPointerException v triede Manager.java (211).

Riešenie: „bug uzatvorený kvôli duplicite“

Bug: Nie je možné skrolovať obrazovku s nastaveniami aplikácie.

Riešenie: „presunuté do ďalších šprintov.“

Bug: Neobslúženie všetkých requestov

Riešenie: „apka (zatiaľ z neznámej príčiny) stiahla stránku, ktorú opera prezentovala ako text, tým pádom nemohlo dôjsť k ďalším requestom, pretože opera parsuje HTML a nie text. Favicon.ico sťahuje samotná Opera. Keďze URLConnetion nevracia odpoveď 404 ani ho umelo nevytvára (iba zavrie spojenie - socket), tak prehliadač opakuje stiahnutie daného elementu stránky určitý počet krát. Novy httpClient by to mal ošetriť.“

4.4 Producenti konzumenti

Cieľom tejto úlohy bolo vylepšiť aktuálny *threadpool* a navrhnúť a implementovať riešenie problematiky producenti konzumenti pre spracovanie streamov v našej aplikácii. Úspešná implementácia riešenia producenti konzumenti sa nám nepodarilo stihnúť v tomto šprinte.

Analýza threadpool-u

Po ďalšej analýze sme zistili že je potrebné zmeniť techniku *threadpool*-u aby jednotlivé vlákna na seba nečakali. Po implementácii v minulom šprinte sa vytvorilo N vlákien, ktoré čakali na seba až skončí posledný svoju prácu. Tento stav chceme prerobiť, aby tieto vlákná na seba nečakali a ako náhle skončí svoju úlohu, ukončí sa a vytvorí sa ďalšie vlákno, ktoré bude spracovávať ďalšiu úlohu.

Návrh threadpool-u

Na toto riešenie použijeme knižnicu *ThreadPoolExecutor*, ktorá tieto nedostatky odstraňuje a následne sme ju implementovali.

Implementácia threadpool-u

Pre implementáciu novej techniky *threadpool*-u bolo potrebné vytvoriť novú triedu *RequestWorker*. Táto trieda implementuje rozhranie *Runnable* aby ju bolo možne spustiť v samostatných vláknach. Do metódy *Run* z tohto rozhrania sme presunuli všetky operácie spojené so spracovaním požiadaviek od prehliadača. Tým sme zabezpečili, že jednotlivé vlákna, ktoré realizujú triedu *RequestWorker*, môžu spracúvať požiadavky paralelne v *threadpool*-e. Objekty triedy *RequestWorker* sa vytvárajú v metóde *threadPool.execute(),* po prijatí požiadavky

Obr. 4.3 Trieda Manager

Manager

- *doInBackground()* – metóda bola upravená tak, že časť spojenú so spracovaním presunuli do triedy *RequestWorker*. Metóda teraz len odchytáva požiadavky od prehliadača a posiela ich na spracovanie do *threadpool-uRequestWorkerom*.
- *Manager()* – konštruktor sa nezmenil.
- *start()* – metóda sa nezmenila.
- *stop()* – metóda sa nezmenila.

Analýza problematiky producenti konzumenti

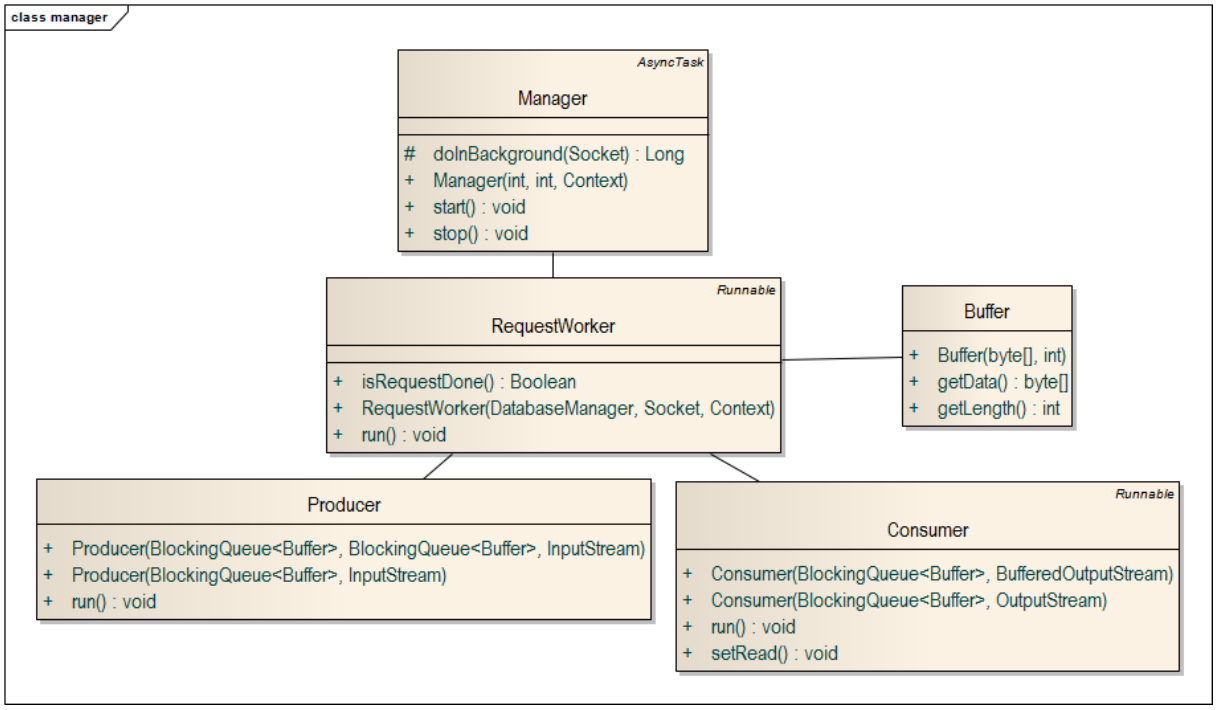
Aplikovať riešenie producenti konzumenti sme sa rozhodli na základe princípov akými funguje naša aplikácia. Najprv sťahovala stream z internetu a následne ho posielala opere a aj do súboru, ak sme danú stránku nemali stiahnutú. Keďže tento proces bol pomalý rozhodli, tak sme sa rozhodli použiť paralelizmus.

Návrh riešenia producenti konzumenti

V našom prípade ide o typický problém nazývaný producenti konzumenti. Pri posielaní údajov na proxy sme produkovali údaje, ktoré sme potom konzumovali posielaním opere, prípadne do súboru.Trieda *RequestWorker* spracováva jednotlivé požiadavky od opery. Samotné produkovanie údajova odosielanie údajov na proxy bude vykonávať trieda *Producent.* Táto trieda bude vytvárať triedy *Buffer*, ktoré budú predstavovať časť sťahovaného streamu. Konzumovať tieto údaje, čiže ukladať do súboru, alebo posielať opere bude trieda *Consumer*.

Implementácia riešenia producenti konzumenti

Pri implementácii sme použili triedy navrhnuté pri návrhu. Pričom sme využili pre knižnicu *LinkedBlockingQueue*, ktorá je implementovaná v jazyku Java. Do tejto rady vkladáme jednotlivé triedy *Buffer*. Táto rada je synchronizovaná a ide o typ FIFO. Veľkosť zdieľanej pamäti je 1024 bajtov. Správne nastavenie tejto hodnoty je vhodné otestovať. Pri sťahovaní stránky z internetu trieda *RequestWorker* vytvorí dve vlákna, ktoré sú implementované triedou *Consumer*. Táto trieda na základe vytvoreného konštruktora posiela spracované údaje opere, alebo ich ukladá do súboru. Následne trieda *RequestWorker* vytvorí triedu *Producent,* ktorý začne sťahovať stream. V prípade, že sťahujem z SD karty tak *RequestWorker* vytvorí nové vlákno implementované triedou *Consumer*, ktoré posiela časti streamu opere. Pričom *RequestWorker* vytvorí triedu *Producent,* ktorá bude vytvárať stream zo súboru, v ktorom je uložená odpoveď pre danú požiadavku.



Obr. 4.4 balík ownet.aplication.manager

Trieda Manager je opísaná vyššie v časti Implementácia threadpool-u.

RequestWorker

- isRequestDone() – vracia *true* ak je požiadavka vykonaná inak *false*.
- RequestWorker() – konštruktor, ktorý vykoná inicializáciu potrebných objektov.
- run() – metóda predpísaná rozhraním *Runnable*, vykonáva spracovanie jednej požiadavky.

Buffer

- Buffer() – konštruktor uloží údaje do atribútov tejto triedy.
- getData() – vráti údaje uložené v tejto triede.
- getLenght() – vráti dĺžku údajov uložených v tejto triede.

Consumer

- *Consumer()* – konštruktor pre spracovanie údajov pre operu.
- *Consumer()* – konštruktor pre spracovanie a uloženie údajov do súboru.
- *run()* – metóda predpísaná rozhraním *Runnable*, posiela streamopere resp. na ukladanie do súboru.

Producer

- *Producer()* – konštruktor pre produkovanie údajov pre spracovanie údajov operou.
- *Producer()* – konštruktor pre produkovanie údajov pre uloženie do súboru.

- *run()*–metóda predpísana rozhraním *Runnable*, vykonáva sťahovanie údajov z internetu resp. súboru.

Táto implementácia nie je plne funkčná. V. Sudor predpokladá, že tam sú možné dve chyby:

1. Pri sťahovaní údajov stále ukladáme do referencie toho istého objektu, čiže si údaje prepisujeme.
2. Nastane blokujúce čakanie v triede *Consumer* spôsobené tým, že v rade nie sú žiadne údaje a on čaká dovtedy kým neprídu. Avšak *Producer* už prestal vytvárať údaje takže tam nikdy ani neprídu, čo má za následok nekonečné čakanie.

5 Vyprážaný syr

Tento šprint bol naplánovaný na refactoring a tvorbu Junit testov. Keďže sa tento plán sa počas semestra nezmenil, posledný šprint sme sa naozaj zamerali na kompletný refactoring a tvorbu testov.

5.1 Refactoring

Jednotlivé balíky zdrojového kódu sa rozdelili medzi členov tímu, ktorí ich mali za úlohu refactorovať. Ich dosiahnuté výsledky kontroloval Jaro Rais ako manažér kvality, ktorí s nimi prípadné nedostatky konzultoval.
Michal Roško – `ownet.aplication.portal, ownet.aplication.GSFD`
Jozef Arpáš – `ownet.aplication.util`
Jaroslav Rais – `ownet.aplication.andnet`
Marek Lóderer – `ownet.aplication.proxy, ownet.aplication.http`
Vladimír Sudor – `ownet.aplication.manager`
Pavol Ružička – `ownet.aplication.sqldatabase`

5.2 Testovanie

Časť tohto šprintu bol zameraný na refaktorovanie a písanie JUnit testov pre vzniknuté triedy.

Analýza

Nakoľko Android platforma je postavená na programovacom jazyku Java rozhodli sme sa prepísanie JUnit testov. Tento typ testov je vhodný na kontrolu funkčnosti jednotlivých častí zdrojového kódu. Poskytuje nám možnosti pre písanie rôznych testov na všetky metódy a ichpod časti v Android aplikácií.Pre potreby riadneho testovania potrebujeme vytvárať takzvané mokovacie triedy, objektya dáta. Každému takémuto objektu vieme povedať, akú hodnotu má vracať, resp. čo má obsahovať a ako sa bude správať. Pre potreby mokov je možné použiť knižnicu EasyMock,jMock, Androidmock. Všetky tieto knižnice nám poskytujú zaručené možnosti pre vytváranie mokovaných dát. Najviac používaným je EasyMock, avšak aj jMock alebo Androidmock másvoje postavenie.

Všetky knižnice nám poskytujú rôzne metódy pre nastavenie testovacieho prostredia a teda zaručenie rovnakého prostredia pre každý vykonávaný test. Testy je možné zoskupovať do väčších časti TestSuite, ktorých môžeme zadefinovať, ktoré triedy testov budú spustené. Samozrejmosťou je, že v každej testovacej triede je možné predpísať testovacie metódy. Každá takáto trieda musí dediť od triedy TestCase a každá testovacia metóda musí vyhadzovať výnimku(throwsThrowable).

Návrh

Nakoľko jMock a EasyMock nevedia priamo pracovať Android platformou, zvolili sme knižnicu Androidmock a metódy, ktoré poskytuje JUnit3 a JUnit4. Vďaka knižniciAndroidmockujeme triedy, metódy a dáta tak, aby nám vrátili také údaje aké očakávame,resp. aby sa správali ako potrebujeme.

Pre písanie testov je vhodné dodržiavať určitú štábnu kultúru zdrojového kódu. Tieto pravidlású zapísané v dokumente riadenia v časti testovanie. Pravidlá a spôsob ako vytvárať testovací projekt, TestCase triedy a samotné testovacie metódy sú taktiež zapísane v dokumente riadenia v časti testovanie. Dodržiavanie týchto pravidiel nám poskytne prehľadnosť a ľahšie pochopenie častí zdrojového kódu. Všetky testy by mali byť na jednom mieste, aby boli dostupné všetkým členom, vývojárom a testerom v tíme. Umiestnenie testov bude v hlavnom projekte v zložke test.

Implementácia

V pôvodnom projekte, v ktorom je vytváraná aplikácia bol založený priečinok tests. Do tohto priečinku budú ukladané všetky testy, ktoré vzniknú, aby mal každý programátor, tester alebo vývojár prístup k týmto testom. Programátori môžu medzi sebou zdieľať testy, používať rozhrania, ktoré vytvorili iný programátori a takto si ušetriť prácu v ďalšom vývoji samotných testov, ale aj aplikácie.

Bolo vytvorených niekoľko JUnit testov na overenie správnosti fungovania metód aplikácie.

Balík	**Trieda**	**Metóda**	**Počet testov**
ownet.aplication.operations	Aliaser	containForbidenChar()	11
		makeAliasName()	7
		removeSlash()	6
		removeWWW()	2
		makeCurrentPath()	2
ownet.aplication.util	PageSaver	createFile	2
	PageLoader	getPageStream	2
ownet.aplication.proxy	RequestParser	getAddress()	2
		getMethod()	2
		getHeaders()	6
		isSavable()	2
		getContentLength()	1
		getPostData()	1
ownet.aplication.SQLdatabase	DatabaseManager	insertPageAddress()	1
		getAllPages()	1
		deletePageAddress	1

		findPage()	1
ownet.aplication.manager	Producer, Consumer	Problematika producenti konzumenti	1

Tabuľka 5.1 Prehľad počtu testov pre testované metódy

5.3 Youtube Get

Z dôvodu pretrvávajúcich problémov httpClienta, sme sa rozhodli vyskúšať iné API, ako to čosme pôvodne použili, teda android default httpclient.Na overenie sme sa rozhodli použiť „oficiálne“ API od apachu, pričom úloha bola získať response na GET na youtube.com. Toto sa nám podarilo a dostali sme odpoveď.

6 Mastný chľeba s cibuľou

Tento šprint bol zameraný veľmi rôznorodo a pracovalo sa na rôznych častiach aplikácie. Rozdelenie úloh jednotlivých členov bolo nasledovné:

ID	Popis úlohy	Zodpovedná osoba	Dátum vzniku	Stav úlohy
11.1	Jenkins na fedore	Lóderer Volentier	25.1.2013	nová
11.2	Portál pridanie bannera	Lóderer Roško	25.1.2013	nová
11.3	Protokol	Arpáš Ružička	25.1.2013	nová
11.4	Oprava konzumenta	Sudor	25.1.2013	Nová
11.5	Unset Opera settings	Rais	25.1.2013	Nová
11.6	UTF-8 Escapovanie	Sudor	25.1.2013	Nová
11.7	&=ref	Rais	25.1.2013	Nová

6.1 Analýza a návrh protokolu

Analýza

Primárnou úlohou protokolu má byť komunikácia medzi zariadeniami, ktoré majú nainštalovanú aplikáciu Ownet. Keďže pokračujeme v projekte, na ktorom pracujeme spolu s tímom č. 12 tak naše zariadenia musia byť kompatibilné s ich zariadeniami. Takže sme najprv skontaktovali druhý tím a naštudovali ich implementáciu protokolu. Následne sme sa dohodli na nových a povinných a nepovinných atribútoch, ktoré si budeme posielať.

Naším cieľom bude implementovať funkcionalitu, kde si klienti budú medzi sebou posielať informácie o tom kto má daný request. Taktiež klient bude musieť vedieť aký klienti sú na sieti pripojený, akú majú IP adresu a port, na ktorom počúvajú. Klienti budú musieť posielať svoje novinky ostatným a tieto novinky budú musieť spracovať.

Návrh

Pre výmenu údajov budeme používať JSON formát. Tieto JSON elementy sa budú posielať dvomi spôsobmi a to multicastom a cez TCP/IP.

Hello packet

Multicastom sa budú posielať tzv. „Hello packety“, kde každý klient bude posielať svoj identifikátor, port, typ zariadenia a skóre. Zoznam všetkých atribútov môžeme vidieť v tabuľke 5.1.

Názov atribútu	Typ	Povinný
JSON_type	String	nie
id	String	áno
score	Integer	áno
status	String	nie
port	Integer	áno
workspace_id	String	nie
workspace_name	String	nie
initialized	Boolean	nie
num_available_clients	Integer	nie
device	String	áno

Tabuľka 6.1 Multicast JSON.

- JSON_type – pre tento JSON je vždy „hello“.
- Id – Android zariadenia majú prefix A.
- Score – hodnota je z intervalu android < 10 < PC. Kto má najvyššie score je najvýkonnejší, čiže master.
- Status – nebudeme zatiaľ implementovať.
- Port – port, na ktorom počúva proxy.
- workspace_id – nebudeme zatiaľ implementovať.
- workspace_name – nebudeme zatiaľ implementovať.
- Initialized -nebudeme zatiaľ implementovať.
- num_available_clients – počet aktívnych klientov, nebudeme zatiaľ implementovať.
- Device - 1=PC; 2=tablet; 3=mobil.

Atribút score bol schválne navrhnutý tak, aby PC malo vždy väčšiu hodnotu a to z dôvodu výdrže batérie a výkonnosti jednotlivých zariadení. Batérie sú veľmi zaťažené pri používaní našej aplikácie.

Journal Request

Zvyšné JSON elementy sa budú posielať cez TCP/IP. Pôjde o „journalResponse“ a „journalRequest“.

Pomocou „journalRequest“ si klient pýta novinky od mastera, prípadne master od klienta.

Názov fieldu	Typ	Povinný
JSON_type	string	nie

client_id	string	áno
sync_all_groups	boolean	áno
group	string	áno
CLIENT_N	string	áno

Tabuľka 6.2 Žiadosť o novinky.

- JSON_type – pre tento JSON je to vždy „jrequest“.
- client_id – identifikátor klienta.
- sync_all_groups – pre android zatiaľ stále false, nebudeme zatiaľ implementovať.
- Group – pre android stále 10.
- CLIENT_N – názov tohto atribútu bude predstavovať klienta, od ktorého chceme zmeny a hodnotou bude identifikátor posledného žurnálového[3] záznamu.

Journal Response

Pomocou „journalResponse“ odpovedáklient mastrovi alebo master od klientovi ohľadom zmien v žurnálovej tabuľke daného klienta. Úlohou týchto jsnov je prezentovať zmeny jednotlivých klientov v ich žurnálovej tabuľke. Pre každý záznam sa odosiela samostatný JSON.

Názov fieldu	**Typ**	**Povinný**
JSON_type	string	nie
client_id	string	áno
Client_rec_number	Integer	áno
Table_name	String	áno
uid	String	áno
Operation_type	String	áno
Group_id	Integer	áno
Sync_with	String	áno
Date_created	String	áno
COLUMN_NAME_1	Typ1	áno
COLUMN_NAME_2	Typ 2	
COLUMN_NAME_3	Typ N	

Tabuľka 6.3 Journal Response

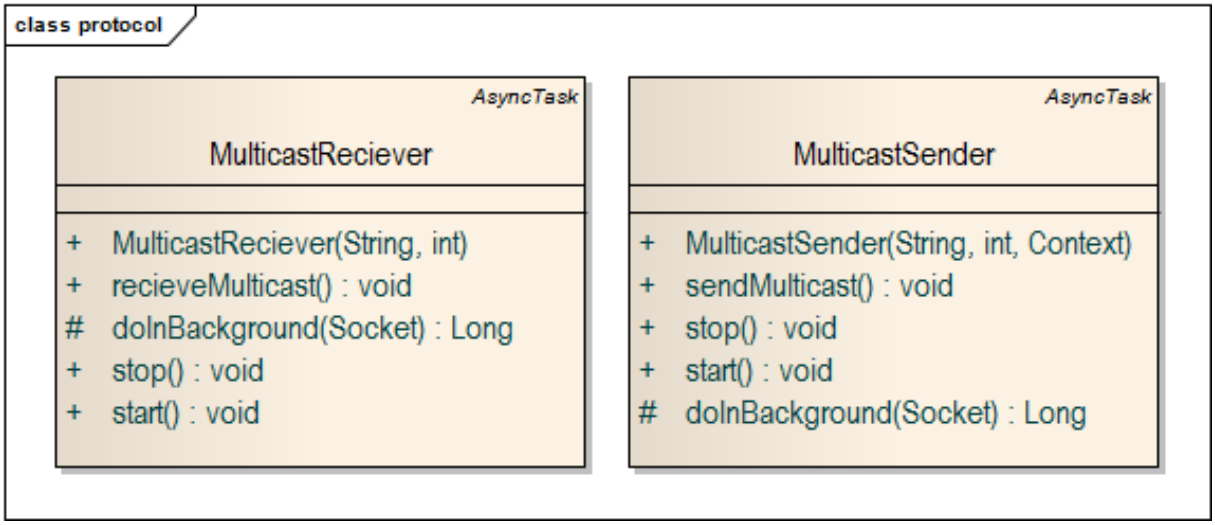
- JSON_type – pre tento JSON je to vždy „jresponse“
- Client_id – id klienta, ktorý posiela JSON
- Client_rec_number – id záznamu žurnálovej tabuľky
- Table_name–názov tabuľky s requestmi(pre android „client_caches“)

[3] Žurnál – je v skratke prehľad zmien vykonaných nad databázou.

- uid - unikátne ID záznamu, formát: "clientid_clientrecnum“
- operation_type – „INSERT/UPDATE/DELETE“
- group_id – id skupiny
- sync_with – id klienta, s ktorým sa synchronizujem
- date_created – dátum vloženia do databázy
- COLUMN_NAME_x – názov stĺpca requestovej tabuľky

6.2 Prototyp protokolu

V rámci overenia návrhu protokolu sme sa rozhodli implementovať zjednodušené posielanie a prijímanie Hello packetov. Pre odosielanie a prijímanie sme využili Multicast API dostupné pre Javu. Odosielané JSONy mali pevne definované hodnoty, ktoré predpokladala aj implementácia na strane prijímateľa.



Obr. 6.1 balík ownet.aplication.protocol

Boli vytvorené 2 nové triedy, ktoré boli implementované ako asynchrónne tasky, ktorý boli spustené po spustení *proxy listenera*.

MulticastSender

- *sendMulticast()* – každých 5 sekúnd odošle JSON tzv. *Hello packet* s pevne definovanou štruktúrou na multicastovú adresu.

MulticastReceiver

- *sendMulticast()* – príjíma a spracúva hello packety, pričom očakáva ich pevne daný obsah.

7 Palacinky

V tomto šprinte bolo cieľom implementovať kompletnú funkcionalitu protokolu pre vzájomnú komunikáciu zariadení.

7.1 Send Hello packetu

Analýza

Analýza k odosielaniu Hello packetov bola vykonaná v predchádzajúcom šprinte.

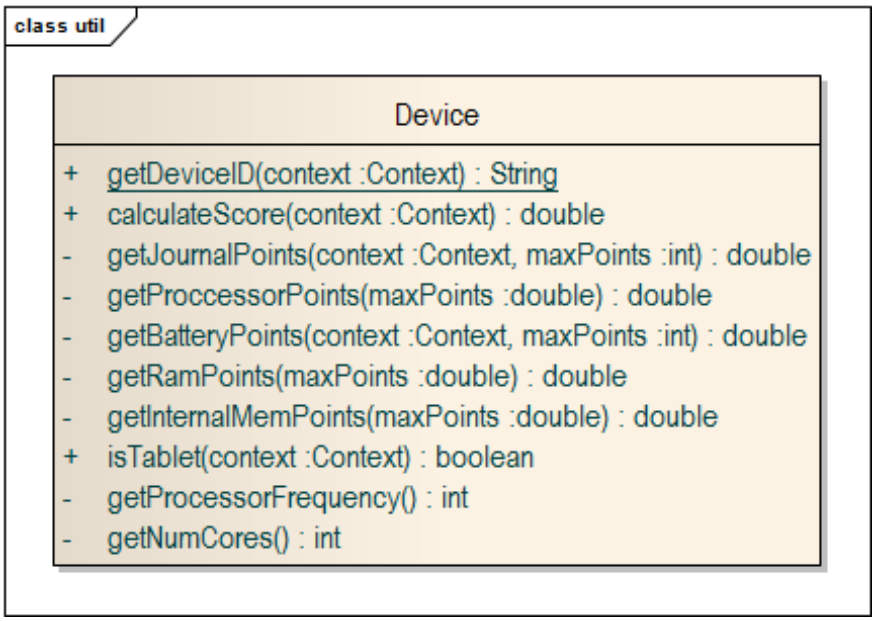
Návrh

Hello packety bude každé zariadenie odosielať pravidelne v 5 sekundových intervaloch na zvolenú multicastovú adresu. Štruktúra packetov je popísaná v kapitole 6.1. Pre výpočet skóre zariadenia sme zvolili nasledovné kritériá:

- Stav batérie(váha 3)
- Výkon procesora(váha 1,5)
- Veľkosť RAM pamäte(váha 1,5)
- Veľkosť voľnej internej pamäte(váha 2)
- Počet záznamov žurnálovej tabuľky(váha 2)

Maximálne skóre ktoré môže získať zariadenie platformy Android je vždy menšie ako číslo 10(z dôvodu odlíšenia od stolových a laptop počítačov).

Implementácia



Obr. 7.1 trieda ownet.aplication.util.Device

Device

- *getDeviceID()* – metóda vracia ID zariadenia, ktoré začína vždy písmenom „A“, za ktorým nasleduje zahešovaná mac adresa wifi zariadenia.
- *calculateScore()* – metóda vracia výsledné skóre pre dané zariadenia, podľa daných kritérií s pridelenými váhami.
- *getJournalPoints()* – určuje počet bodov za rozsah žurnálovej tabuľky.
- *getProccessorPoints()* – určuje počet bodov za výkon procesora. Do úvahy sa berie počet jadier a ich frekvencia.
- *getBatteryPoints()* – určuje počet bodov za aktuálny stav batérie. Ak má zariadenie nad 70% dostáva maximálny počet bodov za batériu.
- *getRamPoints()* – určuje počet bodov za veľkosť RAM pamäte.
- *getInternalMemoryPoints()* – určuje počet bodov za veľkosť dostupnej internej pamäte.
- *isTablet()* – vracia „true“ v prípade, že je dané zariadenie Tablet. Inak vracia „false“. Využíva sa pri napĺňaní flagu *device*.

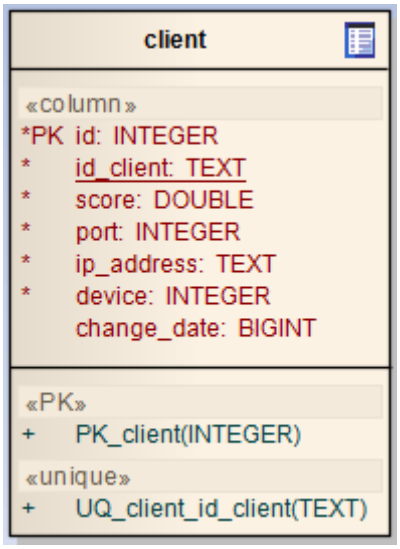
7.2 Reiceve Hello packetu

Analýza

Analýza bola vykonaná v minulom šprinte v rámci úlohy Analýza a návrh protokolu.

Návrh

Do súčasného prototypu pre prijímanie multicastov doplníme získanie IP adresy a údaje zapíšeme do databázy. V databáze vytvoríme tabuľku client. Túto tabuľku môžeme vidieť na obrázku 6.1.



Obr. 7.2 Tabuľka pre klientov

Implementácia

Pri implementácii bolo potrebné vytvoriť triedu *Client* do package *model*. Táto trieda predstavuje jeden riadok pre danú tabuľku. Následne bolo potrebné implementovať metódy do triedy *DatabaseManager*, ktoré budú vkladať jednotlivých klientov.

getAllClients()– vráti všetkých klientov z databázy.

insertClient() – vloží klienta na základe vstupných parametrov, vráti počet vložených záznamov.

updateClient() - aktualizuje klienta na základe id_client a vráti počet zmenených záznamov.

Tieto metódy sú volané po spracovaní jednotlivých *multicastov*. Po prijatí *multicastu* sa pokúsime aktualizovať klienta a ak sa nám vráti 0 aktualizovaných riadkov tak klienta vložím do databáze. V tabuľke je uložený ako klient aj používateľ aplikácie.

7.3 Čistič

Analýza

Cieľom tejto úlohy je odstránenie neaktívnych klientov. Títo klienti musia byť odstránení nielen z tabuľky *client*ale aj z *client_caches* a *journal*.

Návrh

ORMLite pomocou, ktorého pristupujeme k databázy nám neumožňuje vytvorenie príkazu „delete on cascade“, preto toto vymazávanie budeme robiť ručne.

Implementácia

Pri implementácii sme prekonali metódu *delete(Collection<Client> clients)* implementovanú pomocou *BaseDaoImpl*. V tejto metóde sme pre každý prvok kolekcie najprv vymazali záznamy vo vyššie spomínaných tabuľkách a až potom v tabuľke pre klientov. Pre realizáciu tohto riešenia bolo potrebné vytvoriť triedy *ClientDao*, *ClientsCachesDao* a *JournalDao*. Taktiež tieto zmeny bolo nutné zapracovať do triedy *DatabaseHelper*. Na obrázku 6.2 môžete vidieť diagram tried.

7.4 Požiadavky na master klienta

Analýza

Na základe požiadaviek bolo potrebné vytvoriť niekoho kto bude rozposielať údaje, novinky, dáta ostatným používateľom v sieti zariadení. Tento hlavný klient (master) má prehľad o dianí v sieti a posiela požadované dáta klientom, ktorý o ne požiadajú. Jeho význam je kľúčový nakoľko uvažujeme o distribuovanej sieti, kde dáta jedného budú dátami všetkých. Preto je úloha master tak dôležitá. Úloha master pri distribuovanej sieti má tieto hlavné časti:

- Prijímanie požiadaviek na novinky od ostatných klientov v sieti
- Posielanie aktualizácii v sieti
- Posielanie informácii o uložených dátach na jednom zariadení ostatným

Návrh

Podstatou tejto úlohy je aktuálnosť tabuľky journal každého klienta. Klient potrebuje vedieť o aktuálnych zmenách u každého klienta, aby vedel, kto mu aké možnosti v rámci distribuovanej pamäte ponúka. Odpoveď na túto otázku mu poskytuje master. Master je zariadenie, ktoré rozosiela žurnálové novinky každému klientovi, ktorý o ne požiada v podobe JSON správy obsahujúcej posledné zmeny všetkých klientov, ktoré má uložené v tabuľke journal. Odpoveďou master je JSON súbor obsahujúci všetky zmeny vyžiadaných klientov. Tieto zmeny si klient uloží do tabuľky journal a tabuľky client cashes. Požiadavku na zmeny posiela klient periodicky v nejakom časovom intervale.

Implementácia

Pre dosiahnutie stanovených cieľov popísaných v analýze a návrhu bolo potrebné vytvorenie triedy JournalReciever, ktorá zo strany klienta vysiela požiadavky v podobe JSON súborov a následne počúva odpoveď od master v podobe JSON súboru. Následne túto správu rozloží a dáta uloží do tabuľky journal a client cashes. Požiadavky na aktualizáciu sú posielané v pravidelných časových intervaloch.

Trieda *JournalReciever*obsahuje nasledujúce metódy:

- *JournalReciever()* – konštruktor triedy, preberá databázového manažéra a kontext zariadenia
- *sendRecieveJournal()* - metóda riadi komunikáciu medzi master a klientom. Klient pošle požiadavku a následne očakáva odpoveď. Po prijatí odpovedi, ju rozloží a vykoná potrebné zmeny v databázových tabuľkách journal a client cashes. Komunikácia je sprostredkovaná http klientom.
- *findMaster()* – metóda vyhľadá v databázovej tabuľke client používateľa s najväčším skóre, pretože client s najväčším skóre je vždy master.
- *setMaster()* – metóda nastaví globálne premenné IP adresu a port na hodnoty, aké má master.
- *buildJSONToMaster()* – metóda vytvára JSON súbor obsahujúci požiadavky na aktualizáciu databázových tabuliek journal a client cashes.
- *createJSONFromMaster()* – nakoľko klient s master komunikujú cez http spojenie, je potrebné, aby pred samotným prevedením obdržanej správy(input stream), bola odstránená hlavička http formátu a až potom konvertovaná na JSON súbor.
- *saveJSONToDB()* – metóda uloží všetky dáta z JSON súboru od master do databázovej tabuľky journal a client cashes.

7.5 Prijímanie journal požiadaviek a odpoveď na ne

Analýza

Ďalšou rozširujúcou funkciou, ktorú musí Ownet Android poskytovať je zdieľanie stiahnutých webových stránok medzi mobilnými zariadeniami (klientmi). Jednotliví klienti sa navzájom informujú o stránkach, ktoré majú u seba v pamäti, alebo ktoré u seba zmazali a už ich viac nemôžu nikomu poskytnúť. Journálove požiadavky prijíma každý klient a vytvára ne príslušnú odpoveď.

Návrh

Riešenie pozostáva z vytvorenia vlastného protokolu, ktorý sa posiela v podobe presne definovaného json súboru. V súbore sú obsiahnuté informácie o stránkach, ktoré majú klienti u seba na SD karte.

Snahou je redukovať množstvo a veľkosť posielaných journálových správ, preto sa posielajú iba informácie, ktoré daný klient ešte nemá.

Prijímanie journal požiadaviek prebieha na rovnakom porte ako počúvanie http požiadaviek z internetového prehliadača.

Journálová požiadavka obsahuje:

- ID klienta, ktorý poslal požiadavku
- skupiny klientov
- každá skupina klientov obsahuje zoznam klientov a číslo ich poslednej novinky (*client_rec_num*).

Na základe týchto informácií vytvorí príjemca odpoveď, ktorá obsahuje všetky novinky, ktoré sú novšie ako číslo novinky prijatej v požiadavke.

Implementácia

Zachytenú journalovú požiadavku spracováva trieda JournalUtil:

parseJournalRequest() - metóda dostane na vstupe journálovú požiadavku ako textový reťazec. Ten sa transformuje do JSON objektu, ktorý sa následne spracuje. Z objektu sú extrahované informácie o skupinách, klientoch a čísle ich poslednej journálovej novinky. Všetky informácie sa vkladajú do objektu triedy JournalRequestInformation.java.

getJournalsFromDB() - metóda dostáva na vstupe objekt triedy JournalRequestInformation, ktorý obsahuje informácie z požiadavky. Na základe ID klienta a jeho *client_rec_num* hodnoty sa vyhľadajú v tabuľke Journal všetky novšie záznamy (ich journálové číslo je väčšie ako client_rec_num). Každý záznam sa skladá z jedného riadku z tabuľky Journal a jedného riadku tabuľky ClientCaches.

Tabuľka Journal obsahuje cudzí kľúč UID do tabuľky ClientCaches.

Informácie sa ukladajú do objektu JournalResponseRow.

createJournalResponse() - metóda vytvára z listu JournalResponseRow objektov journálovú odpoveď vo forme JSONArray.

```
[
   {
      "client_id": "CLIENT_ID",
      "client_rec_num": "RECORD_ID",
      "table_name": "DB_TABLE",
      "sync_id": "SYNC_ID",
      "operation_type": "OPERATION_TYPE",
      "group_id": "GROUP_ID",
      "sync_with": "CLIENT_ID",
      "date_created": "DATE",
      "columns": {
         "cache_id ": "VALUE_1",
         "client_id ": "VALUE_2"
         "date_created": "VALUE_3"
      }
   },
   { ... }
]
```

getJournalResponseStream() - metóda transformuje vytvorenú journálovú odpoveď na InputStream. Na začiatok odpovede sa pridáva http hlavička: "*HTTP/1.1 200 OK\r\n\r\n*".

Triedy JournalRequestInformation, JournalResponseRow obsahujú iba metódy get a set, ktoré nastavujú ich premenné. Slúžia ako štruktúry na dočasné uloženie a prenos informácií medzi metódami.

7.6 Proxy request na iné zariadenie

Analýza

Primárnou funkciou journalu je posielanie informácií o aktuálnom stave nacachovaných stránok. Zo journalu sa následne vytvára zoznam zariadení a ich uložených stránok. Ak na našom telefóne chceme otvoriť stránku, ktorú už ma v pamäti uloženú niekto iný, môžeme sa pokúsiť stiahnuť content stránky z druhého zariadenia. Na druhom zariadení budeme musieť testovať či request prichádza z iného zariadenia, aby sme sa necyklili a mu tam nevkladali náš vlastný banner na stránku.

Návrh

Keď príde request na proxy, pozreme či ide z iného zariadenia. Ak áno, pozreme stránku v lokálnej cache a vrátime ak existuje alebo ju stiahneme z internetu. Následne ju ale nebudeme cachovat u nás. V druhom prípade, ak request ide z toho istého zariadenia a v našej cache sa nenachádza, pozrieme sa na iné zariadenia, či majú stránku v cache. Ak stránku majú, nastavíme nášmu http klientovi proxy parameter podľa daného zariadenia.

Implementácia

Na zistenie, či request na stránku prichádza z toho istého alebo iného zariadenia sme použili metódu *java.net.InetAddress.isLoopbackAddress()*. Ak vráti true, tak bol request poslaný z aktuálneho zariadenia.

RequestWorker.run()– V tejto metóde sme pridali kontrolu pôvodu requestu. Ak je z localhostu pozrieme sa do tabuľky client_caches a vyberieme iné zariadenie, ktoré ma stránku už nacachovanú a zoradíme podľa score.

HttpClient.getPage()– Do metódy sme pridali dva nové parametre pre proxy adresu a port proxy. Ak sú vyplnené nastavíme http clientovi proxy cez parameter *http.route.default-proxy*. V takom prípade request nepôjde priamo na stránku ale cez iné zariadenie. V prípade že proxy parameter je null, posielame request priamo na internet.

8 Grilované kuriatko

Cieľom šprintu bolo opraviť chyby v implementácii protokolu a začať implementovať webový portál.

8.1 Maintenance

Analýza

Súčasná implementácia našej aplikácie využíva paralelný prístup. Pri používaní technológie ORMLite dochádza ku konfliktom pri paralelnom prístupe pri generovaní hodnoty *client_rec_num*v tabuľke *Journal*. Táto hodnota sa má postupne pre každý záznam daného klienta inkrementovať.

Návrh

Táto hodnota sa bude získavať pomocou metódy, ktorá bude synchronizovaná, čiže vždy v nej bude len jedno vlákno. V rámci tejto metódy sa získa aktuálna najvyššia hodnota pre daného klienta a vykoná sa aj zápis. Tým pádom sa zabezpečí konzistentnosť údajov.

Implementácia

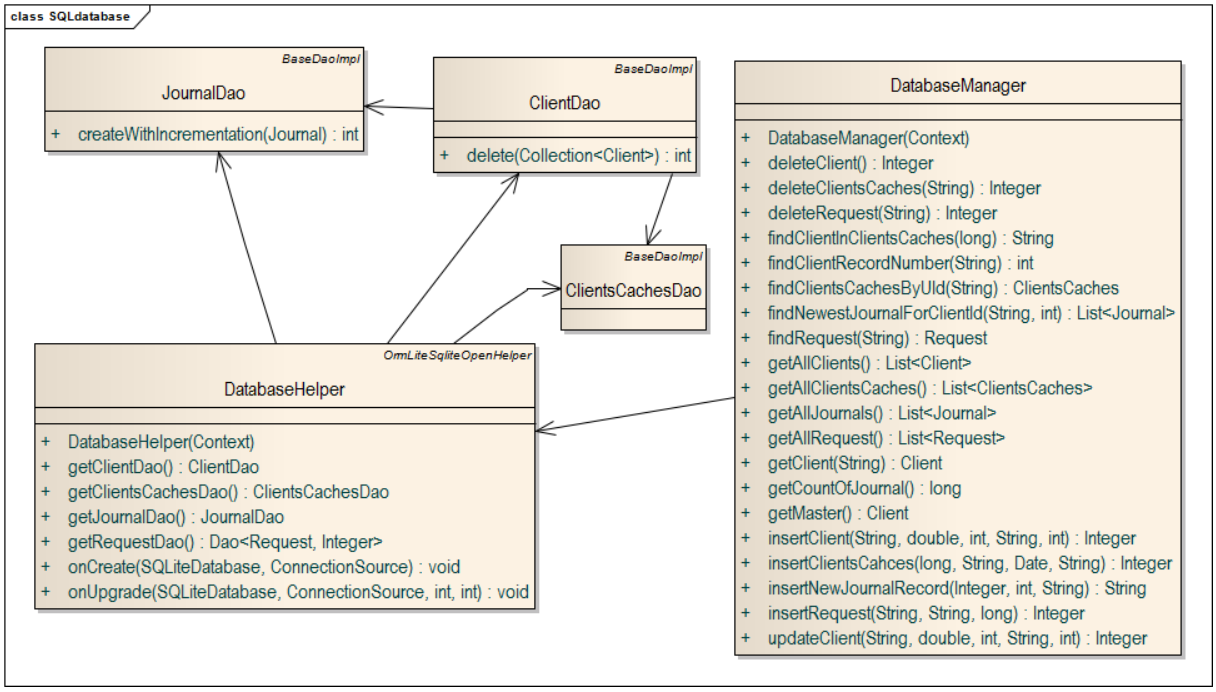
V triede *DatabaseManager* sme doplnili do metódy *insertNewJournalRecord()* rozhodovanie o tom, či bol zadaný vstupný parameter *client_rec_num*. V prípade, že nebol (príde *null*) tak sa zavolá novo vzniknutá metóda *addJournalWithIncrementation()*. V prípade, že daný atribút bude zadaný (ide o zápis z prijatého *JSON*)

DatabaseManager

- *addJournalWithIncrementation()* – metóda vyvolá synchrónnu metódu na, ktorá si získa hodnotu *client_rec_num* a zapíše daný riadok do tabuľky *Journal*. Vráti *UID* daného záznamu. V prípade chyby vráti *null*.

JournalDao

- *createWithIncrementation()* – synchrónna metóda získa správnu hodnotu*client_rec_num* a zapíše riadok do tabuľky. Vráti hodnotu *client_rec_num*. Vyhadzuje SQLException v prípade chyby.



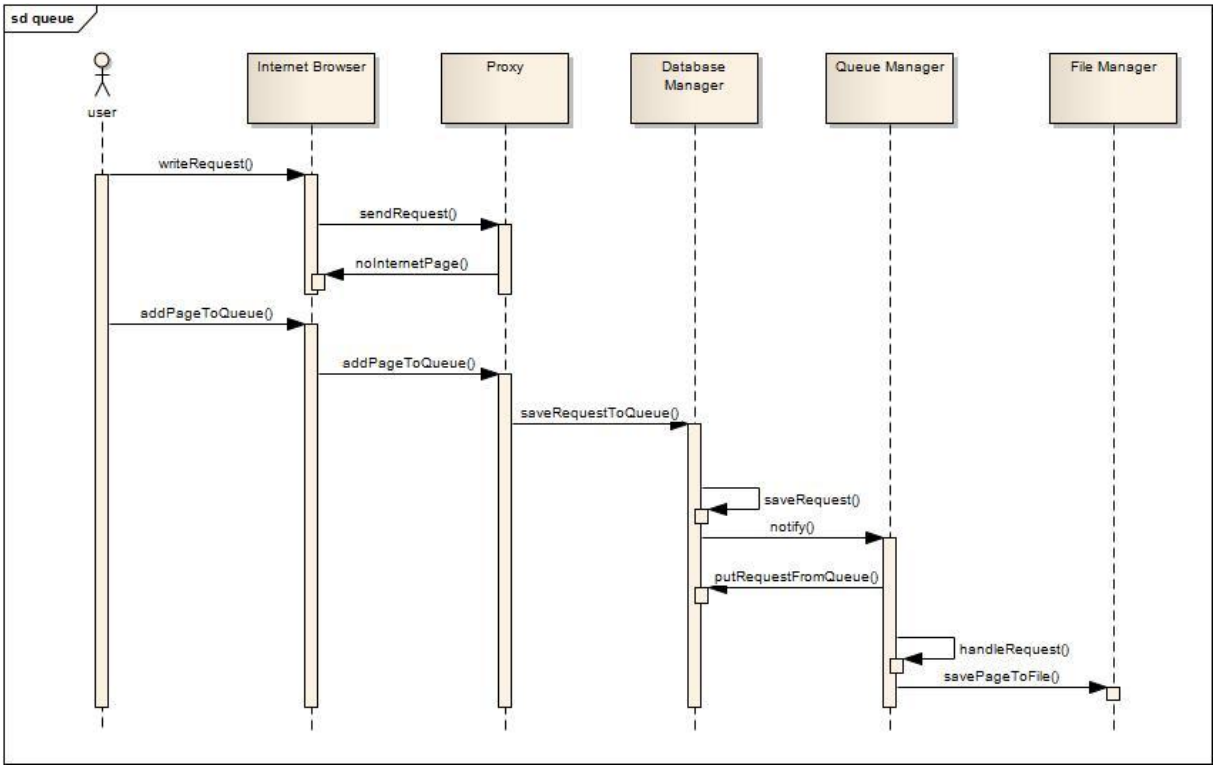
Obrázok 8.1 Baliík SQLDatabase

8.2 Queue

Na obrázku je sekvenčný diagram popisujúci proces pridávania požiadaviek do queue.

Kroky v prípade pridávania stránky do queue:

- Používateľ napíše link ktorý chce aby sa načítal z internetu.
- Prehliadač odošle požiadavku
- Proxy našej aplikácie túto požiadavky odchytí
- V prípade že nie je práve dostupné internetové pripojenie odošle prehliadaču odpoveď, kde si bude môcť používateľ požadovanú stránku , naplánovať na prednačítanie.
- V prípade že používateľ určí aby sa stránka prednáčítala, odošle prehliadač požiadavku z adresov požadovanej stránky.
- Proxy adresu odchytí .
- Databazový manažér uloží požiadavku do databázi a následne upovedomí o tom manažéra queue.
- Manažér queue vyberie požiadavku z databázi a čaká pokiaľ nebude mať používateľ internetové pripojenie.
- Keď sa používateľ pripojí na internet, queue manager automaticky stiahne celú stránku a prostredníctvom file manažéra ju uloží na disk.



Obr. 8.1 Sekvenčný diagram Queue

8.3 Webový portál

Webový postál slúži ako používateľské grafické rozhranie dostupné aj offline v podobe internetovej stránky.

8.3.1 Analýza

Úlohou webového portálu je poskytnúť používateľovi grafické rozhranie na prezeranie a manažovanie základných akcií nad dátami, ktoré buď on sám, alebo iný členovia v sieti pridali do databáz. Mal by poskytovať:

- Prehľad stiahnutých stránok
- Prehľad pridaných tagov, záložiek, nahratých súborov
- Možnosť vyhľadávania v obsahu
- Prehľad najnovších udalostí v sieti
- FAQ
- Informácie o projekte a stave jeho riešenia

Samozrejme by mal byť dostupný aj offline. V prípade slabého alebo žiadneho internetového pripojenia, by malo poskytnúť možnosť zaradiť stránku do radu úloh na splnenie (stiahnutie), ktoré sa spustia, keď bude mat zariadenie prístup k internetovému pripojeniu.

8.3.2 Návrh

Ako už bolo napísané, úlohou webového portálu je poskytnúť používateľovi grafické rozhranie na prezeranie a manažovanie základných akcií nad dátami v databáze. Na to aby tieto akcie mohol zabezpečiť je nutné vytvoriť stromovú štruktúru adresárov a súborov, ktoré samotný portál predstavujú.

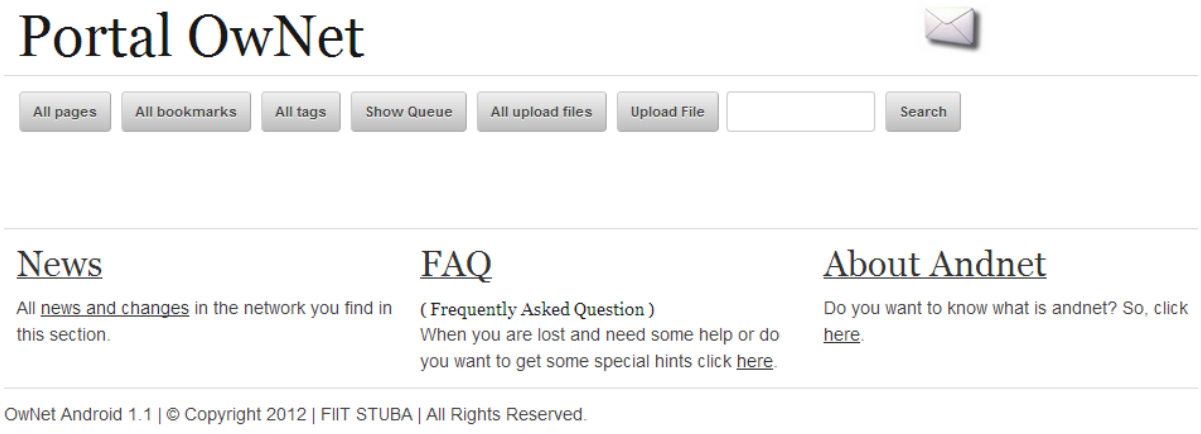
Stromová štruktúra adresárov pre webový portál (portal pages):

- **Js** – adresár bude obsahovať všetky skripty v jazyku JavaScript
- **Styesheets** – adresár bude obsahovať všetky CSS súbory
- **Graphics** – adresár bude obsahovať všetky obrázky a grafické prvky použité v portáli
- Samotné**HTMLstránky** – nazov.html

Aby sme zabezpečili funkcie portálu, ktoré boli zadefinované v analýze pre portál, odchytíme requesty so špecifickou URL adresou. Na základe tejto adresy sa vykoná požadovaná akcia (zobrazenie všetkých stránok, tagov, záložiek, atď.) a vráti výsledok späť na portál, kde sa zobrazia používateľovi.

8.3.3 Implementácia

Na vytvorenie webového portálu (úvodná stránka portálu na obrázku Obr. 1) bolo vytvorených niekoľko HTML, CSS, JS súborov, ktoré boli usporiadané do stromovej štruktúry podľa návrhu. Tieto súbory zabezpečujú správne zobrazovanie a funkcionalitu na strane používateľa.



Obr. 1Ukážka úvodnej stránky webového portálu

Zoznam všetkých súborov webového portálu sú uvedené v tabuľke **Chyba! Nenašiel sa iaden zdroj odkazov.**.

Adresár	Názov súboru	Popis
graphics	add.png, android.png, androidDevice.jpg, androidDeviceSett.jpg, androidOpera.png,	obrázky

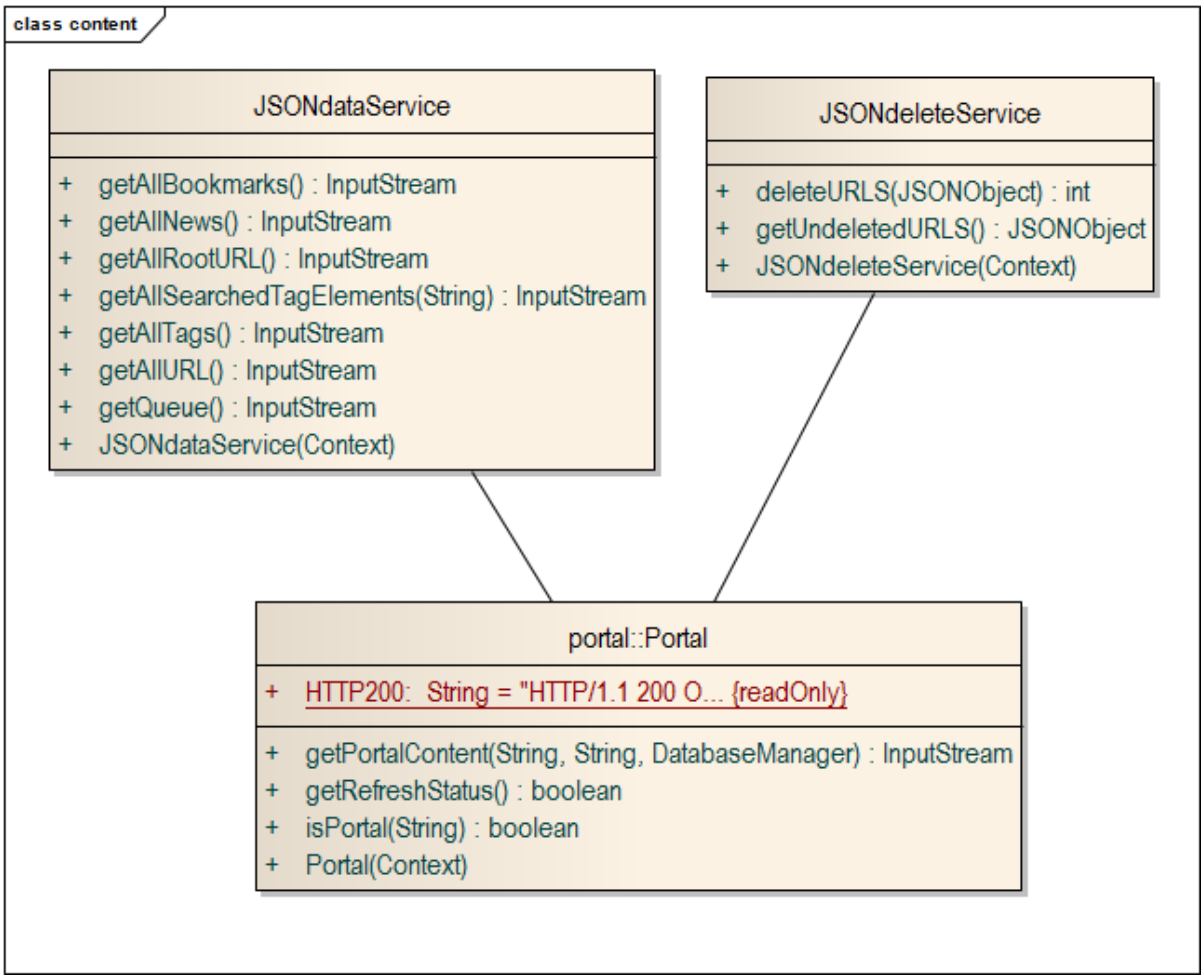
	bookmark.png, bookmark2.png, browserSettings.png, close.png, deviceAndnet.jpg, home.png, httpServer.png, mail.png, next.png, proxyPort.png, refresh.png, useHttp.png	
js	deleteURL.js	Funkcie poskytujúce mazanie
	Jquery-1.9.0.js, jQuery.js, json2.js	Všeobecné JS súbory
	printData.js	Funkcie poskytujúce funkcie na vypísanie obsahu portálu
	tabframeinject.sj	
stylesheets	base.css	Definovanie vizualizácie všetkých html prvkov
	layout.css	Definovanie typov zariadení
	skeleton.css	Definovanie zobrazovania a 16 typov rozlíšení
	tab.css	Definovanie vizualizácie prvkov pre zobrazenie v tabuľke
stránky	about.html	Stránka obsahuje informácie o projekte a jeho možnostiach a funkciách.
	addToQueue.html	Stránka slúži na pridávanie stránok do radu na stiahnutie, keď zariadenie nemá prístup na internet.
	faq.html	Stránka obsahuje často kladené otázky a odpovede na ne.
	index.html	Úvodná stránka, obsahuje menu na prechody k ostatným stránkam a tabuľku, v ktorej sa zobrazujú všetky stránky, záložky, atď.
	news.html	Stránka obsahuje prehľad posledných aktivít v sieti.

	search.html	Stránka obsahuje obsah databázy podľa hľadaného vzoru.

Tabuľka 8.1Stromová štruktúra adresárov a súborov s podrobným opisom ich obsahu

Aby portál správne fungoval a zariadenie potrebuje okrem grafického rozhrania aj funkcionalitu pod grafickými prvkami. Pre tieto potreby bol vytvorený balík ownet.aplication.portal. Trieda Portal slúži na prideľovanie a vyhodnocovanie requestov, ktoré sa viažu na webový portál. Táto trieda je úzko naviazaná na triedy JSONdataService, JSONdeleteService a JSONHelper, ktoré poskytujú funkcie na tvorbu požadovaných JSON súborov, ktoré obsahujú dáta z databázy podľa vstupných parametrov.

Prepojenie tried môžeme vidieť na nasledujúcom obrázku Obr. 8.2 balík portal.contentObr. 8.2.



Obr. 8.2 balík portal.content

Trieda *Portal* obsahuje nasledujúce funkcie:

- *getPortalContent()* – funkcia vráti obsah z databázy v podobe *InputStream* pre potreby zobrazenia v portáli

- *getRefreshStatus()* – funkcia vráti informáciu o poslednom načítaní stránky a teda či je aktuálna alebo je potrebné ju obnoviť pre načítanie nového obsahu.
- *isPortal()* – funkcia overí či stránka, ktorá je ako vstupný parameter je zo stránok patriacich webovému portálu, alebo nie.
- *Portal()* – konštruktir triedy *Portal*.

Trieda *JSONDataService* obsahuje nasledujúce funkcie:

- *getAllBookmarks()* – funkcia vráti všetky záložky uložené v databáze.
- *getAllNews()* – funkcia vráti všetky novinky v sieti.
- *getAllRootURL()* – funkcia vráti všetky hlavné (väčšinou je to prvá adresa, ktorá je poslaná ako request) URL adresy.
- *getAllSearchedTagElements()* – vráti všetky záznamy z databázy pokiaľ tag alebo záložka obsahuje hľadaný výraz.
- *getAllTags()* – vráti všetky tagy z databázy.
- *getAllURL()* – vráti všetka URL adresy z databázy.
- *getQueue()* – vráti rad URL adries, ktoré čakajú na stiahnutie ako náhle sa zariadeniu podarí pripojiť do siete s internetovým pripojením.
- *JSONDataService()* – konštruktor triedy *JSONDataService*.

Trieda *JSONDeleteService* obsahuje nasledujúce funkcie:

- *deleteURLS()* – funkcia odstráni všetky záznamy z databázy, ktoré obsahujú URL adresu prijatú v parametri funkcie. Vráti 1 ak sa jej podarí odstrániť všetky záznamy, inak vráti -1.
- *getUndeleterURLS()* – funkcia vráti JSON súbor obsahujúci všetky nevymazané záznamy.
- *JSONDeleteService()* – konštruktor triedy *JSONDeleteService*.

8.4 Synchronizácia tagov, bookmarkou a uploadfiles cez journal

8.4.1 Analýza

Všetky tagy a bookmarky by mali byť zdielané so všetkými ostatnými zariadeniami. Ak pridáme bookmark alebo otagujeme zaujímavú stránku, ostatný používatelia sú o tom hneď informovaný a môžu si danú stránku pozrieť ak ich tag zaujal.

8.4.2 Návrh

Jednotlivé udalosti pri vytváraní bookmarkov, tagov alebo upload súborov budeme synchronizovať pomocou existujúcej funkcionality journalu. Journal doteraz slúžil iba na vymieňanie informácií o stave cache stránok.

8.4.3 Implementácia

Každá trieda reprezentujúcu tabuľku, ktorá sa bude synchronizovať dostala spoločné rozhranie *ownet.aplication.model.IJournaled*. Jedná sa o triedy Bookmark, UploadFiles a ClientCaches.

Pridaná nová tabuľka pre bookmarky:

Názov	Typ	Popis
COLUMN_ID	Int	ID záznamu
COLUMN_CLIENT_ID	String	ID device
COLUMN_DATE	Date	Dátum vytvorenia
COLUMN_UID	String	UID pre journal
TITLE	String	Hodnota
COLUMN_IS_TAG	Int	1 pre tag, 0 bookmark
COLUMN_URL	String	url stránky

Obr. 8.3 tabuľka bookmark

Úpravy journalu pre prácu s inými tabuľkami:

Journal.saveJSONToDB() – upravená funkcia pre spracovanieostatných tabuliek pre synchronizáciu

JournalUtil.createJournalResponse– pridané vyskladanie response JSON pre journal o nové tabuľky

JournalUtil.getJournalsFromDB – pridané vyberanie nových typov pre synchronizáciu z databáze na základe hodnoty table_name v journal zázname.

JournalResponseRow. journalObject – úprava typu z client_cache na rozhranie IJournaled.

DatabaseManager.insertBookmark() – nová metóda pre vloženie bookmarku zo journalu iného zariadenia

DatabaseManager.insertUploadFile() – nová metóda pre vloženie uploadFile záznamu do databázy zo journalu.

9 Teľací steak

Cieľom šprintu bolo opraviť pretrvávajúce chyby v implementácii protokolu

9.1 Maintenance

Analýza

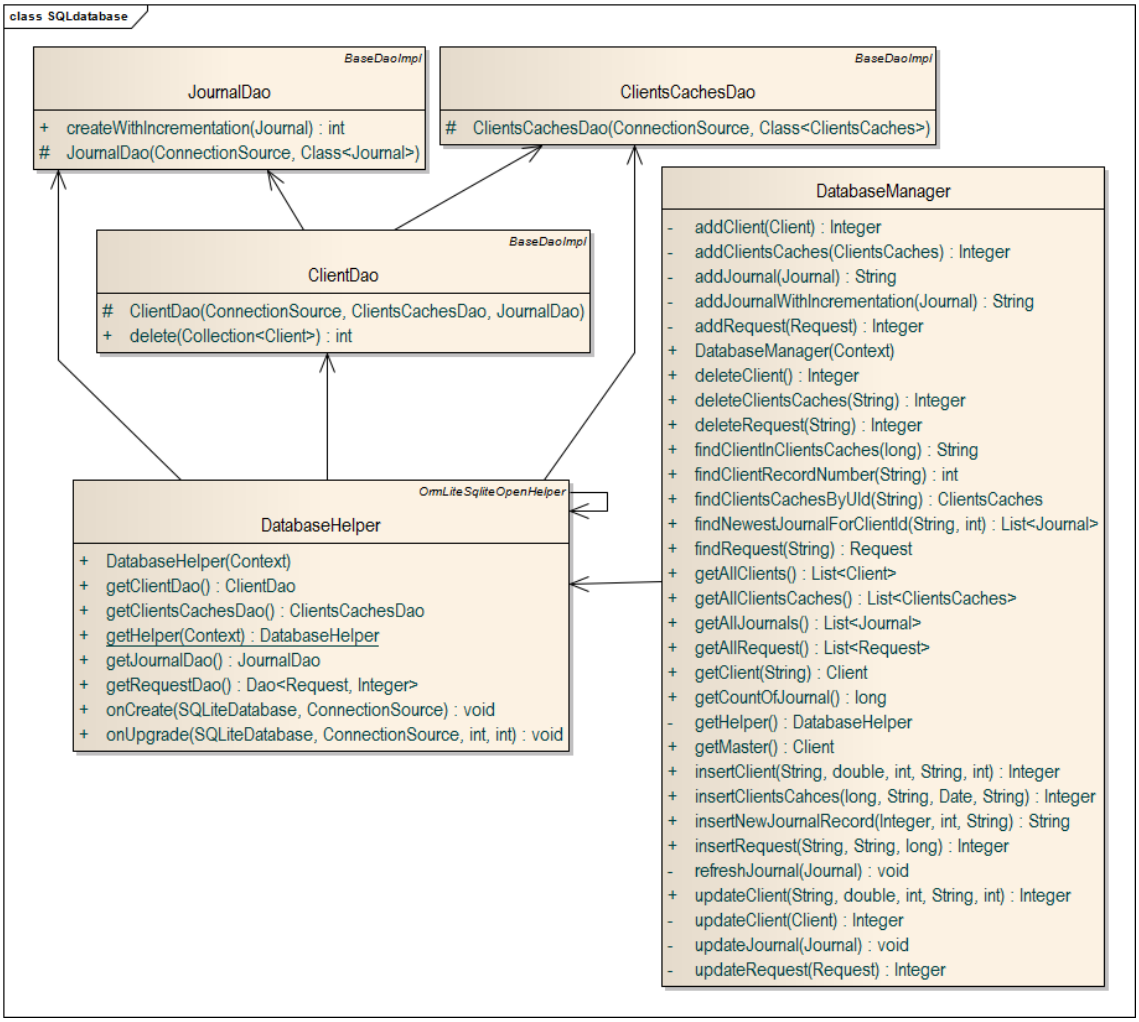
V rámci paralelizmu sa každé vlákno vytvára vlastnú inštanciu triedy *DatabaseManager*, čím sa vytvorí aj vlastná inštancia triedy *DatabaseHelper* a tým sa aj vytvorí vlastná inštancia tried: *ClientDao, ClientsCachesDao* a *JournalDao*. Keď je viacej tried, ktoré sa snažia pristupovať k rovnakej tabuľke tak dochádzalo ku konfliktom a nie vždy sa podarilo vykonať danú operáciu.

Návrh

Vytvorí sa jedna statická inštancia triedy *DatabaseHelper* a tú budú používať všetky inštancie triedy *DatabaseManager*. Pri takomto riešení sa vytvorí vždy len jedna inštancia z tried: *ClientDao, ClientsCachesDao* a *JournalDao*. Teraz sa bude k danej tabuľke pristupovať len jeden objekt, ktorý je synchrónny a nebudú vznikať konflikty.

Implementácia

V triede *DatabaseHelper* sa vytvorila jej statická inštancia, ku ktorej *DatabaseManager* pristupuje cez metódu *getHelper()*.



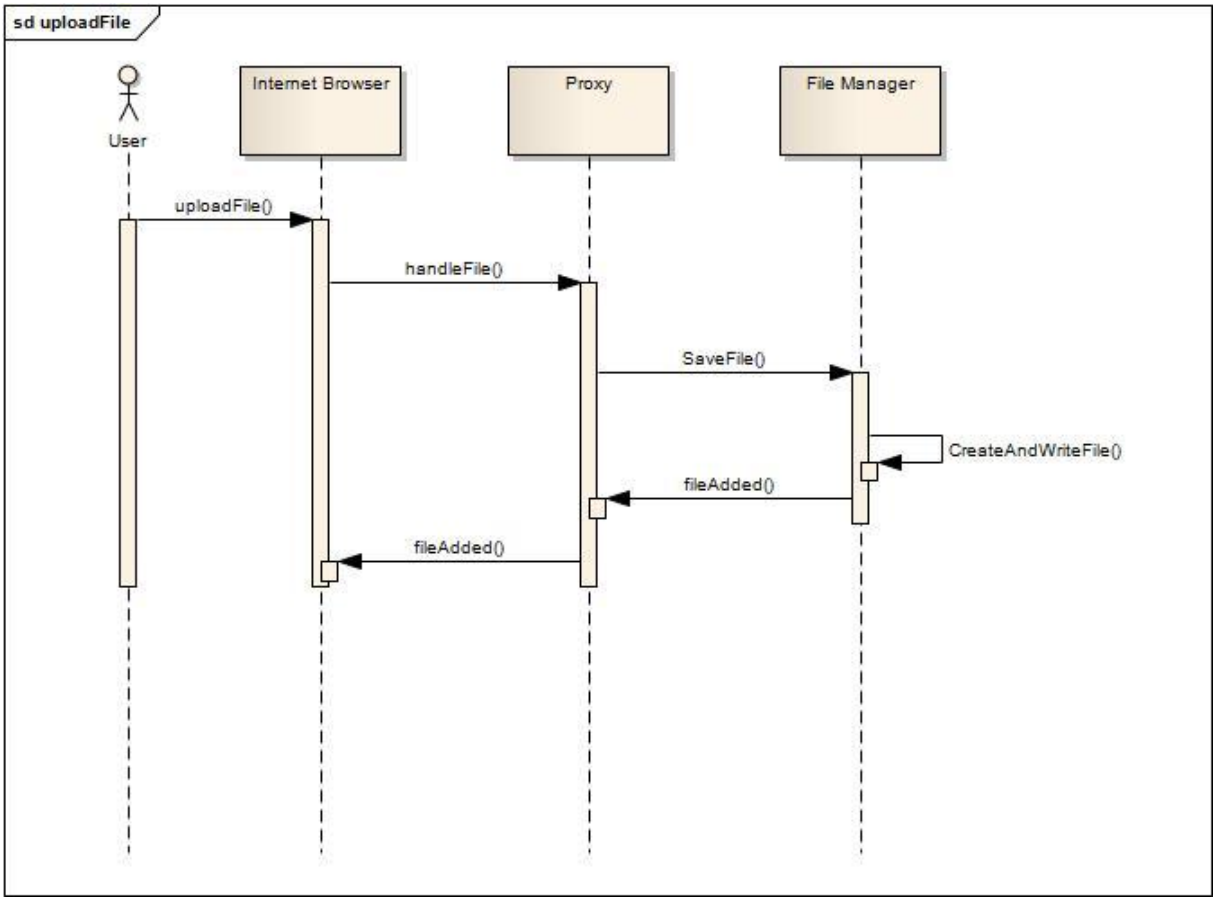
Obr. 9.1 Balík SQLDatabase

9.2 Upload súborov

Na Obr. 9.2 je sekvenčný diagram popisujúci proces uploadovania súborov na ownet portál.

Proces sa skladá z nasledovných krokov:

- Používateľ cez ownet portál odošle súbor na upload.
- Prehliadač odošle požiadavku pre upload.
- Proxy odchytí túto požiadavku z nej vyberie obsah súboru.
- Obsah súboru sa následne uloží na pamäťovú kartu.
- Následne proxy vráti odpoveď prehliadaču v podobe stránky, ktorá informuje používateľa, že sa upload úspešne podaril.



Obr. 9.2 Sekvenčný diagram pre upload súborov