Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Stanič

Varnostna razširitev vtičnika za ogrodje Ionic

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Mojca Ciglarič
Somentor: dr. Dušan Gabrijelčič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pripravite pregled ogrodij za razvoj hibridnih mobilnih aplikacij. Preverite možnosti overjanja uporabnika in nato na izbranem ogrodju poiščite najboljši način za izvedbo overjanja uporabnika z digitalnimi potrdili. Overjanje tudi implementirajte in testirajte najprej z namensko testno aplikacijo, nato pa tudi z realno mobilno aplikacijo.

Zahvaljujem se somentorju dr. Dušanu Gabrijelčič skupaj z odsekom E5 - Laboratorij za odprte sisteme in mreže na Inštitutu Jožef Stefan ter mentorici izr. prof. dr. Mojci Ciglarič za strokovno pomoč pri izdelavi diplomske naloge. Prijateljem in družini se zahvaljujem za vso podporo ob študiju.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	Aplikacijski programski vmesnik
CA	Certificate Authority	Certifikatska agencija
CLI	Command Line Interface	Vmesnik ukazne vrstice
HTTP	Hypertext Transfer Protocol	Protokol za prenos hiperbesedila
HTTPS	HTTP Secure	Varni HTTP
IDE	Integrated Development Enviornment	Integrirano razvojno okolje
IoT	Internet Of Things	Internet stvari
JSON	JavaScript Object Notation	JavaScript zapis objektov
MAC	Message Authentication Code	Koda za overjanje sporočila
MITM	Man In The Middle attack	Napad z vmesnim členom
NPM	Node Packet Manager	Upravljalnik NodeJS paketov
REST	Representational State Transfer	Predstavitveni prenos stanja
SDK	Software Development Kit	Orodje za razvoj programske opreme
SSL	Secure Sockets Layer	Varna povezovalna plast
TLS	Transport Layer Security	Varnost na transportni plasti
URL	Uniform Resource Locator	Enolični naslov vira
XML	Extensible Markup Language	Razširljivi označevalni jezik

Povzetek

Naslov: Varnostna razširitev vtičnika za ogrodje Ionic

Avtor: Klemen Stanič

Uporaba ogrodij za razvoj hibridnih mobilnih aplikacij zaradi preprostosti ter hitrosti razvoja hitro narašča. Problem se pojavi, ko ogrodje, ki smo ga izbrali za razvoj aplikacije, ne omogoča vseh potrebnih funkcionalnosti. Ogrodje Ionic tako ne omogoča vzpostavitve HTTPS povezave, kjer bi strežnik overil uporabnika z uporabo uporabnikovega digitalnega potrdila. Na tak način bi lahko strežnik uporabniku dodelil določene pravice dostopa. V okviru diplomskega dela to funkcionalnost dodamo že obstoječemu vtičniku za ogrodje Ionic. Vtičnik nato ovrednotimo z uporabo v testni mobilni aplikaciji ter mobilni aplikaciji, namenjeni prikazu porabe električne energije v gospodinjstvu.

Ključne besede: Ionic, hibriden razvoj aplikacij, pametni telefon, mobilna aplikacija, vtičnik, TLS, uporabnikovo digitalno potrdilo, overjanje.

Abstract

Title: Ionic Plugin Security Extension

Author: Klemen Stanič

Hybrid mobile application frameworks are becoming more and more popular, due to how simple and fast it is to develop a mobile application that will work on multiple platforms. Sometimes, these frameworks don't provide all the functionality that native application development does. Ionic framework doesn't provide a way to establish an HTTPS connection with client authentication. In this thesis, we add an option to use client authentication to an Ionic plugin. We evaluate the plugin by using it in a test application. We also use the plugin in a real-life smart mobile application, that is used for presenting the electric consumption of a specific household, and evaluate it that way also.

Keywords: Ionic, hybrid app development, smartphone, mobile app, plugin, TLS, client certificate, authentication.

Poglavje 1

Uvod

Življenje brez pametnih telefonov si dandanes težko predstavljamo. V začetku so bili mobilni telefoni namenjeni le klicanju ter pisanju sporočil, vendar so s časom postali pravi mali prenosni računalniki. Uporabljamo jih lahko kot fotoaparat oziroma kamera, na njih lahko brskamo po spletu, igramo igre, nameščamo in uporabljamo aplikacije, gledamo večpredstavnostne vsebine ter podobno. Z vedno večjim številom uporabnikov mobilnih telefonov se veča tudi potreba po novih aplikacijah. Trg aplikacij je ogromen, veliko programerjev živi le od izdelovanja in vzdrževanja aplikacij. Tako je za Android mobilno platformo na Google Play trgovini na voljo več kot 3.3 milijone aplikacij, za platformo iOS pa več kot 2.2 milijona aplikacij [38]. Poleg Android in iOS platform so na trgu prisotne še ostale platforme, kot so Windows Phone, Windows 10, Blackberry, vendar imajo v primerjavi z vodilnima platformama veliko manjše število aktivnih uporabnikov. Tako razvijalci mobilnih aplikacij večinoma razvijajo za Android in iOS platformi.

Poleg platformno-specifičnih orodij za razvoj aplikacij obstajajo tudi ogrodja za izdelavo hibridnih aplikacij. Najpopularnejša so React Native, Ionic, Framework 7, PhoneGap, Onsen UI ter ostala [1]. Ta so velikokrat uporabljena tudi pri razvoju aplikacij, namenjenih le eni platformi, saj omogočajo hitrejši razvoj kot platformno-specifična orodja (Android Studio, Xcode, itd). Predvsem aplikacije, ki imajo velik poudarek na uporabniški

izkušnji, lahko hitreje razvijemo z uporabo hibridnih ogrodij. Določeni gradniki uporabniškega vmesnika so že vnaprej definirani, in jih tako samo vstavimo v našo aplikacijo.

S povečevanjem števila pametnih naprav se povečuje tudi potreba po varni in zaupni povezavi med različnimi napravami ter storitvami. Informacije, ki se prenašajo med napravami, so namreč velikokrat zaupne narave ter lahko vsebujejo uporabnikove osebne podatke. Tako moramo kot načrtovalci programske opreme zagotoviti določeno stopnjo varnosti in zaupnosti. Da je povezava med dvema členoma v IoT sistemu šifrirana in zaupna, uporabljamo protokole, ki zagotavljajo varen prenos podatkov. Tako naprimer namesto navadne povezave HTTP uporabimo povezavo HTTPS, ki temelji na protokolu SSL/TLS.

Ob razvoju aplikacije, ki je namenjena pregledu električne porabe gospodinjstva smo naleteli na težavo. Namreč ogrodje Ionic, ki smo ga uporabljali pri razvoju, ne omogoča vzpostavitve povezave HTTPS z uporabo odjemalčevega digitalnega potrdila (ang. *client certificate*). Ta funkcionalnost je bila za delovanje obvezna. Namreč strežnik s podatki o porabi električne energije od uporabnika zahteva, da se predstavi s svojim digitalnim potrdilom. Tako strežnik overi odjemalca in mu priredi določene pravice dostopa ter mu omogoči vzpostavitev varne povezave.

Zato smo sklenili, da v okviru diplomskega dela manjkajočo funkcionalnost ogrodju Ionic dodamo sami. Funkcionalnost smo dodali obstoječemu vtičniku (ang. *plugin*), ki je namenjen ogrodju Apache Cordova [9], ter posledično tudi ogrodju Ionic [13]. Da smo lahko delovanje vtičnika testirali, smo morali postaviti še testni spletni strežnik, ki omogoča več načinov povezovanja. Vtičnik smo ovrednotili z uporabo na testni mobilni aplikaciji ter na mobilni aplikaciji, namenjeni prikazu električne porabe gospodinjstva.

Poglavje 2

Pregled

Za boljše razumevanje, kaj smo v sklopu diplomske naloge naredili, bomo najprej opisali vsa orodja, protokole, ogrodja ter storitve, ki smo jih pri razvoju uporabljali. Najprej bomo opisali mobilni platformi, nato pa še ogrodja za razvoj hibridnih mobilnih aplikacij. Opisali bomo tudi povezavo SSL/TLS, kako se le-ta vzpostavi ter predstavili nekaj terminologije iz tega področja. Na koncu poglavja bomo opisali še Ionic vtičnik „cordova-plugin-advanced-http“, ki smo ga v sklopu te diplomske naloge razširili.

2.1 Mobilne platforme

Razvijalci mobilnih aplikacij večinoma razvijajo aplikacije za platformi Android in iOS. Aplikacije morajo biti zanimive, uporabne, hitre, privlačne ter varne. Vsaka izmed platform mora omogočati hiter, preprost ter poceni razvoj aplikacij. Če želimo aplikacijo narediti dostopno tako Android kot iOS platformi, moramo bodisi razviti dve aplikaciji, torej za vsako platformo napišemo celotno aplikacijo, ali pa uporabimo eno izmed ogrodij za izdelovanje hibridnih aplikacij.

2.1.1 Android

Android [Slika 2.1] je mobilni operacijski sistem, ki ga je razvil Android Inc. Kasneje, leta 2005 ga je odkupil Google. Prva komercialno dostopna Android naprava je na trg prišla leta 2005. Osnovan je na Linux jedru [33] ter ostali odprtokodni programski opremi. Namenjen je predvsem mobilnim napravam z zaslonom na dotik, kot so pametni telefoni in tablice. Prav tako je nameščen na velikem številu igralnih konzol, digitalnih kamerah, televizijah ter pametnih urah. Je najbolj popularen mobilni operacijski sistem, saj ima trenutno 70% uporabnikov mobilnih naprav Android operacijski sistem, kar predstavlja več kot 2 milijardi mesečno aktivnih uporabnikov [2, 38].

Za razvoj Android aplikacij programerji najpogosteje uporabljajo prostodostopno orodje Android Studio [6] oziroma Eclipse [10]. Uporabniške vmesnike definiramo z uporabo XML shem [3], samo logiko aplikacije pa pišemo v programskem jeziku Java [14]. Od oktobra 2017 pa je poleg Jave uradno podprt tudi programski jezik Kotlin [32].

Slika 2.1: Ikona operacijskega sistema Android

2.1.2 iOS

IOS [Slika 2.2] je mobilni operacijski system, ki ga je zasnoval in razvil Apple Inc. [8]. Prva naprava s tem operacijskim sistemom je bil iPhone, ki je s prodajo začel leta 2007. Je namenjen le mobilnim napravam, ki jih proizvaja Apple in je trenutno drugi najbolj popularen mobilni operacijski sistem, takoj za Android platformo. Aplikacije za iOS so na voljo na App Store, kjer je trenutno več kot 2.2 milijona aplikacij [38, 31].

Za razvoj iOS aplikacij razvijalci večinoma uporabljajo IDE XCode [23], ki ga je razvil Apple in je uradno okolje. Aplikacije so napisane v programskem jeziku Swift [46] ali pa Objective-C [39]. Za definicijo uporabniških vmesnikov se prav tako kot pri Androidu uporabljajo sheme XML [3].

Slika 2.2: Ikona operacijskega sistema iOS

2.2 Ogrodja za razvoj hibridnih aplikacij

Če razvijamo aplikacijo, namenjeno večim platformam, oziroma je pri naši aplikaciji poudarek bolj na uporabniški izkušnji, lahko uporabimo eno izmed ogrodij za razvoj hibridnih aplikacij. Te omogočajo hkratni razvoj aplikacij, ki bodo delovale na več platformah. Tako nam uporabniških vmesnikov ter delovanja aplikacije ni potrebno implementirati za vsako platformo posebej. Aplikacijo napišemo bodisi s spletnimi tehnologijami, bodisi z enim za to namenjenim programskim jezikom. Taka aplikacija se nato v okviru ogrodja

prevede v platformno-specifične aplikacije.

Pri izbiri ustreznega ogrodja za razvoj hibridnih aplikacij imamo veliko izbire. Pri sprejemanju odločitve moramo upoštevati kakšne vrste aplikacijo želimo razviti ter kaj nam je pri razvoju najbolj pomembno.

Ob pregledu, katere načine vzpostavitve povezave HTTPS različna ogrodja omogočajo, smo ugotovili, da trenutno nobeno ogrodje ne omogoča uporabe uporabnikovega digitalnega potrdila pri vzpostavitvi seje TLS. Pri večini je dodajanje te funkcionalnosti v planu oziroma možno le na nivoju domorodnega (ang. *native*) razvoja, vendar tega nihče še ni implementiral. Med spodnje naštetimi ogrodji je Ionic najbolj popularna izbira, zato smo se odločili, da to funkcionalnost dodamo temu ogrodju.

Trenutno so najbolj priljubljena in v uporabi naslednja ogrodja [1]:

2.2.1 React Native [20]

Ogrodje omogoča razvoj hibridnih aplikacij z uporabo JavaScript programskega jezika ter React.js [21] knjižnice. Razvil ga je Facebook, je odprtokoden ter omogoča hiter razvoj aplikacij, ki so istega izgleda kot domorodne (ang. *native*) aplikacije. Ponuja veliko vtičnikov, aplikacije narejene s tem ogrodjem so hitre in zelo odzivne. Ponuja veliko število že narejenih komponent, kar pospeši razvoj aplikacij.

2.2.2 PhoneGap [17]

Je odprtokodno ogrodje, ki omogoča preprost razvoj hibridnih aplikacij s pomočjo spletnih tehnologij (CSS, HTML, JavaScript). Ponuja storitev imenovano „Build“, ki razvijalcu omogoča testiranje aplikacije na večih platformah, brez namestitve SDK-jev za vse platforme. Žal aplikacije razvite s tem ogrodjem niso najbolj hitre ter odzivne, vendar je ogrodje še vedno primerno za razvoj preprostih mobilnih aplikacij.

2.2.3 Framework 7 [12]

Prednost ogrodja Framework 7 je, da za razliko od drugih hibridnih ogrodij ne uporablja nobenih zunanjih storitev, kot sta Angular.js [7] in React.js [21]. Aplikacije razvijamo s pomočjo spletnih tehnologij (CSS, HTML, JavaScript), kar predstavlja relativno lažji razvoj aplikacije. Je odprtokoden, aplikacije razvite s tem ogrodjem pa so odzivne ter hitre. Ima veliko število že narejenih komponent uporabniškega vmesnika, ki jih lahko kar vstavimo v našo aplikacijo in tako hitro razvijemo moderno ter elegantno aplikacijo.

2.2.4 Xamarin [22]

Ogrodje je izšlo leta 2011 ter je podprto s strani ene največjih tehnoloških podjetij na svetu, t.j. Microsoft. Mobilne aplikacije, ki so razvite z ogrodjem Xamarin so po izgledu, zmogljivosti ter učinkovitosti podobne domorodnim (ang. *native*) aplikacijam. Delovanje aplikacije definiramo z Microsoftovim programskim jezikom .NET [24]. Razvijamo lahko aplikacije namenjene Android, iOS ter Windows platformi.

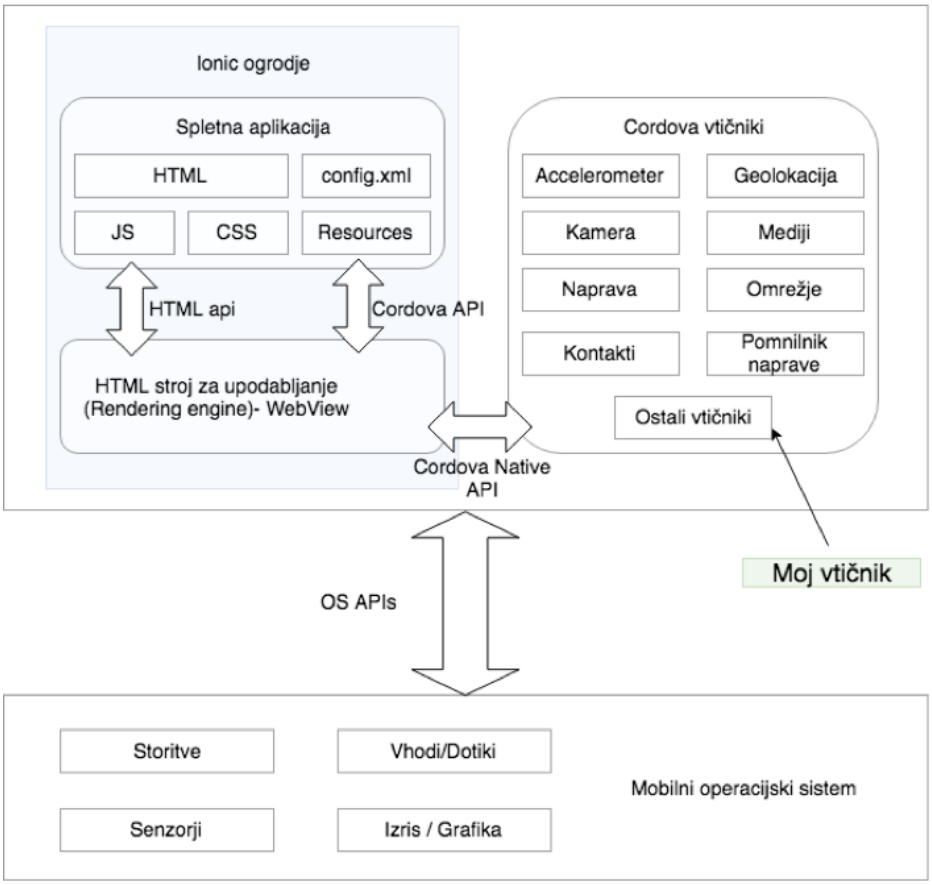
2.2.5 Ionic [13]

Slika 2.3: Ikona ogrodja Ionic

Ionic [Slika 2.3] je trenutno najbolj popularno odprtokodno ogrodje za izdelovanje hibridnih aplikacij, namenjenih mobilnim platformam. Prva verzija ogrodja je izšla leta 2013. Uporablja odprtokodno JavaScript ogrodje AngularJS [7], namenjeno izdelavi vidnega dela aplikacije, ter Apache Cordovo [9]. Slednja omogoča uporabo funkcionalnosti samega operacijskega sistema naprave z uporabo vtičnikov.

Ionic omogoča izdelovanje hibridnih aplikacij s pomočjo spletnih tehnologij, kot so CSS, HTML5, Sass ter JavaScript/TypeScript. Za posamezne zaslone aplikacije uporablja „strani“ (ang. *pages*), po funkcionalnosti podobne aktivnostim (ang. *activity*) pri razvoju za platformo Android. Razvoj aplikacij z ogrodjem Ionic je podoben izdelavi spletne strani, le da je ta na napravi pognana kot aplikacija, ne pa z brskalnikom. Z uporabo spletnih tehnologij določimo izgled ter delovanje aplikacije, vsa koda pa se nato s pomočjo Apache Cordove prevede v več aplikacij, namenjenih večim platformam. Aplikacije, narejene s pomočjo Ionica tako niso povsem domorodne (ang. *native*) – upodabljanje opravlja spletni pogled (ang. *web view*) ne pa platformno ogrodje za uporabniške vmesnike. Hkrati pa niso povsem spletne, saj so zapakirane za distribucijo ter imajo dostop do platformnih APIjev. Delovanje ter sodelovanje Ionica ter Cordove je prikazano na sliki [Slika 2.4].

Ionic uporablja Apache Cordova [9] vtičnike, da lahko dostopa do funkcionalnosti gostiteljskega operacijskega sistema, kot je recimo kamera, GPS, svetilka, internetni dostop itd. Podpira Android, iOS ter Windows operacij-



Slika 2.4: Shema delovanja Ionic in Cordova ogrodij

ske sisteme. Poleg mobilnih aplikacij pa lahko aplikacijo naredimo tudi za sodobne spletne brskalnike.

Trenutna verzija Ionic ogrodja je „Ionic Framework 3“. Omogoča izdelovanje aplikacij za Android naprave z operacijskim sistemom različice 4.1 ter višjimi, iOS 7 oziroma višje in naprave z Windows 10 operacijskim sistemom ter tudi za BlackBerry 10 telefone.

2.3 TLS

TLS [48] je kriptografski protokol, ki omogoča komunikacijsko varnost v omrežju. Pred TLS-jem je bil v uporabi SSL, ki pa je sedaj zastarel in se ga ne uporablja več. Protokol TLS je uporabljen za komunikacijo v odjemalec-strežnik aplikacijah kot je brskanje po spletu, elektronska pošta, telefonija prek IP ter v ostalih aplikacijah, kjer morajo podatki, ki se prenašajo po omrežju ostati zaupni ter nespremenjeni. Preprečuje prisluškovanje ter spreminjanje podatkov. Deluje nad protokolom TCP in ustvari varen kanal za prenos podatkov [42].

TLS protokol stremi k zagotavljanju zasebnosti in celovitost podatkov, ki se prenašajo med dvema členoma v omrežju. Če je komunikacija zaščitena z uporabo TLS, velja [48]:

- Povezava je varna/zasebna, saj je za šifriranje podatkov uporabljena simetrična kriptografija [47]. Ključ, ki je potreben za vzpostavitev take povezave je za vsako povezavo unikaten in temeljij na skupni skrivnosti, za katero se dogovorita obe strani komunikacije ob začetku seje. Odjemalec ter strežnik se odločita kakšen šifrirni algoritem ter ključ bo uporabljen, preden pride do prenosa podatkov,

- ker TLS uporablja šifriranje z javnim ključem (ang. *public key cryptography* [44]), lahko overimo obe strani komunikacije, tako odjemalca kot strežnik. Overjanje ni obvezno, vendar je v splošnem potrebno vsaj s strani odjemalca. Odjemalec se mora prepričati, da je strežnik res tisti, za katerega se izdaja. Tako preprečimo napade MITM [34], kjer odjemalec podatke pošilja napadalcu, namesto strežniku,

- povezava je zanesljiva, saj se ob vsakem poslanem sporočilu preveri tudi celovitost sporočila (ang. *message integrity check*) z uporabo MAC [36]. To zagotavlja, da podatki med prenosom niso bili spremenjeni ter da ni prišlo do izgube podatkov.

Povezava TLS se vzpostavi z naslednjim zaporedjem sporočil, prikazanih na diagramu [Slika 2.5]. Temu postopku pravimo rokovanje (ang. *handshake*). Diagram prikazuje zaporedje akcij pri začetnem delu vzpostavitve povezave TLS, t.j. v postopku rokovanja, kot je opisan v [43]. Znak * je zapisan ob sporočilih/razširitvah, ki ob TLS rokovanju niso nujno potrebni, ampak jih uporabljamo pri naprednejših načinih vzpostavitve TLS povezave.

V splošnem se faza rokovanja opravi v treh fazah:

1. Izmenjava ključev (ang. *key exchange*): Odjemalec in strežnik si izmenjata ključ in izbereta kriptografske parametre. Po tej fazi je ves promet šifriran,

2. strežniški parametri (ang. *server parameters*): Obe strani povezave si izmenjata še ostale parametre rokovanja (ali je odjemalec overjen, podpora protokolov aplikacijske plasti, itd.),

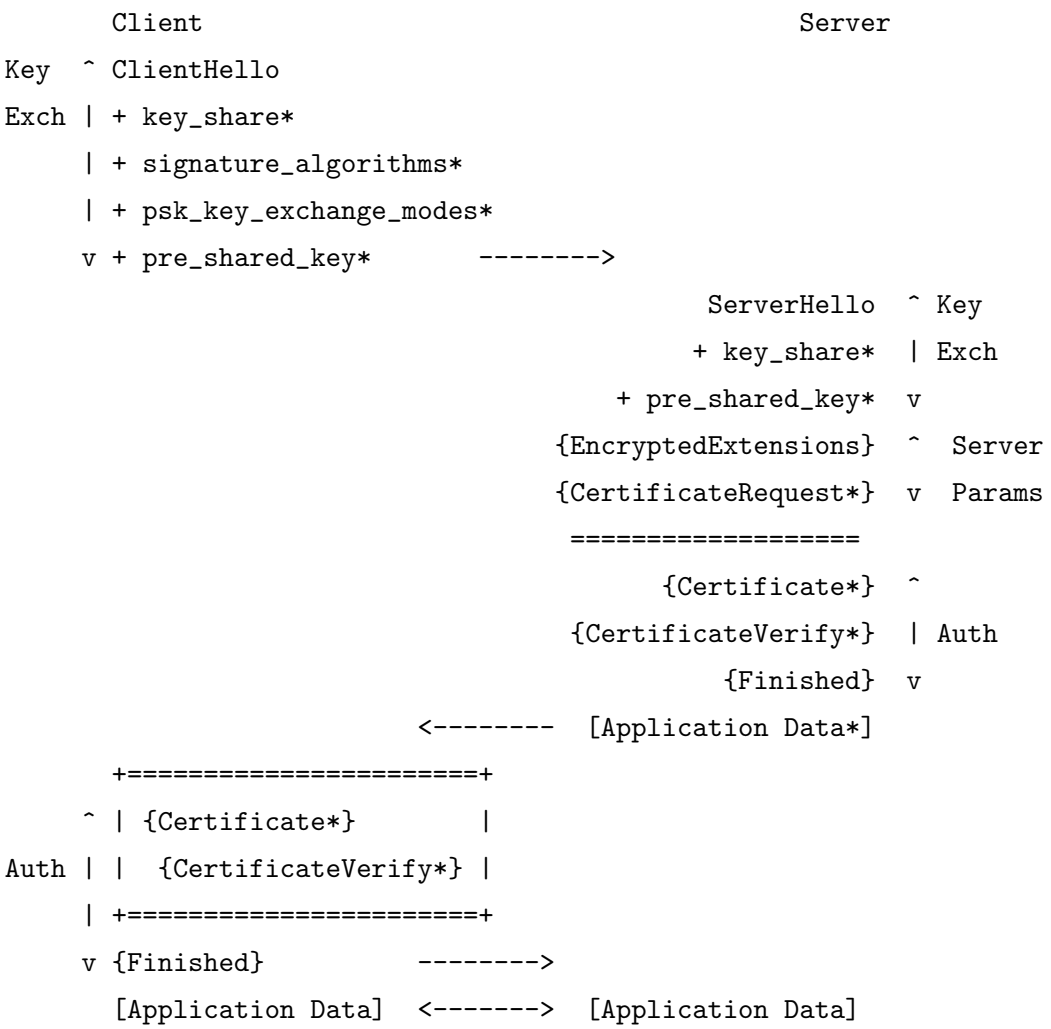
3. overjanje (ang. *authentication*): Pride do overjanja strežnika (ter odjemalca, če je to zahtevano) ter potrdi se ustreznost ključa in celovitost rokovanja.

Za generiranje ključev seje za varno povezavo odjemalec izbira med dvema možnostima:

- S pomočjo strežniškega javnega ključa šifrira naključno število, ter rezultat pošlje strežniku, ki lahko rezultat dešifrira s pomočjo svojega zasebnega ključa. Obe strani povezave uporabita to naključno število za šifriranje in dešifriranje vseh nadaljnih podatkov,

- uporabi Diffie-Hellman [25] izmenjavo ključev za varno generiranje naključnega in unikatnega ključa seje za šifriranje ter dešifriranje podakov v nadaljnji izmenjavi podatkov. Taka povezava ima še dodatno lastnost, t.j. vnaprejšnja tajnost (ang. *forward secracy*) [40]. Četudi bi bil strežnikov zasebni ključ kdaj razkrit, ključev prejšnjih sej ni mogoče razkriti in tako podatki, preneseni v prejšnjih sejah ostanejo varni. To

dosežemo tako, da za vsako novo sejo generiramo nov unikatni ključ seje.



Slika 2.5: Diagram rokovanja v postopku vzpostavitve TLS povezave

2.3.1 Digitalna potrdila

Za overjanje obeh članov povezave, tako strežnika kot odjemalca, uporabljamo digitalna potrdila. Digitalno potrdilo [29] ali potrdilo javnega ključa je elektronski dokument, ki ga uporabljamo za dokazovanje lastnika javnega ključa. Vsebuje informacije o ključu, informacije o lastniku ključa ter digitalni podpis izdajatelja, ki jamči legitimnost vsebine digitalnega potrdila. Če je podpis veljaven, ter program, ki preverja digitalno potrdilo, zaupa izdajatelju, ki je izdal potrdilo, se lahko digitalno potrdilo uporabi pri vzpostavljanju varne povezave TLS. Običajno je izdajalec digitalnih potrdil certifikatska agencija – CA, ki svojim strankam podpisuje potrdila v zameno za plačilo.

Brskalniki hranijo seznam globalnih korenskih potrdil, ki jim zaupajo. Ob dostopu do spletne strani, ki ima omogočen HTTPS protokol, zahtevajo njeno digitalno potrdilo. Ko le-to prejmejo, preverijo, ali je bilo podpisano s strani katerega korenskega CA oziroma enega od vmesnih CA na poti overjanja. Pot overjanja je sestavljena iz ene korenske CA, ter nič ali več nivojev CA-jev [42].

2.3.2 Overjanje uporabnika z uporabo uporabnikovega digitalnega potrdila

Običajno je pri rokovanju ob vzpostavitvi povezave TLS zahtevano le overjanje strežnika. Protokol TLS pa omogoča tudi overjanje odjemalca. V primeru, ko recimo želimo določenim odjemalcem na strani strežnika omejiti pravice dostopa, želimo uporabnika overiti. Če nam uporabniško ime in geslo ne zadoščata, uporabimo overjanje z uporabo uporabnikovega digitalnega potrdila. TLS rokovanje se izvede po postopku, prikazanemu na shemi [Slika 2.5]. Sporočila, pomembna za overitev uporabnika so podčrtana oziroma obkrožena.

Strežnik pošlje sporočilo „CertificateRequest“ odjemalcu. Odjemalec odgovori s sporočilom „CertificateVerify“. Le-to je niz, podpisan z zasebnim ključem odjemalca, povezan z digitalnim potrdilom, ki ga odjemalec pošlje

strežniku. Strežnik preveri legitimnost digitalnega potrdila ter skladnost sporočila „CertificateVerify“, ter v primeru, da je vse v redu, odjemalcu omogoči vzpostavitev varne povezave.

Overjanje odjemalca vedno zahteva strežnik. Mehanizma v okviru protokola TLS, ki bi odjemalcu omogočal, da strežniku ponudi svoje digitalno potrdilo, ni [42].

2.3.3 SSL pripenjanje digitalnih potrdil

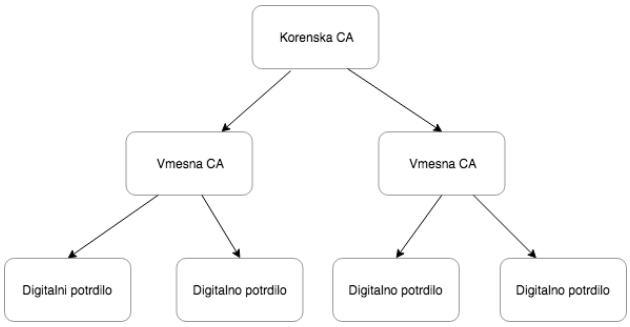
Da se povezava TLS lahko vzpostavi, je potrebno zaupanje. Ob TLS rokovanju strežnik odjemalcu pošlje svoje digitalno potrdilo. Odjemalec preveri, ali je digitalno potrdilo izdala certifikatska agencija, ki ji operacijski sistem na napravi odjemalca zaupa (je v seznamu korenskih CA). Pri razvoju aplikacij (največkrat mobilnih) kot dodatno varnostno zagotovilo uporabljamo SSL pripenjanje. Aplikaciji priložimo strežniško digitalno potrdilo (ali več). Ob vsaki vzpostavitvi nove povezave TLS aplikacije s strežnikom dodatno preverimo, da je digitalno potrdilo, ki ga je poslal strežnik identično digitalnemu potrdilu, ki je bilo „pripeto“ aplikaciji. S tem povezavo še dodatno zaščitimo pred napadi MITM [34], kjer napadelec prestrega (ter mogoče spreminja) sporočila med dvema členoma povezave.

SSL pripenjanja [28] sicer ne moremo uporabiti v vsaki aplikaciji, vendar je uporaba priporočjiva, kjer je to možno. Poskrbeti moramo tudi za periodično posodabljanje „pripetega“ digitalnega potrdila, saj le-temu s časom preteče veljavnost.

2.3.4 Veriga overjanja oz. pot overjanja (ang. *certificate chain/certification path*)

Naprava (odjemalec), ki preverja veljavnost digitalnega potrdila ima shranjen seznam korenskih CA, ki jim zaupa. Vsa digitalna potrdila niso podpisana s strani korenskih CA, ampak so jih podpisala druge vmesne CA. Vmesne CA imajo pooblastilo, da lahko podpisujejo digitalna potrdila ostalim entitetam,

ter njihovo istovetnost in legitimnost jamči CA nad njo. Pot overjanja je tako seznam vseh vmesnih CA, od končne CA do korenske CA. Na sliki [Slika 2.6] je prikazano kako izgleda hiearhija digitalnih potrdil z globino 2. Pot overjanja bi tako vsebovala končno digitalno potrdilo, vmesno digitalno potrdilo ter korensko digitalno potrdilo.



Slika 2.6: Primer hiearhije poti overjanja

Primer: na spletnem strežniku, ki uporablja HTTPS protokol in ima digitalno potrdilo podpisano s strani enega izmed vmesnih CA moramo pri konfiguraciji strežnika dodati še pot overjanja. Ta vsebuje končno digitalno potrdilo strežnika, digitalno potrdilo vmesne CA ter korensko digitalno potrdilo. V primeru, da poti overjanja na strežniku ne dodamo, je pot zaupanja prekinjena in našemu strežniku uporabnik ne bo zaupal [42].

2.3.5 Samo-podpisana digitalna potrdila

Digitalna potrdila niso vedno nujno podpisana s strani zaupanja vredne CA. Digitalno potrdilo lahko podpišemo tudi sami, vendar takega potrdila ni priporočljivo uporabljati v omrežju, ki je dostopno tudi drugim. Takšna potrdila večinoma uporabljamo le pri razvoju ter testiranju novih aplikacij, da nam ni potrebno plačati certifikatski agenciji za podpis digitalnega potrdila. Naprave takim digitalnim potrdilom ne zaupajo (saj v cerifikatski poti ne zaupajo nobenemu), zato moramo za vzpostavitev zaupanja poskrbeti sami [45].

2.4 Vtičnik „cordova-plugin-advanced-http“

Vtičnik [30] je namenjen ogrodju Cordova ter posledično tudi ogrodju Ionic. Doda možnost vzpostavitve povezave HTTP ter HTTPS za platformi Android ter iOS. Na voljo je na Github platformi [26] ter v NPM upravljalniku paketov [37].

Ionic/Cordova vtičnik deluje kot vmesnik med operacijskim sistemom ter aplikacijo. Z Ionic aplikacije ne moremo neposredno dostopati do funkcionalnosti, kot so kamera, internetni dostop, senzorji ter ostalo. Zato moramo naši Ionic aplikaciji dodati vtičnik. V splošnem je vtičnik je sestavljen iz več delov:

- Programske kode v programskem jeziku JavaScript, ki definira vmesnik med vtičnikom ter ogrodjem. Tukaj so definirane metode, ki jih lahko kličemo iz ogrodja Ionic/Cordova, vtičnik pa bo nato klical ustrezno kodo glede na platformo, za katero izdelujemo aplikacijo,
- programske kode v programskem jeziku Java, ki služi implementaciji funkcionalnosti na platformi Android,
- programske kode v programskem jeziku Objective-C, ki služi implementaciji funkcionalnosti na platformi iOS,
- raznih konfiguracijskih datotek, ki definirajo delovanje vtičnika (npr. plugin.xml, kjer definiramo platformno-specifično konfiguracijo, kakšne pravice potrebuje vtičnik za delovanje itd.).

Vtičnik omogoča naslednje metode, ki jih lahko kličemo iz ogrodja Ionic, ter tako upravljamo z delovanjem vtičnika:

- getBasicAuthHeader(username, password): vrne glavo HTTP/HTTPS pri uporabi overjanja z uporabniškim imenom in geslom (ang. *header*),
- useBasicAuth(username, password): omogoči overjanje z uporabniškim imenom in geslom,

- getHeaders(host): vrne HTTP/HTTPS glavo za določeno ime gostitelja (ang. *hostname*),
- setHeader(host, header, value): nastavi glavo (ang. *header*) za vse nadaljne zahtevke HTTP/HTTPS,
- getDataSerializer(): vrne ime serializatorja za pretvorbo podatkov,
- setDataSerializer(serializer): nastavi, v kakšnem formatu bodo podatki v telesu zahtevka pri vseh nadaljnih POST in PUT zahtevkih,
- setCookie(url, cookie): nastavi piškotke,
- clearCookies(): zbriše vse piškotke,
- removeCookies(url, cb): zbriše piškotke za določen URL,
- getCookieString(url): vrne niz za piškotke za določen URL,
- getRequestTimeout(): vrne, koliko časa vtičnik čaka na povezavo, preden je le-ta terminirana,
- setRequestTimeout(timeout): nastavi koliko časa vtičnik čaka na povezavo, preden je le-ta terminirana,
- setSSLCertMode(mode): spremeni nastavitve povezave SSL/TLS. Na voljo imamo več načinov:
  - default: privzeto upravljanje povezave SSL/TLS, z uporabo sistemskih CA,
  - nocheck: onemogočimo preverjanje digitalnih potrdil, namenjeno le za namene testiranja,
  - pinned: naprava zaupa pripetim digitalnim potrdilom.
- disableRedirect(disable): onemogoči avtomatsko preusmerjanje spletnih strani,

- post(url, body, headers): vrne rezultat zahtevka POST na določen url,
- get(url, parameters, headers) : vrne rezultat zahtevka GET na določen url,
- put(url, body, headers) : vrne rezultat zahtevka PUT na določen url,
- patch(url, body, headers) : vrne rezultat zahtevka PATCH na določen url,
- delete(url, parameters, headers) : vrne rezultat zahtevka DELETE na določen url,
- head(url, parameters, headers) : vrne rezultat zahtevka HEAD na določen url,
- uploadFile(url, body, headers, filePath, name) : vrne rezultat zahtevka UPLOADFILE na določen url,
- downloadFile(url, body, headers, filePath) : vrne rezultat zahtevka DOWNLOADFILE na določen url.

Vtičnik lahko našemu projektu dodamo na dva načina, glede na to, ali je vtičnik na voljo v upravljalniku paketov NPM ali ne:

- Če je vtičnik na voljo prek NPMja, lahko uporabimo naslednji ukaz kjerkoli v datotečni strukturi našega Ionic projekta:

  `ionic cordova plugin add << ime vtičnika >>`

  Ko je vtičnik enkrat nameščen, ga iz ogrodja Ionic uporabljamo na naslednji način:

```
1  import { HTTP } from '@ionic-native/http';
2  constructor(private http: HTTP) {}
3
4  ...
5
6  this.http.get('http://ionic.io', {}, {})
7    .then(data => {
8      console.log(data.status);
9      console.log(data.data); // data received by server
10     console.log(data.headers);
11   })
12   .catch(error => {
13     console.log(error.status);
14     console.log(error.error); // error message as string
15     console.log(error.headers);
16   });
```

- V nasprotnem primeru, ko recimo uporabljamo svoj vtičnik, oziroma smo sami spremenili že obstoječ vtičnik, le tega Ionic projektu dodamo s spodnjim ukazom:

  `ionic cordova plugin add << pot do vtičnika na disku >>`

  Iz Ionic ogrodja ga uporabljamo na nekoliko drugačen način, in sicer:

```
1  cordova.plugin.http.get("http://ionic.io", {}, {}, function(ret){
2        console.log("returned data: ", ret);
3      }, function(err){
4        console.log("error: ", err);
5      });
```

V obeh zgornjih primerih naredimo GET zahtevek na spletno stran „http://ionic.io“, ter rezultat zahtevka zapišemo v konzolo. V primeru, da je spletna stran vrnila odgovor s kodo 200, se v konzolo izpiše vsebina spletne strani, v nasprotnem primeru pa sporočilo napake.

V prvem primeru uporabljamo obljube (ang. *promises*), kjer odgovor pridobimo s pomočjo metod *then*, oziroma v primeru napake pa uporabimo metodo *catch*. V drugem primeru pa uporabljamo povratne klice (ang. *callback*), ki so v JavaScript svetu uporabljeni pri asinhronih klicih[1].

Vtičnik „cordova-plugin-advanced-http“ je precej obsežen, saj je že samo za platform Android več kot 15 datotek z kodo, ter približno 4500 vrstic kode. Zato bom strukturo vtičnika ter zaporedje akcij poenostavljeno prikazal kar na primeru.

Želimo izvesti HTTP GET zahtevek na določeno spletno stran na platformi Android. Pri tem ne želimo, da bi nas spletna stran preusmerila na kakšno drugo [Slika 2.7]:

1. Vtičnik dodamo našemu Ionic projektu (opisano zgoraj),

2. Iz Ionic projekta kličemo metodo *disableRedirect(true)*. Vtičnik preveri katero metodo želimo uporabiti, ter glede na navodila v datoteki „advanced-http.js“ izvede del kode, glede na to, za katero platformo izdelujemo aplikacijo. V našem primeru kliče Javin razred *CordovaHttpPlugin* ter na njem metodo *disableRedirect(true*),

```
1  // datoteka advanced−http.js
2  disableRedirect: function (disable, success, failure) {
3      return exec(success, failure, 'CordovaHttpPlugin', 'disableRedirect', [ !!disable ]);
4    },
```

3. *CordovaHttpPlugin* je Javin razred, ki deluje kot vmesnik med JavaScript ter Java kodo. V našem primeru na razredu *CordovaHttp* kliče

[1] Asinhroni klic (ang. *asynchronous call* je klic metode, ki za izvedbo potrebuje več časa. Asinhroni klic se izvede v ozadju, ter tako ne blokira glavnega programa ter vrne rezultat, ko dokonča z delom. Npr. asinhroni klici se uporabljajo pri prenašanju podatkov s strežnika.

metodo *disableRedirect(true)*. Ta metoda na razredu *CordovaHttp* nastavi statično spremenljivko *disableRedirect* na *true*. Ker je ta spremenljivka po tipu statična, pomeni da bodo isto vrednost te spremenljivke imeli tudi vsi objekti, ki bodo generirani iz tega razreda. Nastavitev, da ne sledimo avtomatskim preusmeritvam spletnih strani je sedaj nastavljena.

```
1 // datoteka CordovaHttp.java
2     public static void disableRedirect(boolean disable) {
3         disableRedirect.set(disable);
4     }
```

4. Iz Ionic ogrodja kličemo še metodo *get*, na enak način kot je prikazano zgoraj. Znova vtičnik uporabi datoteko advanced-http.js ter izvede klic na metodo razreda *CordovaHttpPlugin*, ki ustvari nov objekt *CordovaHttpGet*.

```
1  //datoteka CordovaHttpPlugin.java
2  } else if (action.equals("get")) {
3      String urlString = args.getString(0);
4      Object params = args.get(1);
5      JSONObject headers = args.getJSONObject(2);
6      int timeoutInMilliseconds = args.getInt(3) * 1000;
7      CordovaHttpGet get = new CordovaHttpGet(urlString, params, headers,
8                                              timeoutInMilliseconds, callbackContext);
9
10     cordova.getThreadPool().execute(get);
11 }
```

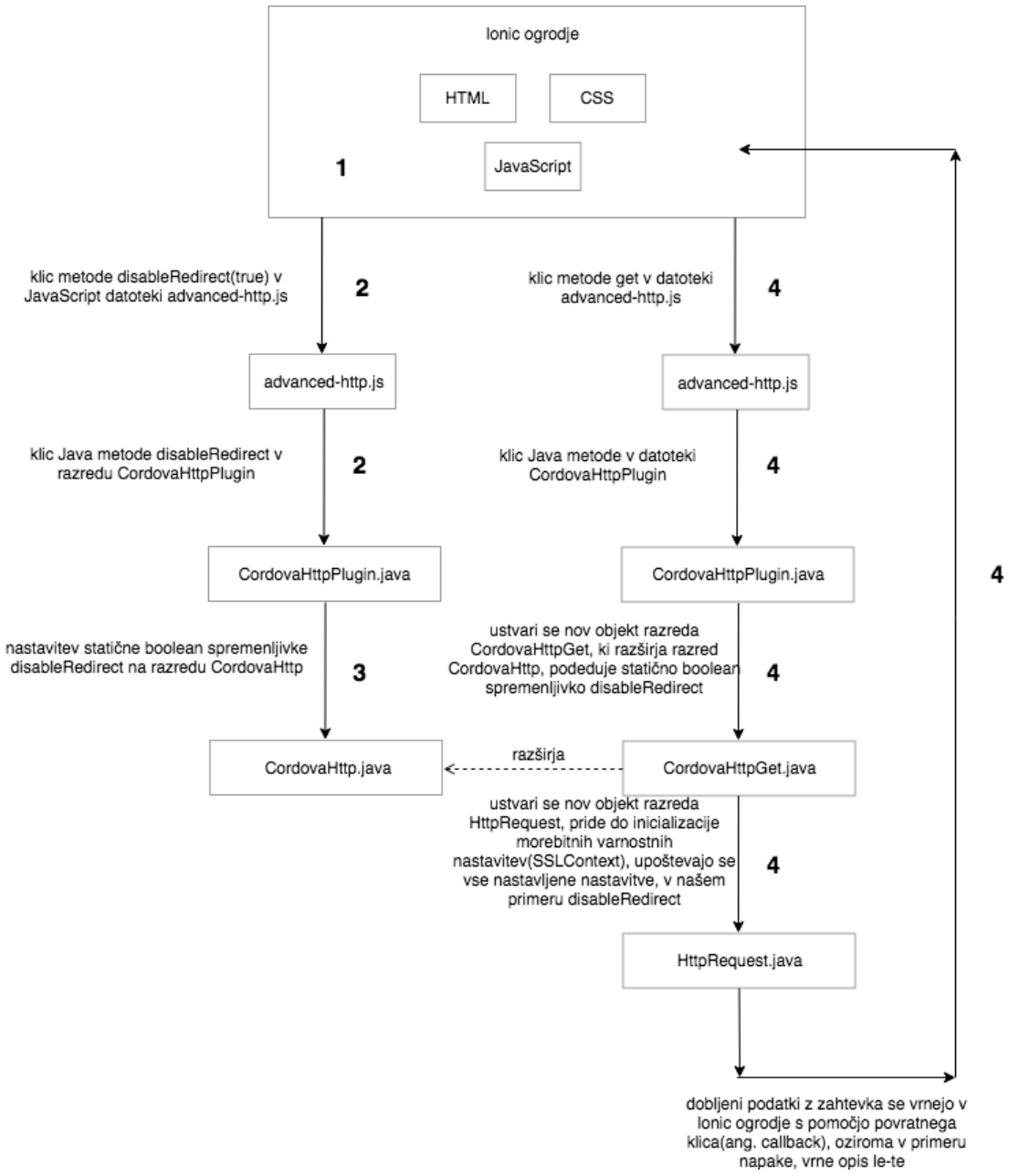
CordovaHttpGet razširja razred *CordovaHttp*, na katerem smo prej nastavili spremenljivko disableRedirect na *true*. Tako ima ta spremenljivka isto vrednost tudi v novo generiranem objektu razreda *CordovaHttpGet*. Ob nastanku tega objekta se ustvari še nov objekt *HttpRequest*, v katerem končno pride do HTTP klica na URL. Pri tem se upoštevajo vse nastavitve, ki smo jih nastavili na razredu *CordovaHttp*, saj so se podedovale vsem nadaljnjim objektom. Odgovor strežnika se

vrne po isti poti nazaj do ogrodja Ionic s pomočjo povratnih klicev (ang. *callback*).

Tak modularen način implementacije omogoča boljšo preglednost kode. Vtičnik je narejen tako, da na osrednjem razredu *CordovaHttp* nastavljamo statične spremenljivke, ki so potem podedovane vseh objektom razreda. Dobra stran tega je, da lahko npr. nastavimo uporabo svojih piškotkov, ter bodo ti uporabljeni pri vseh nadalnjih spletnih zahtevkih. Slaba stran pa je, da se mora za vsak nov zahtevek ustvariti nov objekt, kar pa je lahko zamudno. Pri klicih z HTTP protokolom to ne predstavlja velikega problema.

Če pa želimo uporabiti varnejšo povezavo HTTPS, se ob vsakem novem zahtevku HTTPS vzpostavi nov SSL kontekst[1]. Če bi moral operacijski sistem naprave za vsak nov zahtevek HTTPS vzpostaviti novo povezavo TLS s strežnikom, bi to za to porabilo (pre)veliko časa. Operacijski sistem sam predvidi, da bo ista povezava TLS še uporabljena v prihodnosti, zato povezave ne zapre takoj po prejetem odgovoru strežnika. Tako se kljub temu, da se z vsakim novim klicem ustvari nov SSL kontekst[1], znova uporabi povezava TLS. Tako se zaporedni zahtevki HTTPS na isti strežnik opravijo v sprejemljivem času.

[1]SSL kontekst (ang. *SSL Context*) je razred programskega jezika Java, ki hrani vse podatke, ki so potrebni za vzpostavitev SSL/TLS povezave.



Slika 2.7: Zaporedje klicev pri HTTP zahtevku brez preusmerjanja

Poglavje 3

Načrt izdelave

3.1 Zahteve in načrt dela

Cilj diplomske naloge je, da vtičniku dodamo možnost povezovanja tudi s strežniki, ki zahtevajo, da se uporabnik overi z uporabo uporabnikovega digitalnega potrdila (ang. *client certificate authentication*). Ne osredotočamo se na to, kako uporabnik pridobi digitalno potrdilo. Predpostavljamo, da ima le tega shranjenega v notranjem pomnilniku svoje naprave. Prav tako moramo originalni vtičnik ohraniti karseda splošen, da ga tako lahko uporabi čimvečje število razvijalcev in omogoča preprosto dodajanje funkcionalnosti.

Vtičnik že ima možnost uporabe SSL pripenjanja (ang. *SSL pinning*). Tak način delovanja iz Ionic projekta omogočimo z klicem metode *setSSLCertMode("pinned")*. Vtičnik nato ta digitalna potrdila (lahko jih je več) prebere iz specifične mape *www/certificates* v Ionic projektu, kamor jih postavi razvijalec aplikacije. Da bi ohranil splošnost ter dal razvijalcu več možnosti pri uporabi vtičnika, smo se odločili, da za vklop uporabe uporabnikovega digitalnega potrdila (ang. *client certificate*), ne uporabimo metode *setSSLCertMode*. Dodali bomo svojo metodo z imenom *enableClientCert*, ki kot argumente dobi:

- 1. argument: boolean spremenljivka *enable*. Če ima vrednost *true*, se omogoči uporaba uporabnikovega digitalnega potrdila, v nasprotnem

pa se ta funkcionalnost onemogoči,

- 2. argument: boolean spremenljivka *selfSigned*. V primeru da uporabljamo samo-podpisano digitalno potrdilo, ki mu naprava ne zaupa, ob klicu metode drugi argument postavimo na *true*. Do enakega delovanja pridemo, če poleg metode *setSSLCertMode* kličemo tudi metodo *setSSLCertMode* z argumentom *nocheck*. Tak način je v primeru uporabe v produkciji slab, saj napravi omogočimo, da zaupa vsem digitalnim potrdilom, kar pa ni varno. To opcijo uporabljamo samo v namene testiranja aplikacije,

- 3. argument: niz „password“. Vtičnik pri omogočeni uporabi uporabnikovega digitalnega potrdila pričakuje, da sta v mapi *www/assets/clientCertificate* v Ionic projektu dve datoteki. Prva je uporabnikovo digitalno potrdilo, shranjeno kot shramba ključev (ang. *keystore*) s končnico *.p12*. Druga datoteka je pot overjanja, shranjena v formatu .pem. Uporabnikovo digitalno potrdilo, v formatu *.p12*, je mnogokrat zavarovano z geslom, zato sem razvijalcu aplikacije omogočil vpis tega gesla. V primeru da uporabnikovo digitalno potrdilo ni zavarovano z geslom, lahko kot 3. argument vpiše kar prazen niz.

Metodo *enableClientCert* lahko kličemo neodvisno od metode *setSSLCertMode*. Tako se lahko razvijalec sam odloči kaj bo uporabljal pri dostopu do strežnika. Omogoči lahko bodisi le SSL pripenjanje, bodisi le uporabo uporabnikovega digitalnega potrdila, lahko pa tudi uporablja oboje, ter tako še dodatno ojača varnost in zaupnost povezave med napravo in strežnikom. Seveda mora v takem primeru tudi strežnik omogočati takšno vrsto povezave.

V primeru, da bo želel razvijalec uporabiti overjanje uporabnika z digitalnim potrdilom, bo iz Ionic ogrodja to možnost vklopil na naslednji način:

```
1 cordova.plugin.http.enableClientCert(true, true, "niz z geslom", function(ret) {
2       console.log("success", ret);
3   }, function(fail){
4       console.log("fail", fail);
5   });
```

Vsi nadaljni zahtevki bodo tako uporabljali tudi overjanje uporabnika, če pri vklopu te funkcionalnosti ne bo prišlo do nobene napake. Vzpostaviti bomo morali tudi testni strežnik. Ta bo moral omogočati povezavi HTTP ter HTTPS. Prav tako bo moral imeti možnost overjanja uporabnika z uporabnikovim digitalnim potrdilom. Strežnik bomo uporabili za testiranje delovanja vtičnika.

3.2 Načrt ovrednotenja

Delovanje vtičnika bomo preverili z razvojem testne aplikacije, kjer bomo merili čas, ki ga vtičnik potrebuje pri zahtevkih HTTP ter HTTPS. Hitrost vtičnika bomo primerjali z Pythonovo knjižnico *Requests* [19]. Podali bomo ugotovitve ter se poskušali opredeliti ali je takšna hitrost ustrezna za uporabo v pravi aplikaciji. Uporabnost vtičnika bomo testirali z uporabo vtičnika v pravi mobilni aplikaciji, namenjeni prikazu podatkov o električni porabi gospodinjstva uporabnika.

Poglavje 4

Izvedba

Najprej smo vzpostavili testni strežnik. Pri tem smo uporabili Pythonovo knjižnico *Flask* [11]. Strežnik ima 3 načine delovanja:

- kot navaden HTTP strežnik
- kot HTTPS strežnik
- kot HTTPS strežnik, ki uporabnika overi z uporabo uporabnikovega digitalnega potrdila

Za generiranje digitalnih potrdil smo uporabili odprtokodno knjižnico OpenSSL [16]. Vsa digitalna potrdila so samo-podpisana, zato smo morali za upravljanje zaupanja poskrbeti sami.

Ko smo vzpostavili strežnik, smo začeli z delom na vtičniku. Vtičnik je precej obsežen in ponuja veliko funkcionalnosti. Preden smo mu lahko karkoli dodajali, smo se morali najprej spoznati s tem, kako je vtičnik zasnovan, kaj ponuja ter kako se ga uporablja. Z Ionic ogrodjem smo naredili preprosto testno aplikacijo, ter vanjo vstavili vtičnik. Poskušali smo dostopati do podatkov na testnem strežniku, uporabiti tako HTTP kot HTTPS povezavo, preveriti hitrost delovanja ter omogočiti SSL pripenjanje. Preizkusili smo tudi več vrst zahtevkov (get, post, download, itd.) ter se najbolj osredotočil na dele vtičnika, povezane z vzpostavljanjem povezave HTTPS. Testirali smo na fizični Android napravi.

Ko smo dobro razumeli, kaj vtičnik ponuja, smo začeli z dopolnjevanjem vtičnika, da bo omogočal tudi uporabo povezave HTTPS, ki poleg preverjanja strežnika s strani naprave, preveri tudi napravo na strani strežnika z uporabo uporabnikovega digitalnega potrdila. Ker je bilo kode v vtičniku enostavno preveč, ter je bila preveč razkropljena, veliko izkušenj s takšnim delom pa nismo imeli, smo uporabo uporabnikovega digitalnega potrdila najprej testirali v testni aplikaciji. Tako smo z integriranim razvojnim okoljem Android Studio razvili aplikacijo, kjer smo se z uporabo uporabnikovega digitalnega potrdila poskušali povezati s testnim strežnikom. Tako smo dobil nekoliko občutka, kako se takšna povezava vzpostavi in kaj je potrebno, da naprava strežniku zaupa ter mu pošlje svoje digitalno potrdilo.

Še vedno nismo uspeli funkcionalnosti vstaviti neposredno v vtičnik, zato smo razvili svoj testni Ionic vtičnik, ki je omogočal samo dostop do strežnika z zahtevkom GET z uporabo uporabnikovega digitalnega potrdila. Tako smo se spoznali s strukturo Ionic/Cordova vtičnika, kako se ga načrtuje ter kakšen je tok podatkov in klicev v vtičniku. Kljub mnogim problemov s konsistentnostjo delovanja samega Ionic ogrodja nam je vtičnik uspelo narediti, vendar je omogočal samo osnovno funkcionalnost. Torej, možni so bili le zahtevki GET, upravljanja s piškotki ni bilo, overjanja z uporabniškim imenom in geslom tudi ne. Osredotočili smo se samo na pošiljanje uporabnikovega digitalnega potrdila strežniku pri vzpostavitvi seje TLS.

Nato smo začeli z implementacijo funkcionalnosti v vtičnik „cordova-plugin-advanced-http“, za platformo Android. Pregledali smo celotno kodo in poskušali ugotoviti kje vse bomo morali kodo dodati oziroma spremeniti. Najprej smo možnost uporabe uporabnikovega digitalnega potrdila želeli dodati že obstoječi metodi *setSSLCertMode*, vendar smo se po tehtnem razmisleku odločili za implementacijo nove metode *enableClientCert*. Tako je razvijalcu aplikacije omogočena hkratna uporaba tako SSL pripenjanja kot overjanje z uporabo uporabnikovega digitalnega potrdila. Če bi uporabili kar metodo *setSSLCertMode*, bi na tak način nekoliko zakomplicirali uporabo večih načinov vzpostavitve povezave TLS hkratu.

4.1 Postopek delovanja vtičnika ob overjanju uporabnika z uporabnikovim digitalnim potrdilom

Vtičnik v primeru uporabe overjanja z uporabnikovim digitalnim potrdilom deluje po naslednjem postopku:

1. Razvijalec iz Ionic ogrodja kliče metodo *enableClientCert* s prvim argumentom *true*. V primeru da uporabljamo digitalno potrdilo, ki je podpisano s strani certifikatske agencije, je drugi argument nastavljen na *false*. Če uporabljamo samo-podpisano digitalno potrdilo, je drugi argument *true*. Tretji argument je niz z geslom, ki je uporabljeno za odklepanje uporabnikovega digitalnega potrdila.

2. Vtičnik v datoteki *advanced-http.js* preveri katero metodo kličemo, ter kliče metodo *enableClientCert* na Java razredu „CordovaHttpPlugin“. Ob klicu poda še argumente.

```
1 // Ogrodje Ionic
2 cordova.plugin.http.enableClientCert(true, true, "niz z geslom", function(ret) {
3         console.log("success", ret);
4       }, function(fail){
5         console.log("fail", fail);
6       });
```

3. Vtičnik v datoteki *advanced-http.js* preveri katero metodo kličemo, ter kliče metodo *enableClientCert* na Java razredu *CordovaHttpPlugin*. Ob klicu poda še argumente.

```
1 // Datoteka advanced−http.js
2 disableRedirect: function (disable, success, failure) {
3       return exec(success, failure, 'CordovaHttpPlugin', 'disableRedirect', [ !!disable ]);
4     },
```

4. V razredu *CordovaHttpPlugin* se najprej kliče metoda *loadClientCert* ki iz datotečnega sistema naprave prebere uporabnikovo digitalno potrdilo ter pot overjanja. Vtičnik pričakuje, da sta obe datoteki v mapi *www/assets/clientCertificate* v Ionic projektu. Uporabnikovo digitalno potrdilo mora v imenu datoteke imeti niz „client“, pot overjanja pa niz „chain“. Za branje je uporabljen Java razred *BufferedInputStream*. V primeru, da pride pri inicializaciji *BufferInputStream* do napake (digitalnega potrdila oz. poti overjanja ni na napravi, je v napačnem formatu, itd.), vtičnik vrne informacije o napaki nazaj v Ionic ogrodje s pomočjo povratnih klicev. V nasprotnem primeru oba vhodna tokova shrani kot statično spremenljivko na razredu *HttpRequest*.

```
1  // Datoteka CordovaHttpPlugin.java
2      //Add client certificate file, if the name contains string "client"
3      //Add certificate chain file, if the name contains string "chain"
4  private void loadClientCert() throws GeneralSecurityException, IOException {
5          AssetManager assetManager = cordova.getActivity().getAssets();
6          String[] tempFiles = assetManager.list("www/assets");
7
8          String[] files = assetManager.list("www/assets/clientCertificate");
9          String clientCertFile = "";
10         String certChainFile = "";
11         for (int i = 0; i < files.length; i++) {
12           if (files[i].toLowerCase().contains("client")) {
13             clientCertFile = "www/assets/clientCertificate/" + files[i];
14           } else if (files[i].toLowerCase().contains("chain")) {
15             certChainFile = "www/assets/clientCertificate/" + files[i];
16           }
17         }
18
19         InputStream in = cordova.getActivity().getAssets().open(clientCertFile);
20         InputStream buffIn = new BufferedInputStream(in);
21         HttpRequest.addClientCert(buffIn);
22
23         in = cordova.getActivity().getAssets().open(certChainFile);
24         buffIn = new BufferedInputStream(in);
```

```
25            HttpRequest.addChainCert(buffIn);
26        }
```

Po tem se iz razreda *CordovaHttpPlugin* kliče še metoda *setClientCertEnabled* na razredu *HttpRequest*. Ob klicu se prenesejo tudi vsi trije argumenti, ki jih je razvijalec aplikacije dal ob klicu iz Ionic ogrodja.

```
1  // Datoteka CordovaHttpPlugin.java
2  } else if (action.equals("enableClientCert")){
3              boolean enable = args.getBoolean(0);
4              boolean selfSigned = args.getBoolean(1);
5              String clientCertPassword = args.getString(2);
6              try {
7                this.loadClientCert();
8                HttpRequest.setClientCertEnabled(enable, selfSigned, clientCertPassword);
9                callbackContext.success();
10             } catch (Exception e){
11               e.printStackTrace();
12               callbackContext.error("There was an error setting up client certificate
13                                                  setting. Make sure you have client
14                                                  and chain certificates in the folder
15                                                  www/assets/clientCertificate/");
16             }
17 }
```

5. V razredu *HttpRequest* se v statični spremenljivki *CLIENT_KEY_ MANAGERS* ter *CLIENT_TRUST_MANAGERS* shranijo podatki o uporabnikovem digitalnem potrdilu ter poti overjanja. Podatki o ključih se shranijo v seznam upravljalcev s ključi (ang. *key managers*), pot overjanja pa v seznam upravljalcev zaupanja (ang. *trust managers*). V primeru, da razvijalec na vtičniku omogoči tudi SSL pripenjanje, se poleg poti overjanja v seznamu upravljalcev zaupanja (ang. *trust managers*) shranijo tudi digitalna potrdila iz datotečnega sistema v mapi www/certificates.

```
1  // Datoteka HttpRequest.java
2    // Only called when client authentication is enabled, but pinning isn't
```

```
3   private static void getClientTrustAndKeyManagers() throws IOException {
4     try {
5
6       CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509");
7       X509Certificate chainCert = (X509Certificate) certificateFactory
8                                                     .generateCertificate(CHAIN_CERT_IS);
9       String alias = chainCert.getSubjectX500Principal().getName();
10
11      KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
12      trustStore.load(null);
13      trustStore.setCertificateEntry(alias, chainCert);
14
15      KeyStore keyStore = KeyStore.getInstance("PKCS12");
16      keyStore.load(CLIENT_CERT_IS, CLIENT_CERT_PASSWORD.toCharArray());
17
18      KeyManagerFactory kmf = KeyManagerFactory.getInstance("X509");
19      kmf.init(keyStore, CLIENT_CERT_PASSWORD.toCharArray());
20      KeyManager[] keyManagers = kmf.getKeyManagers();
21
22      TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
23      tmf.init(trustStore);
24      TrustManager[] trustManagers = tmf.getTrustManagers();
25
26      CLIENT_KEY_MANAGERS = keyManagers;
27      CLIENT_TRUST_MANAGERS = trustManagers;
28
29    } catch (GeneralSecurityException e) {
30      IOException ioException = new IOException("Security exception configuring
31                                                SSL trust managers");
32      ioException.initCause(e);
33      e.printStackTrace();
34      throw new HttpRequestException(ioException);
35    }
36  }
```

6. Naslednji korak je inicializacija SSL konteksta[1], ki je kasneje uporabljen pri vzpostavitvi same povezave HTTPS. Pri inicializaciji so uporabljeni upravljalci ključev ter upravljalci zaupanja, ki so bili ustvarjeni v koraku 4. Ko je SSL kontekst[1] inicializiran, iz njega pridobimo podatkovno strukturo *SOCKET_FACTORY*. Le-ta je shranjena v statično spremenljivko na razredu *HttpRequest*, imenovano *SOCKET_FACTORY*.

```
1 // Datoteka HttpRequest.java
2       SSLContext context = SSLContext.getInstance("TLS");
3       context.init(CLIENT_KEY_MANAGERS, CLIENT_TRUST_MANAGERS,
4                       new SecureRandom());
5       return context.getSocketFactory();
```

7. Sedaj ima vtičnik vse, kar je potrebno za vzpostavitev varne povezave HTTPS z uporabo uporabnikovega digitalnega potrdila. Ob vseh naslednjih zahtevkih GET, ki imajo v URLju niz „https“, se pri vzpostavitvi povezave uporabi *SOCKET_FACTORY*, ki je bil ravnokar inicializiran.

```
1 // Datoteka HttpRequest.java
2       HttpsURLConnection.setDefaultSSLSocketFactory(SOCKET_FACTORY);
3       ((HttpsURLConnection) connection).setSSLSocketFactory(SOCKET_FACTORY);
```

V primeru, da pride pri konfiguraciji delovanja vtičnika, oziroma pri zahtevkih HTTPS/HTTP do napake, moramo o tem obvestiti razvijalca. Vsa sporočila o napaki se s pomočjo povratnih klicev (ang. *callback*) pošljejo v ogrodje Ionic, prav tako pa se v podrobno poročilo o napaki izpiše v dnevniški izhod (ang. *log output*) Android naprave. Ta sporočila lahko ob testiranju aplikacije pregledujemo s pomočjo konzole programa Android Studio.

Prav tako smo poskrbeli, da razvijalec ob uporabi napačnih uporabnikovih digitalnih potrdil, ali pa da le-teh ni v predvideni mapi, o tem dobi sporočilo v ogrodje Ionic.

[1]SSL kontekst (ang. *SSL Context*) je razred programskega jezika Java, ki hrani vse podatke, ki so potrebni za vzpostavitev SSL/TLS povezave.

4.2 Razhroščevanje

Pri razvoju smo uporabljal programsko opremo Android Studio [6], predvsem funkcionalnost, ki omogoča pregled dnevniških datotek (ang. *log*). Tako smo za razhroščevanje lahko spremljali zapise v dnevniške datoteke ter preverili kje je prišlo do napake pri delovanju. Android Studio smo uporabljali za pregled stanja vtičnika, za delovanje same testne aplikacije pa smo uporabljali orodje za razvijalce v brskalniku Google Chrome. Ker za implementacijo aplikacije pišemo kodo, namenjeno za prikaz na spletnem pogledu (ang.*web view*), lahko podobno kot pri razvoju spletne aplikacije razne informacije izpisujemo v konzolo brskalnika. Naprava, na kateri teče aplikacija mora biti fizično priključena na računalnik. Na URL naslovu „chrome://inspect“ v brskalniku Google Chrome lahko tako pregledujemo kaj se z našo aplikacijo dogaja v realnem času.

Poglavje 5

Ovrednotenje

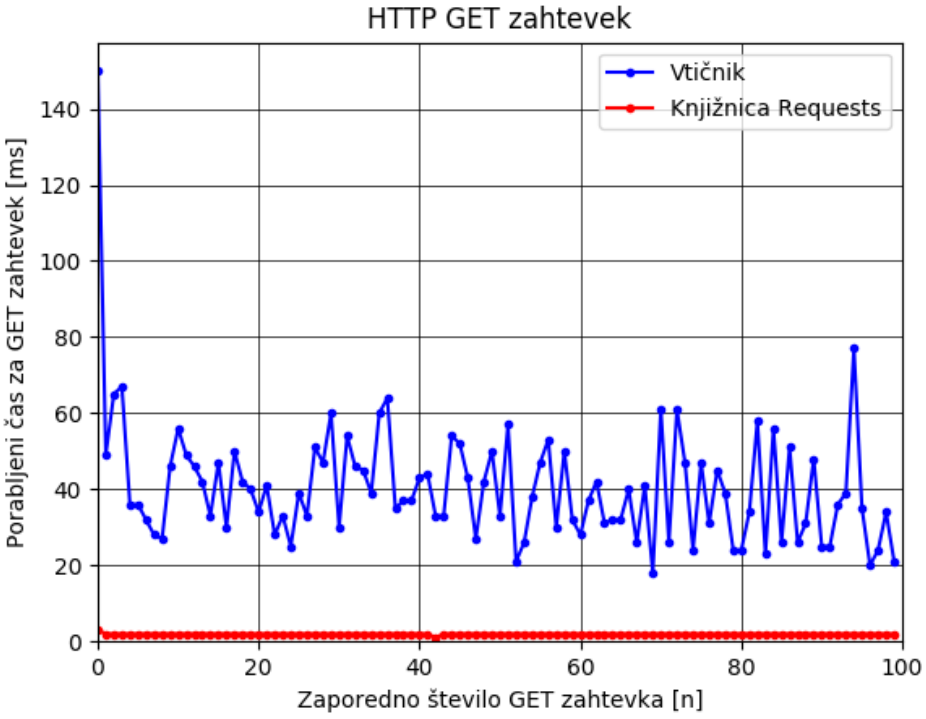
5.1 Testiranje na testni mobilni aplikaciji

Za preverjanje uporabnosti ter delovanja vtičnika smo razvili preprosto aplikacijo za platformo Android, ki uporablja vtičnik za povezovanje s testnim strežnikom. Preveril smo uporabo različnih vrst zahtevkov (GET, POST, PUSH, PATCH, itd.) na testni strežnik z uporabo protokola HTTP. Kot smo pričakovali, tukaj ni bilo nobenih težav, saj dela kode vtičnika, ki skrbi za povezavo HTTP nismo veliko spreminjali. Pomembno je, da vtičnik deluje hitro in zanesljivo. Za boljšo predstavo o tem, kako hiter je naš vtičnik, smo za primerjavo uporabili Python [18] knjižnico *Requests* [19]. Za uporabo te knjižnice smo se odločili, ker jo uporablja veliko razvijalcev, ter velja kot dobra in hitra implementacija funkcionalnosti za delo s povezavo HTTP/HTTPS.

Razvili smo Python [18] skripto, s katero smo pošiljali razne zahtevke na testni strežnik in merili potreben čas za prejem odgovora. Računalnik, na katerem je bila skripta pognana je bil v istem omrežju Wi-Fi kot testni strežnik ter Android naprava, na kateri smo testirali vtičnik. Tako smo omejil morebitna nihanja v internetni povezavi, ter so tako izmerjeni podatki bolj zanesljivi. Potreben čas za prejem odgovora smo merili tudi na Android napravi in časa nato primerjali.

5.1.1 Rezultati testiranja

Navadni HTTP GET zahtevki



Slika 5.1: Meritve hitrosti pri navadnih HTTP GET zahtevkih

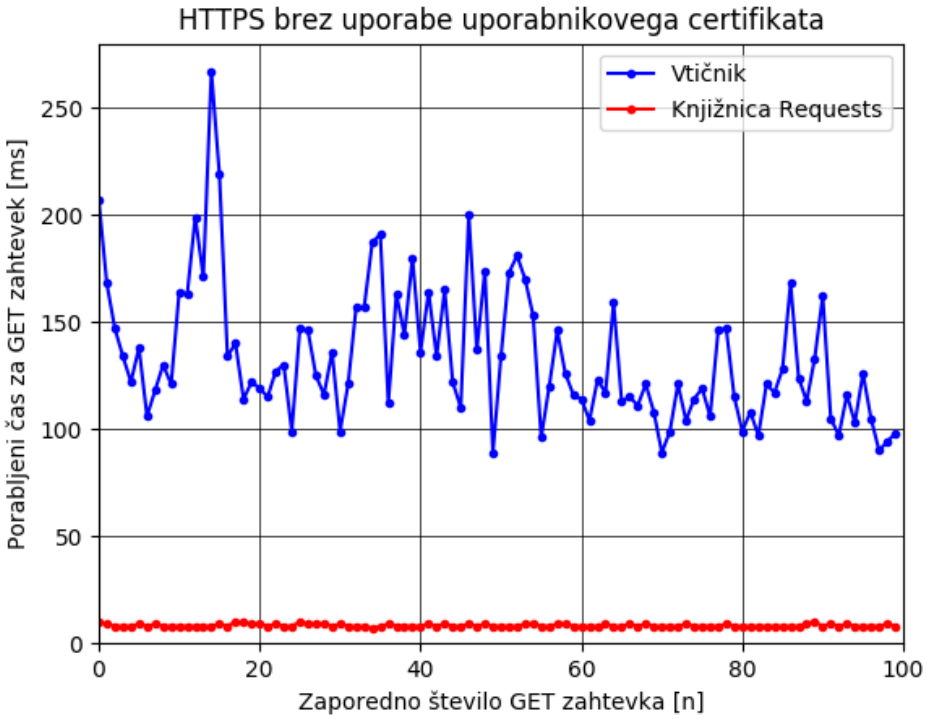
Povprečen čas – vtičnik: 41ms
Povprečen čas – Python knjižnica *Requests*: 2ms

Najprej smo merili čas za odgovor samo z uporabo navadnega GET zahtevka, prek protokola HTTP [Slika 5.2]. Torej brez kakršnekoli varnosti. Tako iz Android naprave in računalnika smo poslali 100 GET zahtevkov, ter merili čas, ki je pretekel od trenutka, ko je bil zahtevek poslan, do prejetega odgovora strežnika. Zahtevke smo izvajali zaporedno, torej naslednji zahtevek je bil poslan šele, ko je naprava dobila odgovor prejšnjega zahtevka. Tako smo onemogočil morebitne napake pri merjenjih, če bi naprava zah-

tevke pošiljala vzporedno. Pomembna je tudi količina podatkov, ki jih vrne strežnik. V našem primeru je strežnik vrnil samo kratek niz z nekaj HTML kode.

Iz grafa je razvidno, da je vtičnik precej počasnejši od knjižnice *Requests*, tako v začetni fazi kot v nadaljnem delovanju. Vtičnik je pri prvem zahtevku potreboval občutno več časa, saj mora aplikacija na mobilni napravi opraviti še veliko klicev na nivo operacijskega sistema. Menimo, da je vtičnik kljub temu nalogo opravil v sprejemljivem času, in bi ga lahko uporabil pri aplikaciji, ki opravlja veliko klicev HTTP. Poskušali smo še ostale tipe zahtevkov (POST, DELETE, itd.), pri katerih smo dobil podobne rezultate.

HTTPS brez uporabe uporabnikovega digitalnega potrdila



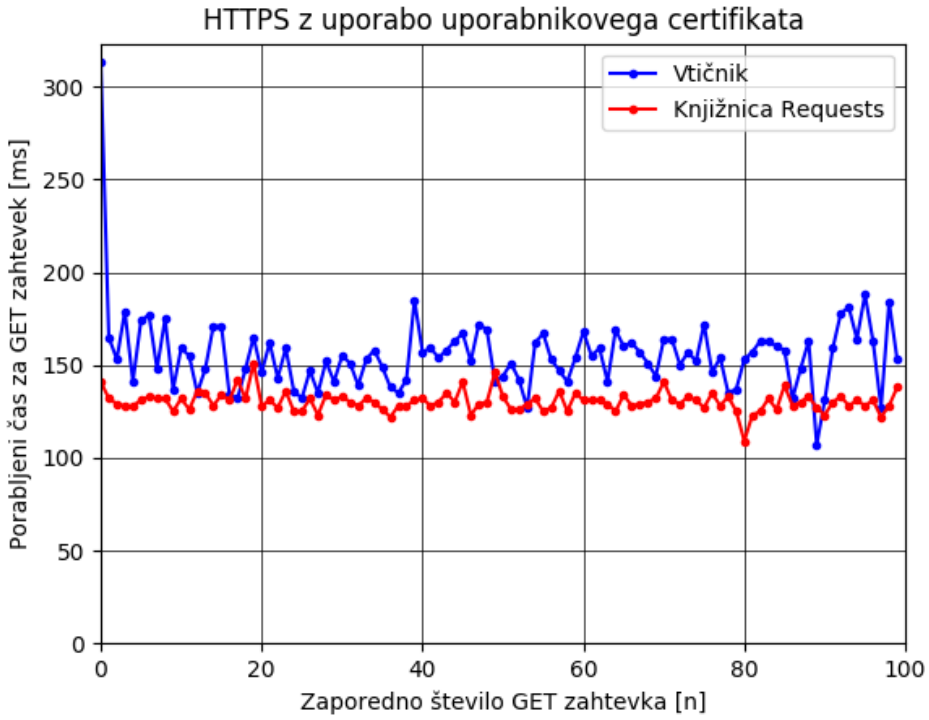
Slika 5.2: Meritve hitrosti pri HTTPS zahtevkih

Povprečen čas – vtičnik: 133ms
Povprečen čas – Python knjižnica *Requests*: 9ms

Isti postopek testiranja smo uporabili tudi za testiranje hitrosti povezave HTTPS [Slika 5.3]. Na svojem testnem strežniku smo omogočili povezavo HTTPS, ter iz mobilne naprave in računalnika poslali 100 zahtevkov GET z uporabo protokola HTTPS.

Rezultati so precej podobni prejšnjim. Vtičnik je precej počasnejši od knjižnice *Requests*, vendar menimo, da 133ms še vedno ni preveč za uporabo v aplikaciji.

HTTPS z uporabo uporabnikovega digitalnega potrdila



Slika 5.3: Meritve hitrosti pri HTTPS zahtevkih, z uporabo uporabnikovega digitalnega potrdila

Povprečen čas – vtičnik : 155ms
Povprečen čas – Python knjižnica *Requests*: 130ms

Preverili smo še uporabo uporabnikovega digitalnega potrdila [Slika 5.1]. Aplikaciji smo priložili uporabnikovo digitalno potrdilo ter pot overjanja. Ker knjižnica *Requests* ne omogoča uporabe digitalnih potrdil, ki so zaščiteni z geslom, smo morali uporabnikovo digitalno potrdilo najprej pretvoriti v drugi format. Tako smo iz ene .p12 datoteke dobil dve. Prva je vsebovala zasebni ključ, druga pa digitalno potrdilo, obe pa sta bili v formatu .pem. Pot overjanja je že bila v formatu .pem. Hitrost vtičnika in knjižnice *Requests* je tukaj precej bolj primerljiva. V splošnem je vtičnik počasnejši le za približno 10ms pri vsakem zahtevku.

Pri prvem zahtevku, ki potrebuje nekaj več časa tudi pri uporabi knjižnice *Requests*, se vzpostavi povezava TLS, ki je nato uporabljena za vse nadalnje zahtevke, in se ne na novo ustvari za vsak zahtevek posebej. V kodi vtičnika se sicer za vsak nov zahtevek ustvari nov SSL kontekst[1], vendar operacijski sistem Android naprave sam prepozna, da gre za isto povezavo, ki jo lahko znova uporabi.

5.2 Testiranje z uporabo vtičnika v mobilni aplikaciji za pregled porabe elekrične energije v gospodinjstvu

Uporabnost vtičnika smo testirali tudi tako, da smo ga uporabili v aplikaciji, ki smo jo razvijali v okviru projekta Flex4Grid [41] na Inštitutu Jožef Stefan. Aplikacija [Slika 13] je namenjena napravam Android, ter je narejena z ogrodjem Ionic.

Na zalednem strežniku se zbirajo podatki o električni porabi uporabnikov.

[1]SSL kontekst (ang. *SSL Context*) je razred programskega jezika Java, ki hrani vse podatke, ki so potrebni za vzpostavitev SSL/TLS povezave.

Uporabnik ima v svojem gospodinjstvu nameščene senzorje, ki merijo porabo električne energije ter podatke pošiljajo strežniku prek protokola MQTT [15]. Strežnik podatke shranjuje v bazo podatkov. Do podatkov lahko dostopamo prek protokola HTTPS prek vmesnika REST [35] [Tabela 5.1]. Vmesnik je bolj podrobno opisan v [5]. Parametri so opisani v tabeli [Tabela 5.2].

#	METODA	Končna točka (ang. *Endpoint*)
1	GET	/HOUSEHOLD_ID/aggregate/year/YEAR_NUM/day/VALUE
2	GET	/HOUSEHOLD_ID/aggregate/year/YEAR_NUM/week/VALUE
3	GET	/HOUSEHOLD_ID/aggregate/year/YEAR_NUM/month/VALUE

Tabela 5.1: REST vmesnik

	TIP	OPIS
HOUSEHOLD_ID	niz	Enolični identifikator gospodinjstva
YEAR_NUM	število	Leto, iz katerega želimo pridobiti podatke
VALUE	število	Dan/teden/mesec v letu, iz katerega želimo pridobiti podatke

Tabela 5.2: Opis parametrov

Strežnik vrne niz v notaciji JSON, ki vsebuje podatke določenega gospodinjstva. V odgovoru je v primeru, da smo zahtevali dnevne podatke, 24 vrednosti (poraba po urah). Če smo zahtevali tedenske podatke, strežnik vrne 7 vrednosti (poraba po dnevih v tednu), če pa smo zahtevali mesečne podatke, strežnik vrne 28/30/31 vrednosti (poraba po dnevih v mesecu). Poleg vrednosti električne porabe strežnik vrne še nekaj ostalih podatkov, kot so časovna značka, merska enota, interval podatkov, itd. Odgovor strežnika za tedensko porabo izgleda tako:

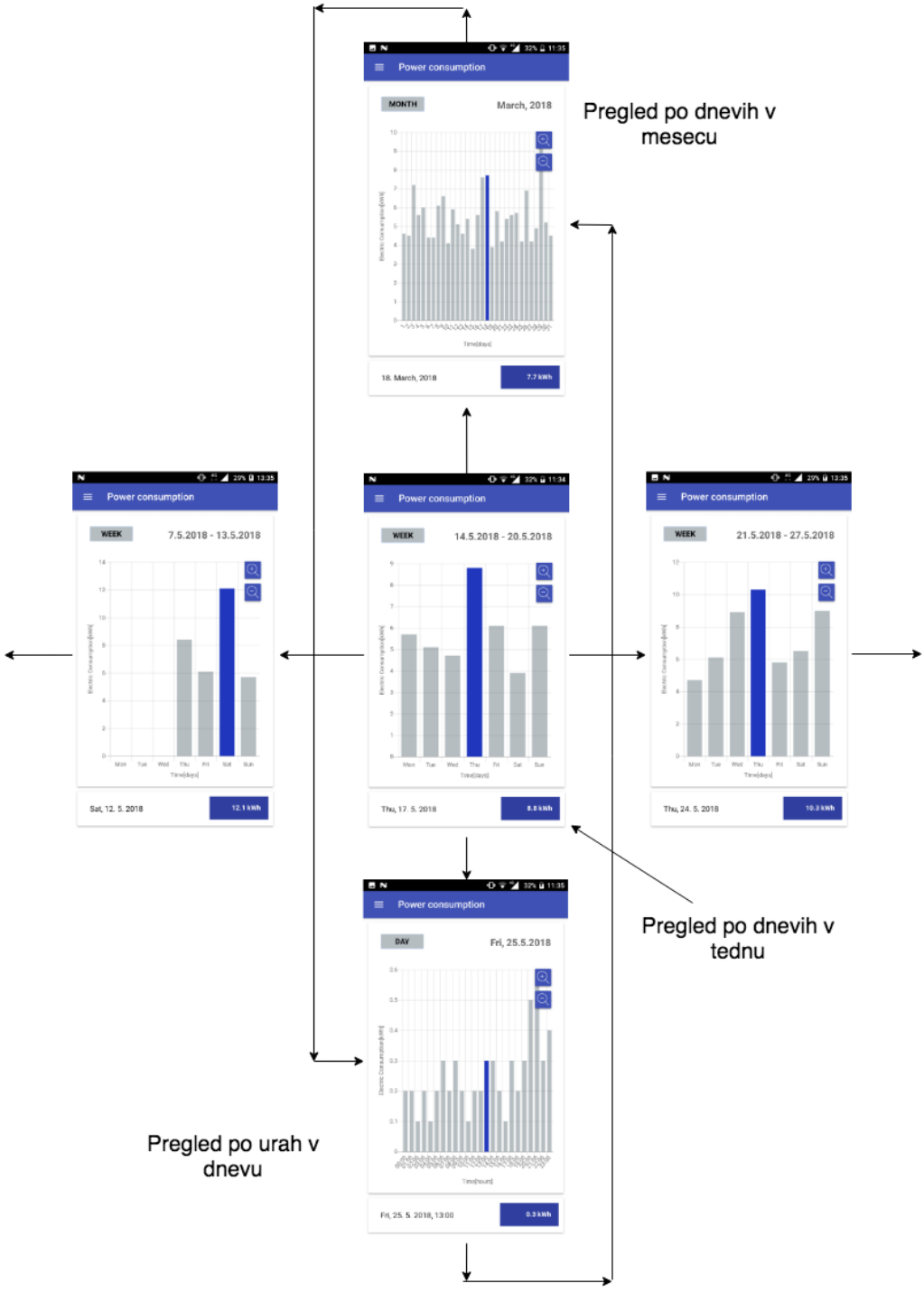
{"name": "energy", "start": "2018-03-04T23:00:00Z",
"interval_unit": "second", "unit": "kWh",
"interval": 86400, "samples": 7,
"data": [6.0, 4.4, 4.4, 6.1, 6.6, 4.1, 5.9]}

Uporabnik aplikacije za dostop do podatkov na zalednem strežniku potrebuje digitalno potrdilo, ki ga pridobi ob postopku registracije na to storitev. Več informacij o celotnem postopku pridobitve digitalnega potrdila je na voljo na [4]. V polju *common name* digitalnega potrdila je vpisan enolični identifikator gospodinjstva – *HOUSEHOLD_ID*. Ob zahtevku, ko odjemalec (aplikacija) ob vzpostavitvi povezave TLS pošlje svoje digitalno potrdilo strežniku, strežnik preveri ali se polje *common name* digitalnega potrdila ujema z *HOUSEHOLD_ID* v URL nizu zahtevka. Če je temu tako, odjemalcu vrne podatke, v nasprotnem primeru pa vrne sporočilo z napako. Tako lahko uporabnik aplikacije dostopa le do podatkov o električni porabi svojega gospodinjstva in do podatkov ostalih uporabnikov nima dostopa.

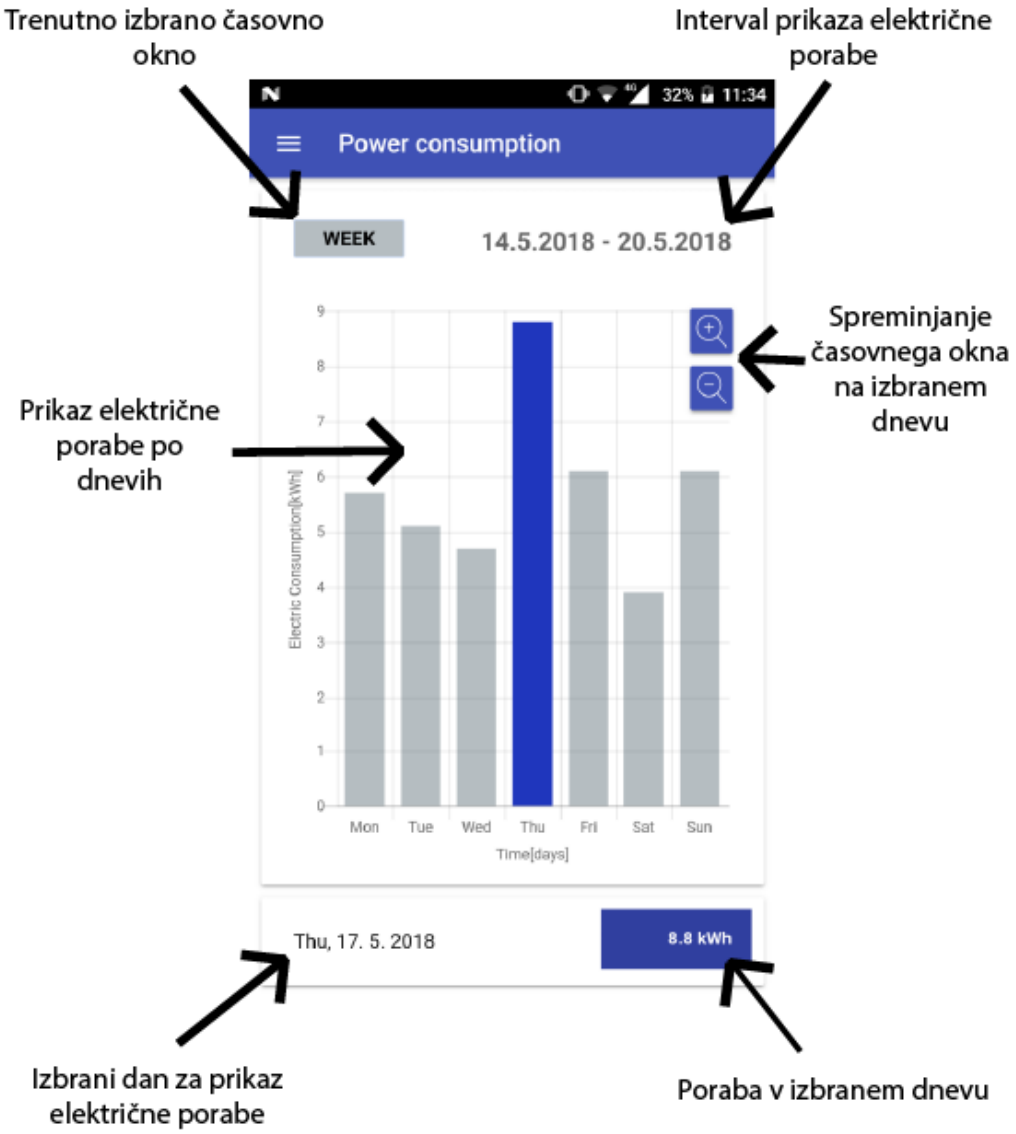
Podatki so v aplikaciji prikazani z grafom. Električno porabo v našem gospodinjstvu lahko pregledujemo v treh časovnih oknih. Prikažemo lahko dnevno porabo po urah, tedensko porabo po dnevih ter mesečno porabo po dnevih. Med različnimi časovnimi okni se premikamo s potegom (ang. *swipe*) po zaslonu gor ali dol. V določenem časovnem oknu pa se po časovnici premikamo s potegom po zaslonu levo ali desno. Kako prehajamo med različnimi časovnimi okni je prikazano na [Slika 5.4], sam uporabniški vmesnik pa je bolj podrobno prikazan na [Slika 5.5].

Podatki se na napravo s strežnika prenašajo sproti. Ko se s potegom premaknemo po časovnici oziroma spremenimo časovno okno se mora izvesti nov zahtevek HTTPS. Vtičnik ob hitri internetni povezavi podatke prenese brez vidne zakasnitve. Podatki, preneseni s strežnika se poleg prikaza še shranijo v notranji pomnilnik mobilne naprave. Tako se podatki za isto časovno obdobje s strežnika ne prenašajo večkrat.

Vtičnik v aplikaciji deluje po pričakovanjih ter dovolj hitro prenaša podatke s strežnika.



Slika 5.4: Uporabniški vmesniki ter prehajanje med pogledi



Slika 5.5: Osrednji uporabniški vmesnik

Poglavje 6

Zaključek

V okviru diplomske naloge smo vtičnik za ogrodje Ionic razširili in nadgradili. Uporabnikom vtičnika smo dodali možnost overjanja uporabnika s pomočjo uporabnikovega digitalnega potrdila. Spoznali smo se z delovanjem in uporabo protokola TLS ter kako poleg overjanja strežnika overimo tudi odjemalca. Delali smo z ogrodjem za izdelavo hibridnih aplikacij Ionic in se naučili, kako funkcionalnost operacijskega sistema prenesti v ogrodje Ionic v okviru vtičnika. Uporabnost in hitrost vtičnika smo ovrednotili z uporabo v testni in pravi aplikaciji.

Z uporabo ogrodij za razvoj hibridnih aplikacij lahko zmanjšamo čas, potreben za izdelavo aplikacije. Problem se pojavi, ko ogrodje ne omogoča določene funkcionalnosti, v našem primeru je to bilo overjanje uporabnika. V okviru diplomskega dela nam je funkcionalnost uspelo dodati za platformo Android. Da bo overjanje uporabnika delovalo tudi za platformo iOS je potrebnega še precej dela. Namreč del vtičnika, ki skrbi za iOS platformo je nekoliko težje razumljiv in bolj razkropljen, veliko izkušenj z programskim jezikom Objective-C pa nimamo.

Kljub temu menimo, da smo odprtokodnemu ogrodju dodali pomemben del funkcionalnosti ter tako marsikateremu razvijalcu olajšali delo. Vtičnik je dostopen na Github platformi na naslovu [27].

Literatura

[1] 12 frameworks for mobile hybrid apps. Dosegljivo: https://blog.jscrambler.com/10-frameworks-for-mobile-hybrid-apps/, [Dostopano: 24. 8. 2018], author=Jscrambler, year=2017.

[2] Android - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Android_(operating_system), [Dostopano: 10. 7. 2018], 2018.

[3] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[4] D2.1 initial security and privacy module. Dosegljivo: https://www.flex4grid.eu/wp-content/uploads/646428-Flex4Grid_-_D2.1_Initial_Security_and_Privacy_Module.pdf, [Dostopano: 28. 8. 2018], 2015.

[5] D2.4 final cloud-based data management module. Dosegljivo: https://www.flex4grid.eu/wp-content/uploads/646428-Flex4Grid-D2.4_Final_Cloud-based_Data_Management_module.pdf, [Dostopano: 28. 8. 2018], 2015.

[6] Domača stran android studio. Dosegljivo: https://developer.android.com/studio/, [Dostopano: 10. 7. 2018].

[7] Domača stran angularjs. Dosegljivo: https://angularjs.org/, [Dostopano: 16. 7. 2018].

[8] Domača stran apple. Dosegljivo: https://www.apple.com/si/, [Dostopano: 24. 6. 2018].

[9] Domača stran cordova. Dosegljivo: https://cordova.apache.org/, [Dostopano: 15. 8. 2018].

[10] Domača stran eclipse. Dosegljivo: https://www.eclipse.org/, [Dostopano: 10. 7. 2018].

[11] Domača stran flask. Dosegljivo: http://flask.pocoo.org/, [Dostopano: 5. 8. 2018].

[12] Domača stran framework7. Dosegljivo: https://framework7.io/, [Dostopano: 24. 8. 2018].

[13] Domača stran ionic. Dosegljivo: https://ionicframework.com/, [Dostopano: 3. 7. 2018].

[14] Domača stran java. Dosegljivo: https://www.java.com/en/, [Dostopano: 12. 7. 2018].

[15] Domača stran mqtt. Dosegljivo: http://mqtt.org/, [Dostopano: 25. 8. 2018].

[16] Domača stran openssl. Dosegljivo: https://www.openssl.org/, [Dostopano: 19. 6. 2018].

[17] Domača stran phonegap. Dosegljivo: https://phonegap.com/, [Dostopano: 24. 8. 2018].

[18] Domača stran python. Dosegljivo: https://www.python.org/, [Dostopano: 19. 8. 2018].

[19] Domača stran python knjižnice requests. Dosegljivo: http://docs.python-requests.org/en/master/, [Dostopano: 5. 6. 2018].

[20] Domača stran react native. Dosegljivo: https://facebook.github.io/react-native/, [Dostopano: 24. 8. 2018].

[21] Domača stran react.js. Dosegljivo: https://reactjs.org/, [Dostopano: 25. 8. 2018].

[22] Domača stran xamarin. Dosegljivo: https://visualstudio.microsoft.com/xamarin/, [Dostopano: 24. 8. 2018].

[23] Domača stran xcode. Dosegljivo: https://developer.apple.com/xcode/, [Dostopano: 15. 8. 2018].

[24] Dot net programski jezik. Dosegljivo: https://www.microsoft.com/net, [Dostopano: 25. 8. 2018].

[25] Rupa Ganjewar. Diffie hellman key exchange. 2010.

[26] Github vtičnika cordova-plugin-advanced-http. Dosegljivo: https://github.com/silkimen/cordova-plugin-advanced-http, [Dostopano: 30. 6. 2018].

[27] Github vtičnika cordova-plugin-advanced-http-with-client-cert. Dosegljivo: https://github.com/klemenStanic/cordova-plugin-advanced-http-with-client-cert, [Dostopano: 25. 6. 2018].

[28] Jay Graves. Ssl pinning for increased app security. Dosegljivo: https://possiblemobile.com/2013/03/ssl-pinning-for-increased-app-security/, [Dostopano: 5. 6. 2018], 2013.

[29] Russell Housley, Warwick Ford, William Polk, and David Solo. Internet x. 509 public key infrastructure certificate and crl profile. Technical report, 1998.

[30] Ionic http vtičnik - dokumentacija. Dosegljivo: https://ionicframework.com/docs/native/http/, [Dostopano: 3. 8. 2018], 2018.

[31] ios - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/IOS, [Dostopano: 15. 8. 2018], 2018.

[32] Kotlin programski jezik. Dosegljivo: https://kotlinlang.org/, [Dostopano: 12. 7. 2018].

[33] Linux kernel. Dosegljivo: https://www.kernel.org/, [Dostopano: 30. 7. 2018].

[34] Man in the middle (mitm) attack. Dosegljivo: https://www.incapsula.com/web-application-security/man-in-the-middle-mitm.html, [Dostopano: 5. 6. 2018], 2018.

[35] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", 2011.

[36] Message authentication code - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Message_authentication_code, [Dostopano: 12. 7. 2018], 2018.

[37] Npm cordova-plugin-advanced-http. Dosegljivo: https://www.npmjs.com/package/cordova-plugin-advanced-http, [Dostopano: 29. 6. 2018].

[38] Number of apps available in leading app stores as of 1st quarter 2018. Dosegljivo: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, [Dostopano: 10. 8. 2018], 2018.

[39] Objective c - wikipedia. Dosegljivo: https://en.wikipedia.org/wiki/Objective-C, [Dostopano: 1. 8. 2018], 2018.

[40] DongGook Park, Colin Boyd, and Sang-Jae Moon. Forward secrecy and its application to future mobile communications security. In *International Workshop on Public Key Cryptography*, pages 433–445. Springer, 2000.

[41] Projekt flex4grid. Dosegljivo: https://www.flex4grid.eu/, [Dostopano: 5. 8. 2018].

[42] Eric Rescola. *SSL and TLS, Designing and Building Secure Systems.* Addison-Wesley, 2000.

[43] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018.

[44] Arto Salomaa. *Public-key cryptography.* Springer Science & Business Media, 2013.

[45] Self-signed certificate. Dosegljivo: `https://en.wikipedia.org/wiki/Self-signed_certificate`, [Dostopano: 13. 6. 2018], 2018.

[46] Swift programski jezik. Dosegljivo: `https://developer.apple.com/swift/`, [Dostopano: 12. 7. 2018].

[47] Symmetric-key algorithm - wikipedia. Dosegljivo: `https://en.wikipedia.org/wiki/Symmetric-key_algorithm`, [Dostopano: 12. 7. 2018], 2018.

[48] Transport layer security - wikipedia. Dosegljivo: `https://en.wikipedia.org/wiki/Transport_Layer_Security`, [Dostopano: 12. 7. 2018], 2018.