



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Стефан Боснић

Реализација мултиплатформских апликација помоћу алата Кордова

МАСТЕР РАД

Нови Сад, 2016



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР :	
Идентификациони број, ИБР :	
Тип документације, ТД :	Монографска документација
Тип записа, ТЗ :	Текстуални штампани материјал
Врста рада, ВР :	Дипломски – мастер рад
Аутор, АУ :	Стефан Боснић
Ментор, МН :	Др. Иштван Пап
Наслов рада, НР :	Реализација мултиплатформских апликације помоћу алата Кордова
Језик публикације, ЈП :	Српски / латиница
Језик извода, ЈИ :	Српски
Земља публикавања, ЗП :	Република Србија
Уже географско подручје, УГП :	Војводина
Година, ГО :	2016
Издавач, ИЗ :	Ауторски репринт
Место и адреса, МА :	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, ФО : (поглавља/страна/ цитата/табела/слика/графика/прилога)	7/56/2/7/39/0/0
Научна област, НО :	Електротехника и рачунарство
Научна дисциплина, НД :	Рачунарска техника
Предметна одредница/Кључне речи, ПО :	Кордова, хибридна апликација, паметна кућа
УДК	
Чува се, ЧУ :	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, ВН :	
Извод, ИЗ :	U ovom radu opisano je rešenje hibridne aplikacije, za upravljanje jednog sistema pametne kuće, sa ciljem doprinosa i napretka u razvoju mobilnih aplikacija. Primarni cilj je evaluacija alata koji omogućava generisanje aplikacija iz jednog koda za više mobilnih platformi. Svrha evaluacije je rešavanje problema razvoja aplikacija za različite platforme. Rešenje je realizovano za dve trenutno najdominantnije platforme: Android i iOS.
Датум прихватања теме, ДП :	
Датум одбране, ДО :	
Чланови комисије, КО :	Председник: Др. Растислав Стухарик, доцент
	Члан: Др Небојша Пјевлаица, доцент
	Члан, ментор: Др. Иштван Пап, доцент
	Потпис ментора



UNIVERSITY OF NOVI SAD • FACULTY OF TECHNICAL SCIENCES
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, ANO :			
Identification number, INO :			
Document type, DT :	Monographic publication		
Type of record, TR :	Textual printed material		
Contents code, CC :	Master Thesis		
Author, AU :	Stefan Bosnic		
Mentor, MN :	PhD Istvan Pap		
Title, TI :	Implementation of Multiplatform applications using tools Cordova		
Language of text, LT :	Serbian		
Language of abstract, LA :	Serbian		
Country of publication, CP :	Republic of Serbia		
Locality of publication, LP :	Vojvodina		
Publication year, PY :	2016		
Publisher, PB :	Author's reprint		
Publication place, PP :	Novi Sad, Dositeja Obradovica sq. 6		
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/56/2/7/39/0/0		
Scientific field, SF :	Electrical Engineering		
Scientific discipline, SD :	Computer Engineering, Engineering of Computer Based Systems		
Subject/Key words, S/KW :	Cordova, hybrid applications, smart home		
UC			
Holding data, HD :	The Library of Faculty of Technical Sciences, Novi Sad, Serbia		
Note, N :			
Abstract, AB :	This paper describes one solution of hybrid application that control system of a smart home. Primary goal is evaluation of tools that allow generating applications from one source code for multiple mobile platforms, regarding solving problem of application development for various platforms. The solution is implemented for the two currently most dominant platforms: Android and iOS.		
Accepted by the Scientific Board on, ASB :			
Defended on, DE :			
Defended Board, DB :	President:	PhD Rastislav Stuharik, docent	
	Member:	PhD Nebojsa Pjevalica, docent	Menthor's sign
	Member, Mentor:	PhD Istvan Pap, docent	

Zahvalnost

Zahvalio bih se rodbini i prijateljima koji su mi pružali podršku i koji su bili tu da isprate moje ideje i vizije. Izdvojio bih prijatelja Marka Dikića kome dugujem ogromnu zahvalnost.

Takođe veliku zahvalnost dugujem profesorima, kolegama i mentoru zbog velike pomoći, razumevanja, pružanja svih uslova za sticanje novog znanja i za konstantno usavršavanje.

Pruženu podršku od mojih bližnjih nikada neću zaboraviti, pre svega majke Lepojke, brata Aleksandra i strica Vladimira. Dugujem im najveću zahvalnost i hvala im zbog pružanja bezrezervne emotivne i moralne podrške u svakom trenutku, kao i finansijske podrške tokom školovanja.

Na kraju bih se zahvalio svima koji su na bilo koji način doprineli izradi master rada.

SADRŽAJ

1. Uvod.....	8
1.1 Internet of Things	8
1.2 Home automation	10
1.2.1 Sistem pametne kuće	10
1.2.1.1 Centralni uređaj	11
1.2.1.2 ZigBee, ZWave i IP uređaji.....	12
1.2.2 Mobilne tehnologije - jedno rešenje za kontrolu upravljanja sistemom pametne kuće	13
2. Teorijske osnove	15
2.1 Trenutni pristup razvoju mobilnih aplikacija	15
2.1.1 Nativne (<i>native</i>) mobilne aplikacije.....	16
2.1.2 Internet mobilne aplikacije	16
2.1.3 Hibridne mobilne aplikacije	16
2.2 <i>Apache Cordova</i>	17
2.2.1 <i>AngularJS framework</i>	20
2.2.1.1 <i>MVC</i> šablon	21
2.2.1.2 Moduli	22
2.2.1.3 <i>Service</i> (Servisi)	22
2.2.1.4 <i>Scope</i>	23
2.2.1.5 View	23
2.2.1.6 Dependency Injection.....	23
2.2.2 Ionic	24
2.2.2.1 Klase platforme	24

2.2.2.2	Kontrola stila aplikacije upotrebom <i>AngularJS</i>	24
2.2.3	Webview	26
2.2.3.1	iOS	26
2.2.3.2	Android	27
2.2.4	<i>Plugin</i>	28
2.2.4.1	Core plugin	29
2.2.4.2	Custom plugin	30
2.2.4.3	plugin.xml	32
3.	Opis rešenja	34
3.1	Pozadinski mehanizam	35
3.1.1	Komunikacioni pod modul	36
3.1.1.1	Realizacija modula za komunikaciju	36
3.1.2	Pod modul za pronalazak uređaja u mreži	43
3.1.2.1	Realizacija modula za pronalazak uređaja u mreži	43
3.2	Grafička korisnička sprega	48
3.2.1	Pod modul za kontrolu uređaja	49
3.2.1.1	Realizacija pod modula za kontrolu uređaja	49
4.	Testiranje i rezultati	52
4.1	Testiranje funkcionalnosti	52
4.2	Testiranje prilikom opterećenja	55
4.2.1	Testiranje iskorišćenosti procesora	55
4.2.2	Testiranje zauzetosti radne memorije	57
4.3	Testiranje u realnim uslovima	59
4.4	Testiranje potrebnog napora prilikom razvoja	60
5.	Zaključak	61
6.	Literatura	63

SPISAK SLIKA

Slika 1.1: Arhitektura <i>Cloud-a</i>	9
Slika 1.2: Arhitektura sistema pametne kuće	11
Slika 1.3: Komunikacija između klijenta i centralnog uređaja	12
Slika 1.4: Uređaji u sistemu pametne kuće	13
Slika 2.1: Arhitektura nativne, internet i hibridne aplikacije	17
Slika 2.2: Proces pakovanja internet aplikacije u nativni kontejner	18
Slika 2.3: Arhitektura <i>Cordova</i> aplikacije	20
Slika 2.4: Bitni delovi <i>AngularJS framework-a</i>	21
Slika 2.5: <i>MVC</i> šablon u <i>AngularJS framework-u</i>	22
Slika 2.6: Redefinisanje izgleda komponente upotrebom <i>Ionic</i> klasa	24
Slika 2.7: Dinamičko generisanje izgleda elemenata	25
Slika 2.8: Generisanje jedne stranice za određeno stanje	25
Slika 2.9: <i>iOs Webview</i>	27
Slika 2.10: <i>Android Webview</i>	28
Slika 2.11: Arhitektura <i>Cordova plugin-a</i>	29
Slika 2.12: Metoda <i>cordova.exec</i>	31
Slika 2.13: Metoda <i>execute</i>	31
Slika 2.14: <i>Action</i> metoda	32
Slika 2.15: <i>plugin.xml</i> datoteka	33
Slika 3.1: Arhitektura celokupno sistema pametne kuće	34
Slika 3.2: Arhitektura klijentske aplikacije	35
Slika 3.3: Nivoi protokola komunikacije	36
Slika 3.4: Uspostavljanje veze sa centralnim uređajem	37

Slika 3.5: <i>SSL</i> kanal za prenos podataka	38
Slika 3.6: Struktura modula za komunikaciju	39
Slika 3.7: Upotreba <i>then</i> metode	41
Slika 3.8: Primer implementacije komande	42
Slika 3.9: Razmena poruka putem <i>SSDP</i> protokola.....	43
Slika 3.10: <i>XML</i> datoteka	44
Slika 3.11: <i>Cordova bridge</i>	45
Slika 3.12: Poziv <i>plugin</i> metode	45
Slika 3.13: Podaci za prikaz uređaja	50
Slika 3.14: Izgled prikaza liste dostupnih uređaja na centralnom uređaju i kontrole odabranog dostupnog uređaja.....	51
Slika 4.1: Grafički prikaz iskorištenosti procesora pri radu hibridne aplikacije.....	56
Slika 4.2: Grafički prikaz iskorištenosti procesora pri radu native aplikacije	56
Slika 4.3: Poređenje ostvarenih rezultata hibridne i native aplikacije za testiranje opterećenja procesora	57
Slika 4.4: Grafički prikaz zauzetosti radne memorije pri radu hibridne aplikacije	57
Slika 4.5: Grafički prikaz zauzetosti radne memorije pri radu native aplikacije.....	58
Slika 4.6: Poređenje ostvarenih rezultata hibridne i native aplikacije za testiranje zauzetosti radne memorije.....	58

SPISAK TABELA

Tabela 3.1: Ugrađeni servisi <i>AngularJS framework-a</i>	40
Tabela 3.2: Metode <i>JavaScript</i> interfejsa	46
Tabela 3.3 Metode <i>Native</i> interfejsa	47
Tabela 3.4: Metoda iz <i>Java</i> klase	48
Tabela 4.1: Manuelni testovi funkcionalnosti	55
Tabela 4.2: Prikaz testova odziva sistema	59
Tabela 4.3: Prikaz potrebnog vremena za razvoj aplikacije	60

SKRAĆENICE

AJAX	- Asynchronous JavaScript And Xml
API	- Application Programming Interface
CSS	- Cascading Style Sheets
CPU	- Central Processor Unit
DOM	- Document Object Model
FFI	- Foreign Function Interface
HTML	- Hyper Text Markup Language
HTTP	- Hyper Text Transfer Protocol
JSON	- JavaScript Object Notation
MVC	- Model View Controller
SDK	- Software Development Kit
SSDP	- Simple Service Discovery Protocol
SSL	- Secure Socket Layer
TCP	- Transmission Control Protocol
UDP	- User Data Packet
UI	- User Interface
URL	- Uniform Resource Locator
UX	- User Experience
XML	- Extensible Markup Language

1. Uvod

Svakim danom okruženje oko nas se usavršava i napreduje. Samim tim i naš kvalitet života se menja i usavršava. Veliki uticaj na kvalitet života ima ubrzan razvoj mnogih tehnologija i njihovo prisustvo postepeno se primećuje. Ali nedovoljno u odnosu na njihovu upotrebu u svakodnevnicu.

Napredak u razvoju tehnologija prouzrokovan je mnogim uspešnim istraživanjima, otkrićima i pre svega realizacijama velikih ideja. Razvoj nikada ne stagnira, nego konstantno napreduje i sve češće se govori o „sledećoj velikoj stvari“, koja će uticati na kvalitet života. Mnogi ne shvataju pojam „sledeće velike stvari“. Ne radi se o nedostatku mašte, nego o nedostatku posmatranja. Nije potrebno zamišljati, nego je potrebno primetiti da je budućnost već u našem vidokrugu [1]. *Internet of Things (IoT)* [2] je jedan od najvećih trenodova tehnologija koji predstavlja budućnost, a već čini naše okruženje.

1.1 Internet of Things

Internet of Things (IoT) je sistem povezanih računarskih uređaja, mehaničkih i digitalnih mašina, objekata, životinja ili ljudi koji sadrže jedinstvenu identifikaciju i sposobnost prenosa podataka preko mreže bez interakcije *human-to-human* ili *human-to-computer*.

IoT je evoluirao iz konvergencije bežičnih tehnologija, mikro-elektromehaničkih sistema (*MEMS*), mikroservisa i interneta. Prethodno pomenuta konvergencija omogućila je otklanjanje prepreke između operativne tehnologije (*OT*) i informacione tehnologije (*IT*).

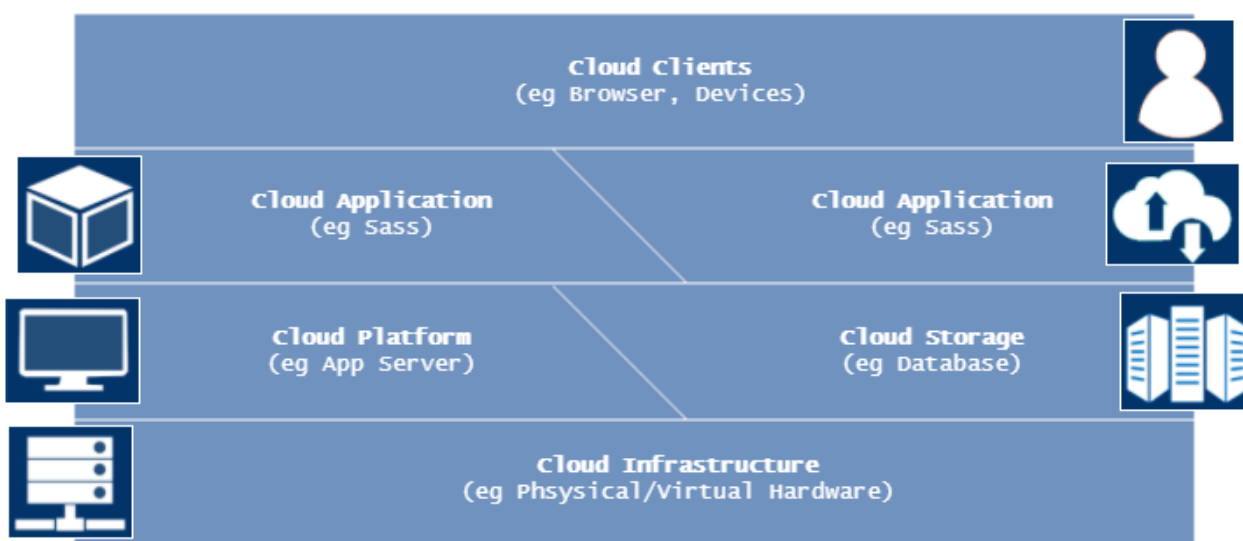
Kevin Ešton (*Kevin Ashton*) prvi put pominje *IoT* u prezentaciji 1999. godine. Objašnjavajući potencijal *IoT*: „Danas računari, samim tim i internet, za informacije gotovo u potpunosti zavise od ljudskih bića. Svi podaci, otprilike oko 50 petabajta (1 petabajt = 1024

terabajta) dostupnih na internetu su prvo stvorila ljudska bića kucanjem, upotrebom dugmeta za snimanje, uzimajući digitalnu sliku ili skeniranjem bar koda.

Ljudi imaju ograničeno vreme, pažnju i preciznost, što predstavlja problem. Odnosno nisu veoma dobri u prikupljanju podataka o stvarnom svetu. Ako bismo imali kompjutere koji znaju sve što se može znati o stvarima, koristeći podatke prikupljene bez bilo kakve naše pomoći. Mi bismo mogli pratiti i brojati sve stvari i veoma smanjiti gubitke i troškove. Znali bismo kada je stvari potrebno zameniti, popraviti ili proveriti da li sadrže najnovije informacije.“ [3].

IoT se trudi da poboljša *machine-to-machine* (*M2M*) komunikaciju. Jedno rešenje predstavljaju *Cloud* serveri. *Cloud* (udaljeni server) omogućava aplikacijama da rade za nas bilo kada i bilo gde. Komunikaciju predstavljaju *cloud* serveri i mreže za prikupljanje podataka senzora, omogućavajući uspostavljanje mobilne, virtuelne i trenutne veze koja će većinu stvari u našim životima učiniti „pametnim“ (*smart*).

Zadatak senzora je da meri, ocenjuje i prikuplja podatke. *IoT* povezuje senzore i mašine, odnosno povezuje prikupljanje informacije i mašine. Sve prikupljene informacije od senzora ne vrede mnogo ako ne postoji infrastruktura za njihovu analizu u realnom vremenu. *Cloud* aplikacije su ključ za upotrebu podataka. Bez upotrebe *cloud* aplikacije za tumačenje i prenos podataka koji dolaze od senzora, *IoT* ne funkcioniše. Arhitektura *Cloud-a* prikazana je na slici 1.1:



Slika 1.1: Arhitektura *Cloud-a*

Takođe imamo *cloud* aplikacije koje podatke prenose mašinama na terenu, unapređujući mašinama inteligenciju i omogućavajući njihove odgovore (reakcije) u realnom vremenu. Tako

stvari u našem okruženju postaju „pametne“. Što je ogromna i fundamentalna promena, jer izradom pametnih stvari povećava se potreba za izradom novih proizvoda i novih usluga. Već postoje mnogi pametni proizvodi, neki od njih su: pametni mobilni uređaji, pametni automobili, pametni mostovi, pametne kuće itd. U daljem tekstu usmerićemo pažnju ka pojmu pametnih kuća ili sistemu kućne automatizacije.

1.2 Home automation

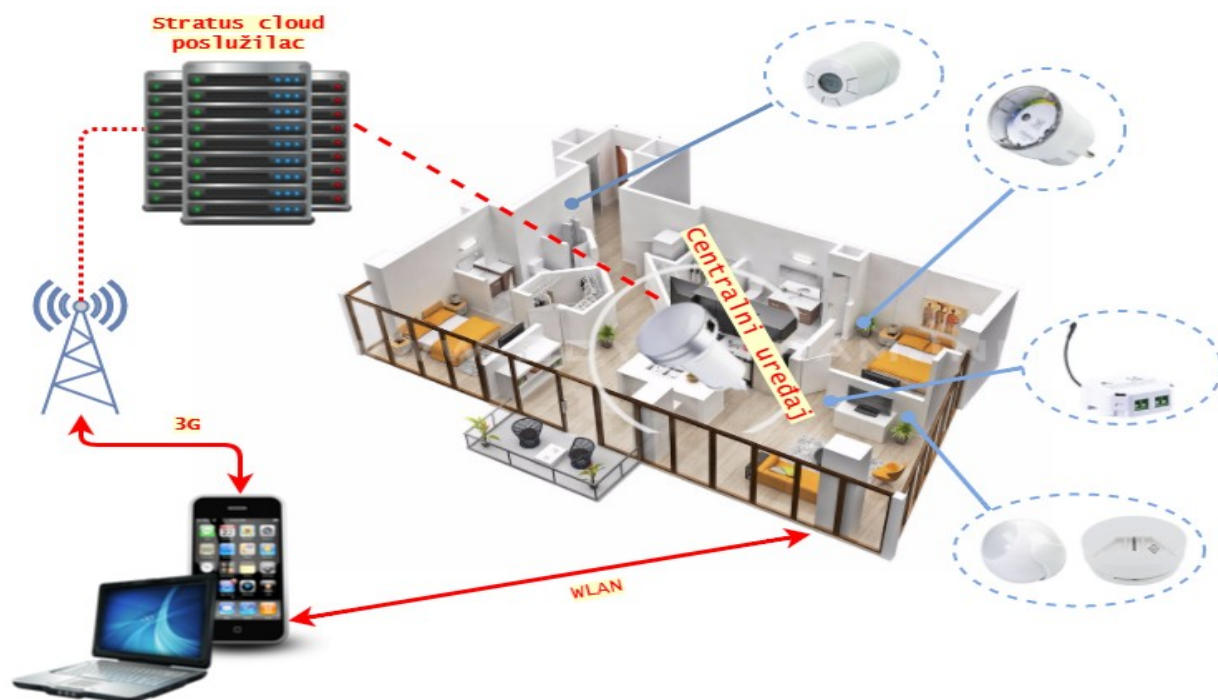
Uvođenjem *IoT* tehnologije u kuće (domaćinstva), omogućeno im je dobijanje inteligencije, čime postaju inteligentne ili pametne kuće. U ovom slučaju novi proizvod je sistem kućne automatizacije, a usluge su kontrola uređaja na željeni način.

Pojam pametne kuće prvi put se pojavljuje u sedamdesetim godinama prošlog veka. U tom periodu, kuća sagrađena sa ciljem iskorištavanja energetske efikasnosti smatrala se pametnom kućom. U današnjici, pametna kuća je objekat u kome kontrolu električnih uređaja vrši računarski sistem, sa ciljem povećanja komfora i energetske efikasnosti objekta.

Sistem kućne automatizacije ili sistem pametnih kuća je sistem koji podrazumeva integraciju raznih uređaja u domaćinstvu. Odnosno sistem kojim se može upravljati na različite načine, u zavisnosti od navika i potreba korisnika. Kontrola sistema se može ostvariti pomoću raznih uređaja: univerzalnog daljinskog upravljača, računara ili uređaja sa ekranom osjetljivim na dodir. Sistem pametne kuće se integriše na već postojeće instalacije bez građevinskih radova.

1.2.1 Sistem pametne kuće

Integracija pametnih kuća u domaćinstvu zasniva se na principu realizacije pametne kuće koja pomoću bežične i lokalne mreže omogućava kontrolu uređaja priključenih na električnu mrežu. Kontrola uređaja se može izvršavati u dva režima: udaljeni (*remote*) režim i lokalni režim. U *remote* režimu komunikacija se odvija preko udaljenog *cloud* servera putem *AJAX* zahteva. Dok se u lokalnom režimu komunikacija odvija u lokalnoj mreži putem *TCP* komunikacije. Komunikacija se ostvaruje sa centralnim uređajem koji kontroliše ostale uređaje na osnovu primljenih komandi kroz lokalnu ili bežičnu mrežu. Arhitektura sistema pametne kuće je prikazana na slici:



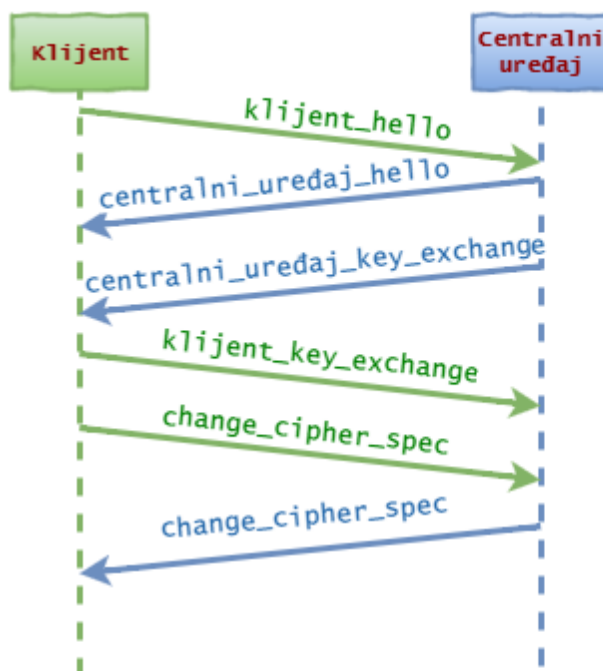
Slika 1.2: Arhitektura sistema pametne kuće

Moguće je upravljati svetlosnim i drugim grupama nezavisno od električnih instalacija. Sistem pruža mogućnost integracije sa drugim uređajima ili sistemima u domaćinskim objektima pružajući fleksibilnost u njihovoj upotrebi.

1.2.1.1 Centralni uređaj

Centralni uređaj obrađuje naredbe dobijene od strane klijenta i dobavlja podatke o stanju uređaja u sistemu. Unutar centralnog uređaja realizovan je rukovalac događajima koji za svaku promenu stanja u okruženju aktivira novi događaj. Programski jezik *C++* predstavlja realizaciju programske podrške. Na strani centralnog uređaja realizovan je *TCP* konektor koji rukuje sa klijentima i preko njega klijenti komuniciraju sa centralnim uređajem.

Komunikacija između centralnog uređaja i klijenta realizuje se u vidu poruka koje su formatirane kao *JSON* objekti. Upravljanje sistemom se odvija preko enkriptovane komunikacije, u ovom slučaju takva komunikacija se naziva *Encrypted handshake*. Osnovni koncept komunikacije prikazan je na slici 1.3:



Slika 1.3: Komunikacija između klijenta i centralnog uređaja

Slanje zahteva za povezivanje je proces gde se od glavnog uređaja nasumično dobija generisani niz znakova. Niz se zatim šifruje koristeći odgovarajući ključ i tako šifrovana poruka šalje se nazad centralnom uređaju. Dešifrovanje dobijene poruke je proces poređenja poslate i dobijene poruke. Navedeni proces se realizuje na strani centralnog uređaja gdje se nakon proces poređenja odlučuje da li je korisnik uspešno autorizovan ili ne.

Klijentima koji izvrše autorizaciju omogućeno je slanje komandi, primanje odgovora i informacija o događajima. Prikazan je slučaj kada je klijentska aplikacija direktno povezana na centralni uređaj.

1.2.1.2 ZigBee, ZWave i IP uređaji

Za efikasno upravljanje i konfiguraciju uređaja sistem se oslanja na dva protokola: *ZigBee*[4] i *Z-Wave*[5]. Određeni uređaji se mogu povezati na već postojeće električne instalacije, ali većina uređaja su uglavnom baterijski napajana. Nekoliko grupa uređaja su podržani u sistemu pametne kuće. Podržani uređaji su: pametne utičnice, pametni prekidači, daljinski upravljači i razne vrste senzora (senzor pokreta, senzor poplave, senzor vlage itd). Na slici su prikazani podržani uređaji:



Slika 1.4: Uređaji u sistemu pametne kuće

Sistem podržava razne *IP* uređaje. Jedan od *IP* uređaja je *VOCO* [6]. *VOCO* je multimedijalni uređaj koji može reprodukovati muziku i predefinsane zvukove, tj. razne vrste alarma. Upotreba *VOCO* uređaja u sistemu pametne kuće je puštanje raznih alarm zvukova u slučaju neželjenih događaja. Odnosno *VOCO* uređaj se konfiguriše da pušta alarm zvuke kada se desi neželjena pojava u kuću, kao npr. poplava.

1.2.2 Mobilne tehnologije - jedno rešenje za kontrolu upravljanja sistemom pametne kuće

Osnovne mogućnosti mobilnih uređaja su pružanje usluga poziva i slanja tekst poruka. Vremenom mobilni uređaji se usavršavaju, postajući uređaji sa integrisanim računarom i dodatnim mogućnostima koje nisu prvobitno povezane sa telefonom, kao što su: operativni sistem, internet pretraživanje i mogućnosti za pokretanje softverske aplikacije.

Mobilni uređaji postaju „pametni“ mobilni uređaji („*smartphone*“). Neprimetno su se pojavili i postali neophodni za današnjicu i kvalitet života. Velikim i ubrzanim razvojem pametnih mobilnih uređaja koji se sastoje od integrisanog operativnog sistema, ekrana osjetljivog na dodir, internet pretraživača i ostalih mnogobrojnih mogućnosti, pojavljuje se ideja i potreba za uvođenjem pametnih mobilnih telefona u sistem pametnih kuća. Pored tehnološke aktuelnosti, koja predstavlja osnovnu prednost kontrole pametnih kuća pomoću pametnih uređaja, najznačajnija prednost je omogućavanje upravljanja pametnom kućom sa bilo kog mjesta i na

način koji korisniku odgovara. Takvo upravljanje omogućava *IoT*, preko *cloud* aplikacija. *Cloud* aplikacije unapredile su razvoj mobilnih aplikacija. Mobilne aplikacije predstavljati aplikativnu podršku za mobilne platforme, omogućavajući korisnicima lakše upravljanje.

Za očekivanje je da pametne kuće budu rasprostranjene širom zemljine kugle. Samim tim potrebno je napraviti aplikativnu podršku za sve dominantne mobilne platforme. Naravno da jedan od problema predstavlja i *UI* zbog različitosti svakog operativnog sistema. Takođe, *UX*, visoke performanse i minimalna upotreba resursa uređaja predstavljaju izazov tokom kreiranja aplikativne podrške. Aplikativna podrška kreirana za jednu mobilnu platformu nije primenjiva na druge mobilne platforme. Odnosno, aplikaciju razvijenu za jednu platformu nije moguće koristiti na drugoj platformi, a da se aplikacija ne razvija od samog početka. Proces razvoja aplikacije za korisnike različitih mobilnih platformi je mnogo dug i „skup“, jer je potrebno iz različitih kodova generisati aplikaciju. Za primjer uzimam dve najdominantnije platforme mobilnih uređaja: *Android* i *iOS*. Aplikacija za *Android* biće opisana u *Java* programskom jeziku, a ista aplikacija za *iOS* u objektnom C (*Objective C*) programskom jeziku. Potrebno je održavanje više verzija jedne aplikacije. Što predstavlja glavni problem, ali i motiv za pronalaženje načina za generisanje aplikacije iz jednog koda, za različite mobilne platforme. Čime bi se neutralizovao dug i „skup“ proces razvoja mobilne aplikacije.

Rešenje problema nudi *Apache Cordova*. *Apache Cordova*[7] je *framework* za razvoj mobilnih aplikacija. Omogućava generisanje aplikacija iz jednog koda za sve mobilne platforme. Naredna poglavlja detaljno opisuju *Apache Cordova framework*. Počevši od arhitekture i strukture, pa do načina upotrebe i implementacije *Cordova* aplikacije.

2. Teorijske osnove

Upoznavanje sa *Apache Cordova framework-om* i načinom na koji rešava predstavljeni problem su osnovni cilj ovog poglavlja. Trenutni pristup razvoju mobilnih aplikacija predstavlja uvod ka razumevanju potrebe za generisanjem aplikacija iz jednog koda za različite mobilne platforme.

2.1 Trenutni pristup razvoju mobilnih aplikacija

Aplikacijski softveri koji su razvijeni da rade na pametnim mobilnim uređajima i tabletima, nazivaju se mobilne aplikacije. Većina mobilnih aplikacije su već instalirane na uređajima i na taj način su dostupne korisniku. Takođe se mogu preuzeti *online* preko *online* prodavnica po imenu *app store* (*App store*, *Google play*, *Windows phone store* itd). Prvobitna njihova upotreba je bila neprimjetna, ali velika potražnja dovela je do njihove ekspanzije. Danas se koriste u raznim oblastima i nezamenjive su. Tržište mobilnih aplikacija nalazi se u stalnom porastu i danas je ono veoma razvijeno.

Razvijanje mobilnih aplikacija zahteva proširivanje ograničenja i iskorištavanje karakteristika mobilnih uređaja. Mobilni uređaji imaju slabije procesore i nemaju napajanje kao personalni računari, a pri tome imaju više funkcionalnosti.

Takođe, razvoj mobilnih aplikacija zahteva upotrebu specijalizovanih razvojnih okruženja. Mobilne aplikacije su prvo testirane u razvojnom okruženju koristeći emulator, a kasnije su podvrgnute fizičkom testiranju. Emulatori pružaju jeftin način testiranja aplikacije bez zahtevanja fizičkog pristupa. Od suštinskog značaja za mobilne aplikacije je korisnički interfejs (*UI*). *UI* predstavlja prostor gde se ostvaruje interakcija između čoveka i mašine. Primarni cilj korisničkog interfejsa je lakša, efikasnija i prijatnija interakcija sa mašinom u cilju postizanja

traženih rezultata. Razumljiv, lak i prepoznatljiv interfejs je želja svakog korisnika, zbog česte interakcije sa svojim uređajem. Aplikacije sa loše dizajniranim korisničkim interfejsom imaju malu vrednost. Mobilne aplikacije se dele na: nativne (*native*), internet i hibridne mobilne aplikacije.

2.1.1 Nativne (*native*) mobilne aplikacije

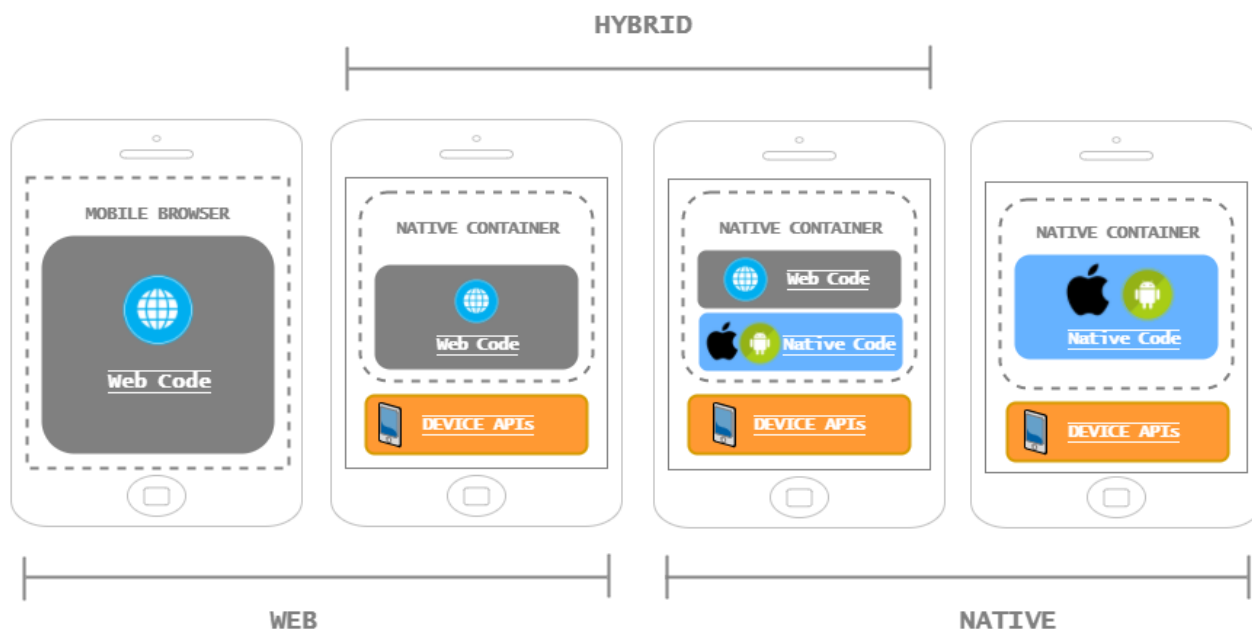
Nativne mobilne aplikacije su aplikacije razvijene pomoću *SDK (Software development kit)* i programskog jezika određene mobilne platforme. Njihova prednost se ogleda u tome što su kreirane po meri za podržanu platformu i ostvaruju najbolje rezultate koje je moguće ostvariti na toj platformi. Osnovno ograničenje ovakvih mobilnih aplikacija je nemogućnost prebacivanja aplikacije na drugu platformu, a da se kompletan postojeći programski kod aplikacije ne piše od samog početka. Nedostatak se sam nagoveštava, veliki trošak za razvoj i održavanje nekoliko različitih verzija aplikacije.

2.1.2 Internet mobilne aplikacije

Internet mobilne aplikacije su nastale kao moguće rešenje nedostatka nativnih aplikacija. Nedostatak je potreba za smanjenjem troškova i transparentnost u odnosu na platformu mobilnog uređaja. Internet aplikacija lako pokriva više platformi za manji trošak. Na taj način se brže širi na tržištu. Ali one nemaju pristup svim funkcionalnostima uređaja. Jedan od nedostataka je pojavljivanje specifičnih problema koji su uzrok sporijeg i lošijeg funkcionisanja aplikacije na određenim platformama.

2.1.3 Hibridne mobilne aplikacije

Hibridne mobilne aplikacije predstavljaju treću vrstu mobilnih aplikacija. Smatraju se za rešenje nedostataka internet aplikacija. Reč je o aplikacijama koje podržavaju sve platforme. One nisu nativne aplikacije, jer se izvršavaju u *webview-u* uređaja. Takođe nisu ni internet aplikacije, jer imaju pristup nativnom *API-u (Application programming interface)* uređaja. *Apache Cordova* uz saradnju sa *AngularJS* i *Ionic framework-om*, omogućava kreiranje hibridne mobilne aplikacije. Na slici 2.1 su prikazane arhitekture nativne, internet i hibridne aplikacije:



Slika 2.1: Arhitektura native, internet i hibridne aplikacije

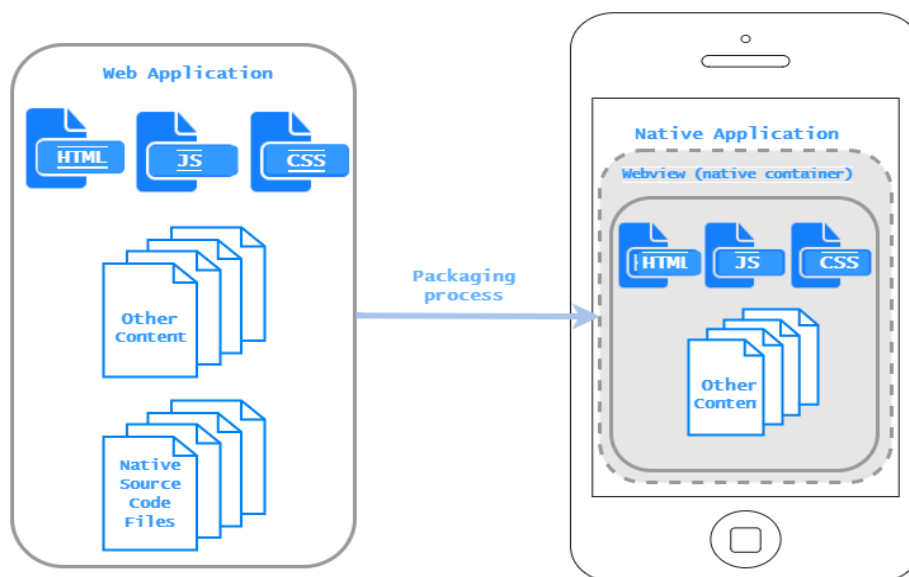
2.2 Apache Cordova

Apache Cordova je besplatan *framework* za izgradnju *cross-platform* nativnih aplikacija upotrebom internet tehnologija: *HTML5*, *CSS3* i *JavaScript-a*. *Framework* je nastao kao rezultat jednostavnijeg kreiranja *cross-platform* mobilnih aplikacija, zasnovan na kombinaciji native i internet aplikacije. Primarna korist je omogućavanje upotrebe nativnih funkcionalnosti izvan mobilnog pretraživača. Najbolji način da se razvije aplikacija u jednom programskom jeziku, koja je podržana na uređajima različitih platformi, jeste upotrebom internet tehnologija. Za mnoge programere potrebno je da aplikacija radi mnogo više nego što internet tehnologije i pretraživači pružaju. Razvoj internet aplikacije koja ima interakciju sa nativnim hardverom i nativnom funkcionalnošću je bilo nemoguće, do pojavljivanja *Cordova-e*. *Cordova* implementira *API-e* koji pružaju native mogućnosti uređaja internet aplikaciji, koja se izvršava u nativnom kontejneru.

Prethodni paragraf čini odgovor na razlog izbora *Apache Cordova framework-a*. U nastavku rada detaljno će biti predstavljena arhitektura i način rada *Apache Cordova-e*.

Da bi se izgradila *Cordova* aplikacija potrebno je izgraditi internet aplikaciju, proslijediti aplikaciju u nativni kontejner, testirati, otkloniti greške aplikacije i zatim je distribuirati

korisnicima. Izvršavanje prethodnog proces omogućavaju tri komponente *Cordova-e*. Prvu čini *source* kod za nativni kontejner aplikacije za svaku platformu uređaja. Druga komponenta je skup osnovnih *API-a* koji omogućavaju aplikaciji pristup nativnim mogućnostima koje su podržane izvan pretraživača. Skup osnovnih *API-a* su predstavljeni kao *plugin-ovi*. Skup alata koji se koriste za kreiranje projekta, upravljanjem životnog ciklusa *plugina*, kreiranja nativne aplikacije (koristeći nativni *SDK*) i testiranje aplikacije na simulatoru i emulatoru, predstavlja trecu komponentu. Kreiranje *Cordova* aplikacije, odnosno pakovanje internet aplikacije u nativni kontejner prikazano je na slici:



Slika 2.2: Proces pakovanja internet aplikacije u nativni kontejner

Nemodifikovana internet aplikacija jednostavno radi u *shell-u* (korisnički interfejs za pristup operativnom sistemu) nativne aplikacije. Prilikom prvog upoznavanja sa *Cordova framework-om*, mnogi pretpostave da se internet aplikacija nekako preved u nativni programski jezik za svaku mobilnu platformu (*Objective C* za *iOS* ili *Java* za *Android*), međutim prave grešku tom pretpostavkom.

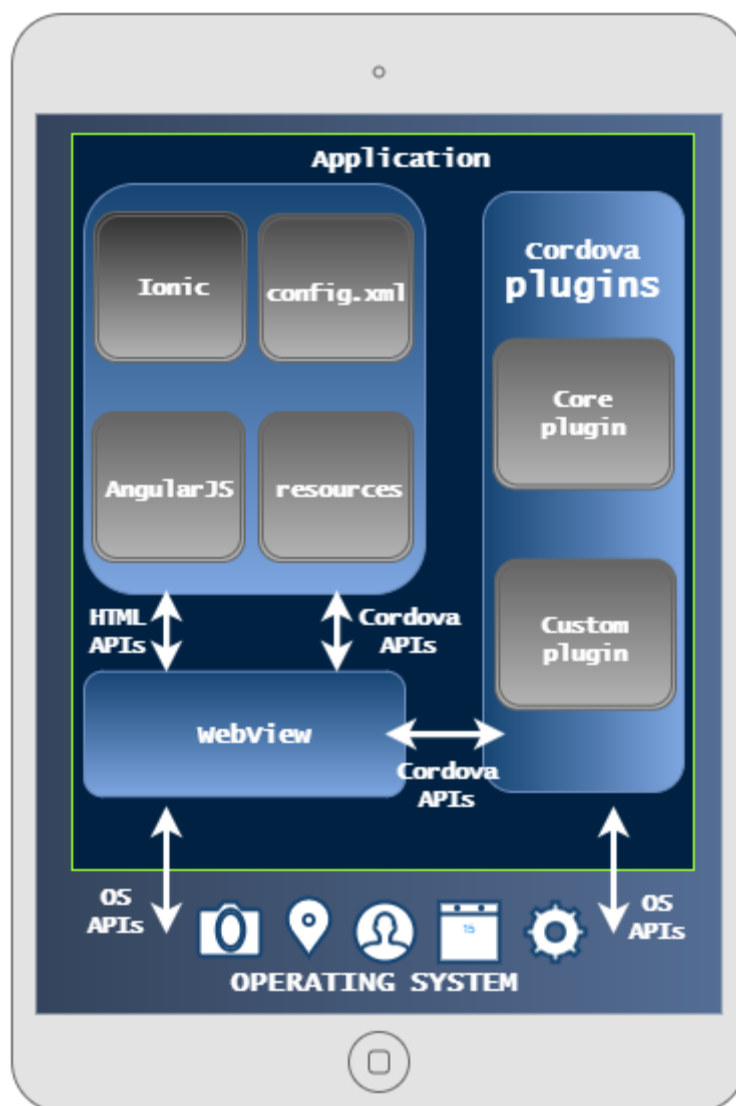
UI se sastoji od jednog ekrana koji sadrži samo *webview*. *Webview* maksimalno koristi raspoloživi prostor ekrana. Pokretanjem aplikacije, učitava se *start-up* stranica internet aplikacije u *webview* uređaja, ako prođe *webview* kontrolu omogućava se korisniku interakcija sa aplikacijom. U većini slučajeva *start-up* je *index.html* stranica, ali vrlo lako se može odabrati druga željena stranica. Aplikacija omogućava učitavanje drugog sadržaja na dva načina. Prvi je iz resursnih datoteka aplikacije, a drugi je dobavljanjem sadržaja sa internet servera ili servera aplikacije.

Rad internet aplikacije u nativnom kontejneru u potpunosti je isti kao rad internet aplikacije u internet pretraživaču mobilnog uređaja. Aplikacija se sastoji od internet tehnologija: *HTML*, *JavaScript* i *CSS*. *JavaScript* obezbeđuje logiku aplikacije, a za izgled brine se *CSS*. Sve što programer može uraditi u internet aplikaciji, takođe može i u *Cordova* aplikaciji.

Tipične internet mobilne aplikacije nemaju pristup nativnim mogućnostima uređaja, hardveru i nativnom *API-u*. Internet aplikacije ne mogu imati pristup ili interakciju sa nativnim funkcionalnostima i mogućnostima, kao što su: kontakti, mikrofon, kamera itd. Nativne mobilne aplikacije često će koristiti prethodno pomenute mogućnosti uređaja. Jedino aplikacije koje omogućavaju pristup nativnim mogućnostim uređaja mogu ispuniti svaki zahtev potencijalnih korisnika. Nagoveštava se potreba aplikacija da imaju pristup nativnim mogućnostima uređaja.

Cordova omogućava pristup nativnim mogućnostima uređaja obezbeđivanjem *JavaScript API-a*. Preko pomenutog *API-a*, internet aplikacije pomoću *Cordova-e* imaju pristup mogućnostima uređaja izvan pretraživača. Strukturu *API-a* čine *JavaScript* biblioteka i nativni programski kod. *JavaScript* biblioteka izlaže nativne mogućnosti internet aplikaciji, uz podršku odgovarajućeg nativnog programskog koda. Navedeni proces se odvija u kontejneru koji implementira nativni dio *API-a*. U suštini *API* je implementiran kao jedna *JavaScript* biblioteka sa posebnom nativnom implementacijom za svaku podržanu platformu mobilnog uređaja.

Od verzije 3.0 svaki *Cordova API* smešten je u poseban dodatni program (*plugin*). Dozvoljeno je dodavanje i uklanjanje dodataka iz *Cordova* projekta pomoću *CLI* (*Cordova command line interface*) ili *plugmana* (*plugin* menadžera). Prethodni pristup omogućava razvoj aplikacije samo sa potrebnim *plugin-ovima*. Novi pristup je obezbedio arhitekturu *Cordova* aplikacije prikazanu na slici 2.3:



Slika 2.3: Arhitektura *Cordova* aplikacije

Cordova je kompatibilna sa dva *frameworka*: *AngularJS*[8] i *Ionic*[9].

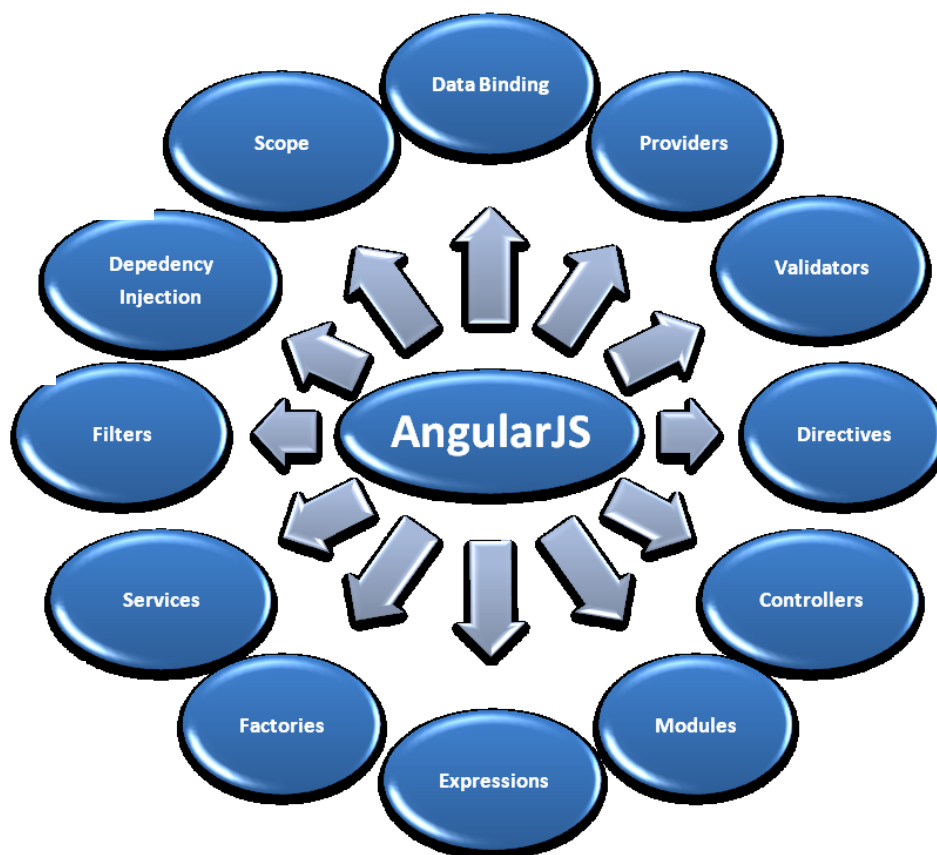
2.2.1 *AnuglarJS framework*

Cordova koristi *AngularJS* za logiku aplikacije kroz *MVC* (*Model View Controller*) šablon. Takođe je *open-source farmework*.

AngularJS je strukturalni framework za dinamičke internet aplikacije. Koristi *HTML* jezik i omogućava proširivanje *HTML* sintakse u cilju jasnijeg definisanja poslovne logike aplikacije. Smanjuje kod za razvoj aplikacije omogućivanjem *data bindings* u samom internet pretraživaču.

Jedna od njegovih osnovih odlika je mogućnost da kreira *RIA* (*Rich Internet Applications*). Korisnici u *JavaScript* mogu pisati klijenstku stranu aplikacije u *MVC* šablonu. U

današnjici postoji mnogo korisnika različitih internet pretraživača. *Framework* obrađuje *JavaScript* kod i prilagođava za svaki pretraživač. *AngularJS* predstavlja *framework* za kreiranje internet aplikacija koje su komplekse, lake za održavanje (*easy-to-maintain*) i zahtevaju visoke performanse. Slika 2.4 prikazuje bitne delove *AngularJS frameworka*. Određeni delove diskutovaće se u dalje tekstu.



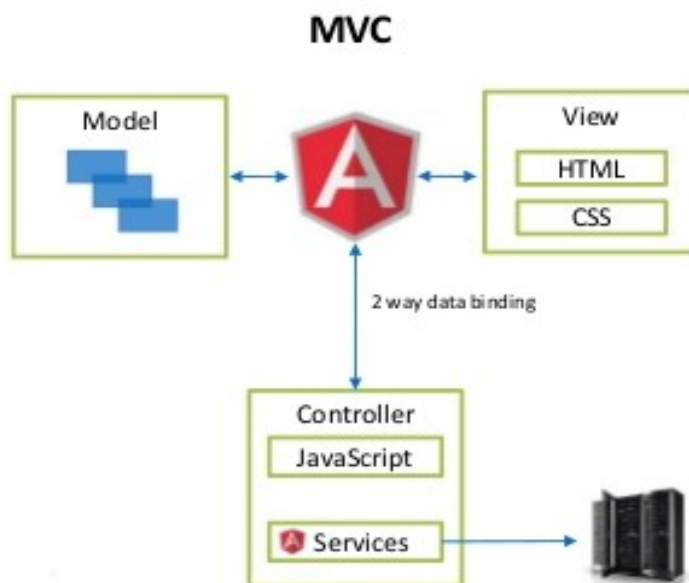
Slika 2.4: Bitni delovi *AngularJS framework-a*

2.2.1.1 **MVC** šablon

MVC je softverski dizajn šablon za kreiranje internet aplikacija. Sastoji se iz nekoliko nivoa. Prvi najniži nivo odgovoran je za održavanje podataka i naziva se *Model*. *View* predstavlja drugi nivo koji je odgovoran za prikazivanje svih podataka ili određenog dela podatka korisniku. Treći nivo je *Controller* koji predstavlja softverski kod za kontrolisanje interakcije između *Modela* i *Viewa*.

MVC je popularan jer izoluje logiku aplikacije od sloja korisničkog interfejsa. *Model* odgovara na zahteve *Viewa* i vrši ažuriranje podataka na osnovu instrukcija iz *Controllera*. *Controller* prima sve zahteve od aplikacije i kroz saradnju sa *Modelom* priprema podatke koji su

potrebi *View-u*. *View* koristi podatke dobijene od *Controllera* da bi generisao finalnu prezentaciju odgovora. *MVC* apstrakcija je prikazana na slici 2.5:



Slika 2.5: MVC šablon u *AngularJS framework-u*

2.2.1.2 Moduli

AngularJS podržava modularni pristup. Moduli se koriste kako bi se logika aplikacije podelila i odvojila u nekoliko modula, kao što su: servisi, kontroleri i modul aplikacije. Moduli se definišu u različitim *.js* datotekama. U većini slučajeva postoje sledeći moduli:

- *Application module* - koristi se za pokretanje aplikacije sa servisima i kontrolerima.
- *Controller module* - koristi se za definisanje kontrolera aplikacije.
- *Service module* - koristi se za definisanje servisa aplikacije.

AngularJS prepušta kontrolerima da kontrolišu tok podataka u aplikaciji. Kontroler se definiše pomoću *ng-controller* direktive. Direktiva se dodeljuje *DOM* (*Document Object Model*) elementu za koji je namenjen kontroler. Kontroler je *JavaScript* objekat koji sadrži atribut i funkcije. Svaki kontroler prihvata *\$scope* kao parametar koji se odnosi na aplikaciju ili modul kojim treba da rukuje.

2.2.1.3 Service (Servisi)

Framework ispunjava princip *separation of concerns* (*SoC*) upotrebom arhitekture servisa (*service*). *JavaScript* funkcije koje su odgovorne za obavljanje konkretnih zadataka nazivaju se

servisi. Kontroleri ostvaruju interakciju sa njima upućivanjem jednostavnog poziva. Servisi se prosleđuju uporebom *AngularJS dependency injection* mehanizma. *AngularJS* pruža mnoge ugrađene servise kao što su: *\$http*, *\$q*, *\$windows* itd. Ugrađeni servisi uvek imaju \$ prefiks.

Svi servisi u *AngularJS-u* su singletoni. Postoje dve vrste servisa, to su *Factory* i *Service*. *Factory* je servis koji ima povratnu vrednost. Koristi se kao *factory* funkcija koja vrši računanje vrednosti koja postaje povratna vrednost *factory* funkcije. *Service* je *JavaScript* objekat koji sadrži skup funkcija za obavljanje određenih zahtjeva.

2.2.1.4 Scope

Specijalni *JavaScript* objekat koji povezuje kontroler i *view* naziva se *Scope*. U kontrolerima se modelu podataka pristupa preko *\$scope* objekta.

2.2.1.5 View

AngularJS podržava *SPA (Single Page Application)* sa više pogleda (*view*) na jednoj stranici. *Framework* obezbeđuje *ng-view* i *ng-template* direktive, i *\$routeProvider* servis, kako bi omogućio više pogleda na jednoj stranici.

Ng-view direktiva mapira prostor gdje se može kreirati odgovarajući *view* na osnovu konfiguracije.

Ng-template direktiva se koristi za kreiranje *HTML view-a* upotrebom skript tagova. Tagove koristi *\$routeProvider* za mapiranje *view-a* sa kontrolerom.

2.2.1.6 Dependency Injection

Dependency injection predstavlja softver dizajn u kome komponente daju njihovu zavisnost (*dependency*) drugoj komponenti, bez ponovnog kreiranja. Takođe *dependency injection-a* omogućava višestruku upotrebu komponenti. *AnguraJS injector* je pod sistem zadužen za prosleđivanje komponenti preko njihove zavisnosti i činiti ih dostupnim drugim komponentama. *AngularJS* poseduje nekoliko ugrađenih komponenti koje mogu biti prosleđene u bilo koju drugu kao zavisnosti, a to su komponente: *Value*, *Factory*, *Service*, *Provider*, *Constant*.

2.2.2 Ionic

Ionic je *open-source SDK* za razvoj hibridne mobilne aplikacije. Odlično se uklapa sa *Cordova* i *AngularJS framework-om*. Pojednostavljuje jedan veliki dio aplikacije: *front-end*. Zadužen je za kontrolu izgleda i *UI-a* aplikacije kroz primenu klasa platforme, upotrebom JS za stil aplikacije ili upotrebom dinamičnih šablona.

2.2.2.1 Klase platforme

Ionic framewrok automatski dodeljuje klase platforme u `<body>` element stranice. Klasa se dodeljuje u zavisnosti od uređaja koji pokreće aplikaciju. Ako se koristi *iOS* uređaj, `<body>` elementu će biti dodeljena *platform-ios* klasa. Postoje dve vrste klasa:

- Klase platforme uređaja - koriste se za dobavljanje informacija o uređaju i dodeljuju klase `<body>` elementu stranice. Klase ove vrste se koriste u nameri da se aplikacija što bolje prilagodi specifičnom izgledu i interakciji platforme. U klase platforme uređaja spadaju: *platform-browser*, *platform-cordova*, *platform-webview*, *platform-ios*, *platform-android* itd.
- Klase *OS* verzije platforme - koriste se za prilagođavanje aplikacije za određenu verziju operativnog sistema. Neke od ovi klasa su: *platform-ios8*, *platform-ios8_4*, *platform-android4*, *platform-android4_4* itd.

Pomenute klase se takođe koriste za redefinisane izgleda vizuelnih komponenti. Primer jedne od klasa je prikazan u nastavku:

```
(style.css)
.platform-android .bar-header{
  text-transform: uppercase;
}
```

Slika 2.6: Redefinisanje izgleda komponente upotrebom *Ionic* klasa

2.2.2.2 Kontrola stila aplikacije upotrebom *AngularJS*

Ionic predstavlja *CSS framework* i *JavaScript UI* biblioteku. Mnoge komponente zahtevaju *JavaScript* kako bi dostigle svoj pun potencijal i omogućile funkcionalnost koja se od njih očekuje. Takve komponente omogućava *Ionic framework*, ali uz kombinaciju sa *AngularJS framework-om*.

Preko *AngularJS* direktiva *Ionic* obezbeđuje dinamičan stil elemenata. Jedna od takvih direktiva je *ng-class* na osnovu koje se dinamički dodeljuju klasa elementima. Ako direktivu *ng-class* kombinujemo sa nekom od *Ionic* korisnih metoda proširuju se mogućnost. Na ovom principu *Ionic* u saradnji sa *AngularJS* proširuje svoje mogućnosti i dostiže puni potencijal. Primer upotrebe jedne *AngularJS* direktive sa *ionic.Platform* metodom prikazaj je na slici:

Controller

```
.controller('AppCtrl', function($scope){
    $scope.platform = ionic.Platform.platform();
})
```

HTML

```
<ion-tabs class="tabs-stable" ng-class="{ 'tabs-positive': platform == 'android', 'tabs-icon-top': platform != 'android' }">
    <!-- ion-tab directives go here -->
</ion-tabs>
```

Slika 2.7: Dinamičko generisanje izgleda elemenata

Ionic podržava *View Controller* šablone. U *View Controller* šablonu tretiraju se različiti delovi interfejsa kao poseban *view* koji se smeštaju u *View Controller*. Zatim *View Controller* generiše svaki *view* iz skupa koji se nalazi u njemu, sa ciljem da obezbedi interakciju i *UI* funkcionalnosti za skup generisanih vidljivih panela. U praksi prehodni proces je generisanje (učitavanje) izgleda jedne stranice ili *template-a* za stanje kojem pripada. Primer je prikazan na slici:

```
.state('parent.selectGateway', {
    url: "/selectGateway",
    data: {
        footerButtonGrid: "gridSelectFooter",
        footerButtonList: "listSelectFooter",
        buttonType: "selectRefresh"
    },
    views: {
        'menuContent': {
            templateUrl: "templates/selectGateway.html",
            controller: "selectGatewayCtrl"
        }
    },
    resolve: {
        gateways: function (GatewaysListService) {
            return GatewaysListService.getList();
        },
    }
})
```

Slika 2.8: Generisanje jedne stranice za određeno stanje

Ionic obezbeđuje sve funkcionalnosti koje su dostupne nativnom *SDK Framework* pruža mogućnost kreiranja prilagođenih komponenti i metoda za njihovu interakciju, bez slabljena performansi, ali u kombinovanoj upotrebi sa *AngularJS framework-om*.

2.2.3 Webview

Webview predstavlja nativni kontejner u kome se izvršava hibridna aplikacija. *Webview* čini jedan od glavnih elemenata arhitekture hibridne aplikacije i omogućava izvršavanje aplikacije, ali sa dostupnim pristupom nativnom *API-u*. *Apache Cordova* ugrađuje *HTML5* kod u *Webview*, nakon toga omogućava *FFI (Foreign function interface)* ili *native bridge* za pristup nativnim resursima uređaja.

Prethodni proces mnogima liči kao pakovanje cele internet aplikacije u jedan *iframe* i kreiranjem mosta (*bridge*) za pristup nativnim tehnologima pomoću *Cordova-e*. Treba primetiti pogodnost koja se odnosi na filozofiju ponovne upotrebe koda. Jednom napisan kod može se koristiti svugdje. Kreiranje mobilnih aplikacija pomoću internet tehnologija primenjenih na nativni *API* neizbežno podrazumeva slabljenje performansi.

Webview se nalazi u središtu alata programera, omogućavajući prikazivanje sadržaja aplikacije pred korisnicima na efikasan način. Za potrebe ovog rada pomenaćemo tipove *webview-a* za najdominantnije mobilne operativne sisteme: *iOS* i *Android*.

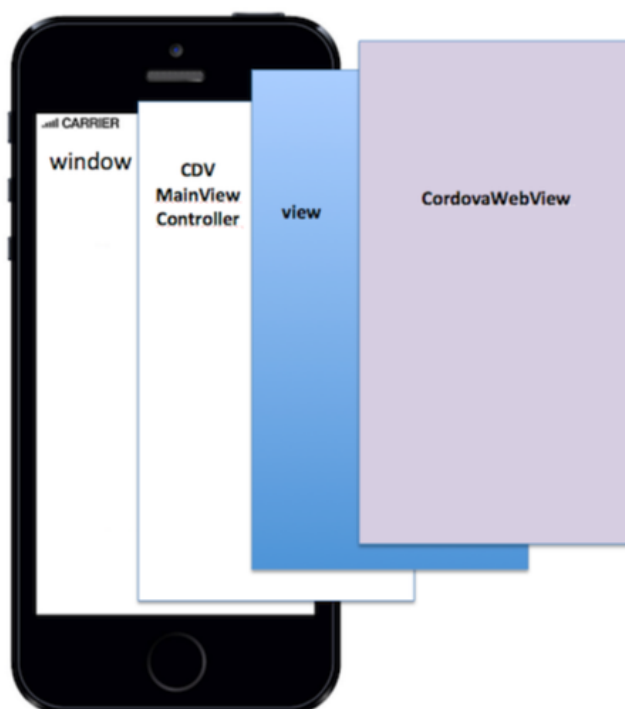
2.2.3.1 iOS

Od momenta kada je *Apple* kompanija donela odluku da ne želi izvršavanje mobilnih aplikacija u mobilnom pretraživaču *Safari*, uvodi se upotreba *webview-a*. Postupak *Apple* kompanije imao je veliki doprinos za hibridne mobilne aplikacije u *iOS-u*. Postoje dve vrste *webview-a* koje podržava *iOS* sistem. To su: *UIWebView* i *WKWebView*.

UIWebView je prvi model koji se našao u upotrebi. Njegove karakteristike nisu bile previše pozitivne. Najviše pažnje su privukli nedostaci koji se odnose na performanse i slabu podršku za *JavaScript* mehanizam. Aplikacije koje koriste pomenuti *webview* se svrstavaju u grupu aplikacija koje su mnogo sporije i poseduju kvarove u odnosu na konkurentne native aplikacije.

Apple kompanija reagovala je uvođenjem *iOS 8* operativnog sistema. Sa *iOS 8* predstavlja se *WKWebView*. Novi model *webview-a* popravljajući sve nedostatke prethodnog modela. Čime aplikacije koje koriste *webview* dobijaju poboljšanja u performansama, te postaju konkurentne sa native aplikacijama. *Cordova* poseduje dodatak (*plugin*) za *WKWebView* i na taj način se olakšava njegova instalacija i upotreba.

Struktura *webview-a* za *iOS* je prikazan na slici:



Slika 2.9: *iOs Webview*

2.2.3.2 Android

Za uređaje koji koriste *Android* operativni sistem, situacija je kompleksnija u odnosu na *iOS*. Starije verzije *Android OS* oslanjale su se na *WebKit* mehanizam za upotrebu *webview-a*. Od 4.4 verzije operativnog sistema, *Android* prestaje da koristi *WebKit* mehanizam i prelazi na upotrebu *Chromium-a*. *Google V8 JavaScript* mehanizam unapređuje *webview*. Svaka verzija *Android-a* podržava drugu verziju *Chromium-a*. Prilikom svakog ažuriranja verzije operativnog sistema uključena je nova verzija *Chromium-a*. Na osnovu toga, pojavljuje se problem u kompatibilnosti podržavanja ranijih verzija *Android-a*.

Jedan od rešenja navedenog problema predstavlja uvođenje koncepta automatskog ažuriranja *webview-a*, koji je uključen u *Androidu 5.0*. *Android 5.0 webview* dostupan je u *Google Play online* prodavnici u formatu *.apk* datoteke. Time je omogućeno ažuriranje *webview-a* u pozadinskom procesu. Ali nije neutralisana zavisnost u odnosu na podržanu verziju *Android* operativnog sistema.

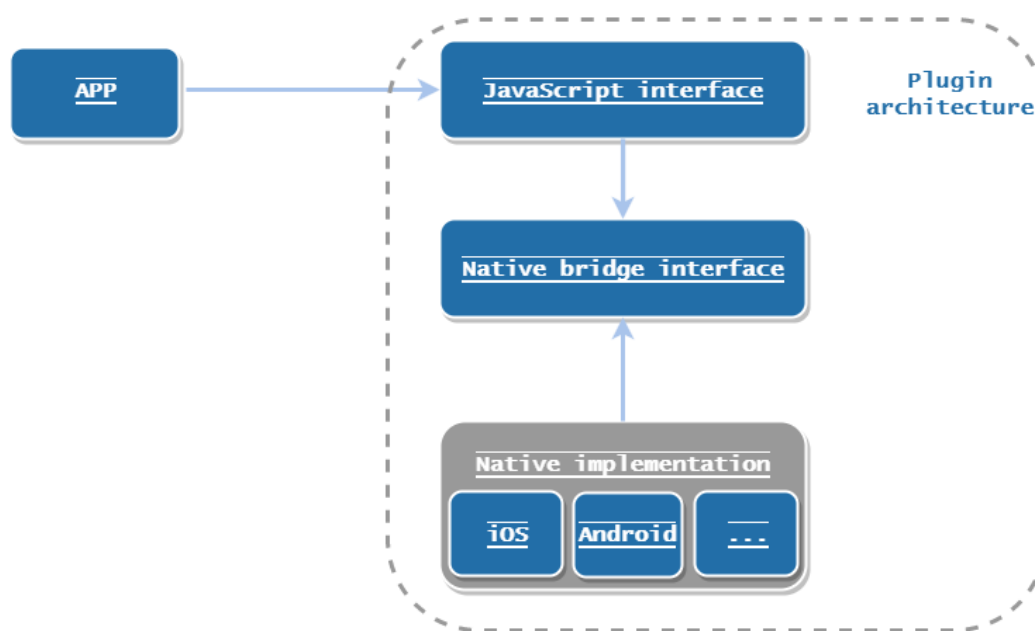
Crosswalk predstavlja drugo, ali i univerzalno rešenje za korisnike *Android-a*. *Intelov Crosswalk* projekat ima zadatak da premosti nedostatke uređaja koji koriste stare verzije *OS-a* i *webview-a*. Rešenje problema je implementirano na način da se izvršavanje aplikacije vrši u sopstvenom *runtime-u*. Njegova upotreba je vrlo korisna za *Android* verzije 4.0 i novije verzije. *Cordova* takođe poseduje *dodatak (plugin)* za *Crosswalk* i ponovo olakšava instaliranje i upotrebu *Crosswalk* projekta. Struktura *Android webview-a* je prikazan na slici:



Slika 2.10: *Android Webview*

2.2.4 *Plugin*

Kao što je pomenuto na početku ovog poglavlja hibridne mobilne aplikacije su nastale kao rešenje za nedostatke internet aplikacija. Osnovo rešenje za nedostatke predstavljaju dodaci (*plugin-ovi*) koji omogućavaju pristup nativnom *API-u* uređaja. Pristup nativnom *API-u* pravi glavnu razliku između internet aplikacija i hibridnih aplikacija, a u isto vrijeme unapređuje mogućnosti hibridnih aplikacija. Dodaci (*plugin-ovi*) omogućavaju pristup funkcionalnostima uređaja i platforme, koje su nedostupne za internet aplikacije. Arhitektura dodatka (*plugin-a*) je prikazana na slici:

Slika 2.11: Arhitektura *Cordova plugin-a*

Dodaci (*plugin-ovi*) su jedan od integrisanih delova *Cordova* sistema. Kreiraju interfejs za *Cordovu* i nativne komponente, kako bi se uspostavila međusobna komunikacija i veza sa standardnim *API-em* uređaja. Strukturu čine *JavaScript* interfejs i biblioteke nativnog koda za podržanu platformu. Takva struktura omogućava sakrivanje raznih implementacija nativnog koda iza zajedničkog *JavaScript* interfejsa. Prilikom kreiranja *Cordova* projekta podrazumevano stanje je da nijedan dodatak (*plugin*) nije uključen u projekat. Potrebno je eksplicitno dodati dodatni program (*plugin*) koji nam je neophodan za aplikaciju. Postoje dva tipa dodataka(*pluginova*): *core* i *custom*. Tipičnu strukturu *Cordova* dodatka (*plugin*) čini: nativni kod za svaku podržanu platformu, odgovarajući *JavaScript* interfejs, nativni interfejs i *plugin.xml* datoteka. *JavaScript* interfejs i nativni interfejs kreiraju most (*bridge*) koji omogućava komunikaciju *JavaScript* koda aplikacije i nativnog koda platforme uređaja.

2.2.4.1 Core plugin

Apache Cordova podržava skup dodataka (*plugin-ova*) koji se nazivaju *Core* dodaci (*plugin-ovi*). *Core* dodaci u prethodnim verzijama *Cordove* bili su uključeni u samu instalaciju *Cordova framework-a* i radili su njihov podrazumevani posao, omogućavanje direktne interakcije sa

uređajem. Neki od *core* dodataka su: *Battery*, *Camera*, *Contacts*, *Device motion*, *Device orientation*, *File transfer*, *Keyboard* itd.

Ne treba zaboraviti da je potrebno eksplicitno dodati dodatak(*plugin*) koji je potreban za implementaciju željene funkcionalnosti aplikacije. Ako aplikacija treba da obezbedi korisniku pristup kameri uređaja, podrazume se potreba za dodavanjem *Camera* dodatka (*plugin-a*). Dodavanjem ostalih *plugin-ova*, bez kojih je moguća implementirati sve funkcionalnosti aplikacije, nije potrebna. Razlog zbog koga se ne dodaju svi dodaci su:

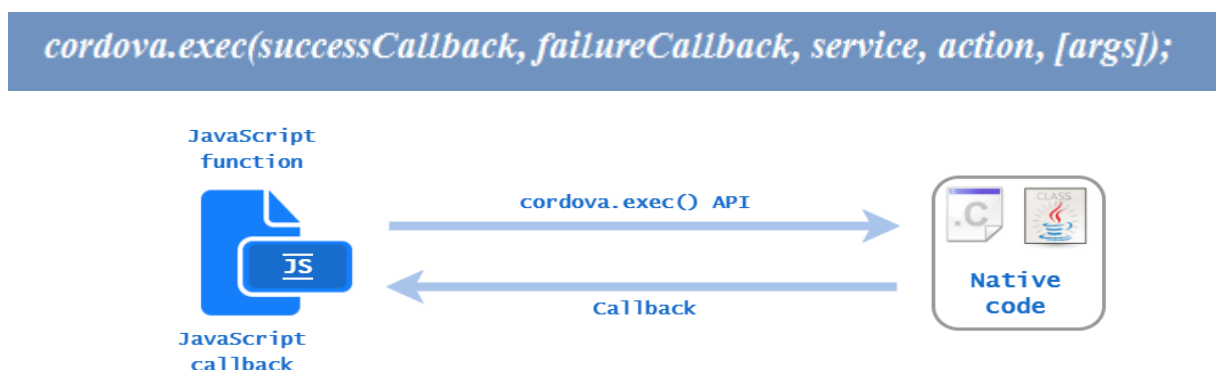
- Upotrebom manjeg broja *core plugin-ova* vrši se brže generisanje rezultata
- Potreba za održavanjem i ažuriranjem *core plugin-ova* na njihove najnovije verzije svedena je na minimum.
- Manja veličina aplikacije

2.2.4.2 Custom plugin

Cordova dodatke možemo pronaći pomoću upotrebe *Cordova plugin* pretraživača ili *npm*. Pored *core* dodataka postoje i nezavisni ili prilagođeni dodaci (*3rd party*) koji pružaju dodatne mogućnosti koje nisu nužno dostupne na svim platformama. Nezavisni dodaci se nazivaju *Custom plugin-ovi*. Za kreiranje *custom* dodataka potrebno je napisati *JavaScript* koji pruža interfejs za pozivanje *plugin-a* iz aplikacije. Zatim nativni kod koji obezbeđuje odgovarajuće potrebne funkcionalnosti. Potrebno je obezbediti nativni kod za sve željene platforme. Sadrži *plugin.xml* datoteka, čiji je značaj mnogo bitan. Datoteka definiše *custom* dodatak i način na koji se ugrađuje u aplikaciju.

Za kreiranje sopstvenog prilagođenog *plugin-a* potreban je običan tekst editor. Za kompleksnije dodatke preporučuje se razvoj i testiranje nativnog koda u *IDE-u* (*Integrated development environment*) svake platforme. Za *Android* uređaje to je *Android Studio*, a za *iOS* *Xcode*.

Kao što arhitektura *Cordova* dodatka ukazuje, kreiranje *custom* dodatka započinjemo *JavaScript* intrefejsom. *JavaScript* intrefejs efikasno definiše šta je potrebno (zahtevano) u nativnom kodu. Najvažniji dio interfejsa je metoda *cordova.exec*, koja se mora definisati u interfejsu da bi nam omogućila pozivanje funkcionalnosti iz nativnog koda. Struktura metode *cordova.exec*:

Slika 2.12: Metoda *cordova.exec*

Cordova.exec omogućava na nativnoj strani poziv *action* metode iz *service* klase, prosleđivanjem argumenata u opcionalnoj listi *[args]*. Ako se nativni kod izvrši uspešno *successCallback* funkcija će biti pozvana. U suprotnom pozvaće se *failureCallback* funkcija, sa opcionim *error* parametrom.

Nativni interfejs takođe sadrži metode koje su od esencijalnog značaja za nativnu stranu dodatka. Ponovo se susrećemo sa malim razlikama kod najdominantnijih mobilnih operativnih sistema *iOS-a* i *Androida*.

- *Android*:

Pored metoda potrebno je da odgovarajuća klasa proširuje (nasleđuje) *CordovaPlugins* klasu. Prva metoda je *initialize* koja se koristi za inicijalizaciju, odnosno *start-up* logiku. U prethodnom tekstu i tokom celog rada, akcenat se stavlja na zajednički rad *JavaScript* koda i nativnog koda. Pozivanje funkcionalnosti nativnog koda omogućava metoda *cordova.exec* sa *JavaScript* strane, ali poslednju reč u ovom procesu ima nativna metoda *execute*. Svaki zahtev koji je poslat kroz metodu *cordova.exec* je prosleđen u metodu *execute* u nativnom interfejsu. Metoda *execute* predstavlja drugu esencijalnu metodu za nativni interfejs. Struktura *execute* metode:

```
public boolean execute(String action, JSONArray args, final CallbackContext callbackContext) throws JSONException { ... }
```

Slika 2.13: Metoda *execute*

Execute metoda se izvršava svaki put kada se pozove metoda *cordova.exec* iz *JavaScript* interfejsa. Metoda se izvršava preuzimanjem prosleđenih parametara, koje čini odgovarajuće ime metode, ulazni paramteri i povratna vrednosti. Nativni most u suštini čine metode *cordova.exec* sa *JavaScript* strane i *execute* sa nativne strane.

- *iOS*:

U odnosu na konkurenciju postoji nekoliko razlika. Nativni interfejs najmanje čine jedna *.h* i jedna *.m* datoteka. U *.h* datoteci ovog nativnog interfejsa potrebno je implementirati *CDVPlugin*. A u *.m* datoteci potrebno je importovati *.h* datoteku i *CDVPlugin* klasu. Takođe prva najbitnija metoda je *initialize* koja se koristi za inicijalizaciju, odnosno *start-up* logiku koja se nalazi u *.m* datoteci. Operativni sistem *iOS* ne sadrži *execute* metodu nego svaka metoda se implementira posebno, ali uslov je da se ime metode i naziv akcije pozvane od *JavaScript* strane poklapaju.

```
(void) action: (CDVInvokedUrlCommand*) command { ... ... }
```

Slika 2.14: *Action* metoda

Metode koje pripadaju nativnom interfejsu nazvaćemo *action* metode, zbog lakše upotrebe. Da bi se bolje razumeo prethodni postupak još jednom je potrebno pogledati strukturu *cordova.exec* metode i obratiti pažnju na parametar *action* i njegovu ulogu na nativnoj strani.

Vrednost argumenata koji su prosleđeni metodom *cordova.exec* se dobija komandom *objectAtIndex*. Nakon što se odradi logika metode, prosleđuje se povratna vrednost koja određuje da li će biti pozvane *success* ili *failure* metode u *cordova.exec*. Nativni most predstavlja *cordova.exec* metoda zajedno sa *action* metodom.

2.2.4.3 plugin.xml

Plugin.xml datoteka je *XML* dokument koji definiše *plugin* i sadrži uputstva za način na koji će *plugin* biti dodat u *config.xml* datoteku aplikacije. Osnovni elementi koje treba definisati u *xml* datoteci su: *namesapce*, *id*, *description* i *version* (verzija). *Name* i *description* elementi trebaju uvek biti definisani. Ostali elementi se definišu u zavisnosti od namene upotrebe dodatka (*plugin*a), npr. ako je namenjen za javnu upotrebu treba definisati elemente kao što su: autor, licenca, repozitoriju, verzija itd. Verzija predstavlja minimalnu podržanu verziju *Cordove*. Primer *XML* datoteke je prikazan na slici 2.15:

```

?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://apache.org/cordova/ns/plugins/1.0"
  id="com.plugin.example"
  version="0.0.1">

  <name>Example</name>
  <description>Cordova plugin</description>

  <engines>
    <engine name="cordova" version=">=3.6.0" />
  </engines>

  <js-module src="www/example.js" name="Example">
    <clobbers target="Example" />
  </js-module>

  <!-- android -->
  <platform name="android">
    <config-file target="res/xml/config.xml" parent="/*">
      <feature name="Example">
        <param name="android-package" value="com.plugin.alert.Example" />
      </feature>
    </config-file>
    <source-file src="src/android/Example.java" target-dir="src/com/plugin/example" />
  </platform>

  <!-- ios -->
  <platform name="ios">
    <config-file target="config.xml" parent="/*">
      <feature name="Example">
        <param name="ios-package" value="Example" />
      </feature>
    </config-file>
    <header-file src="src/ios/Example.h" />
    <source-file src="src/ios/Example.m" />
  </platform>

</plugin>

```

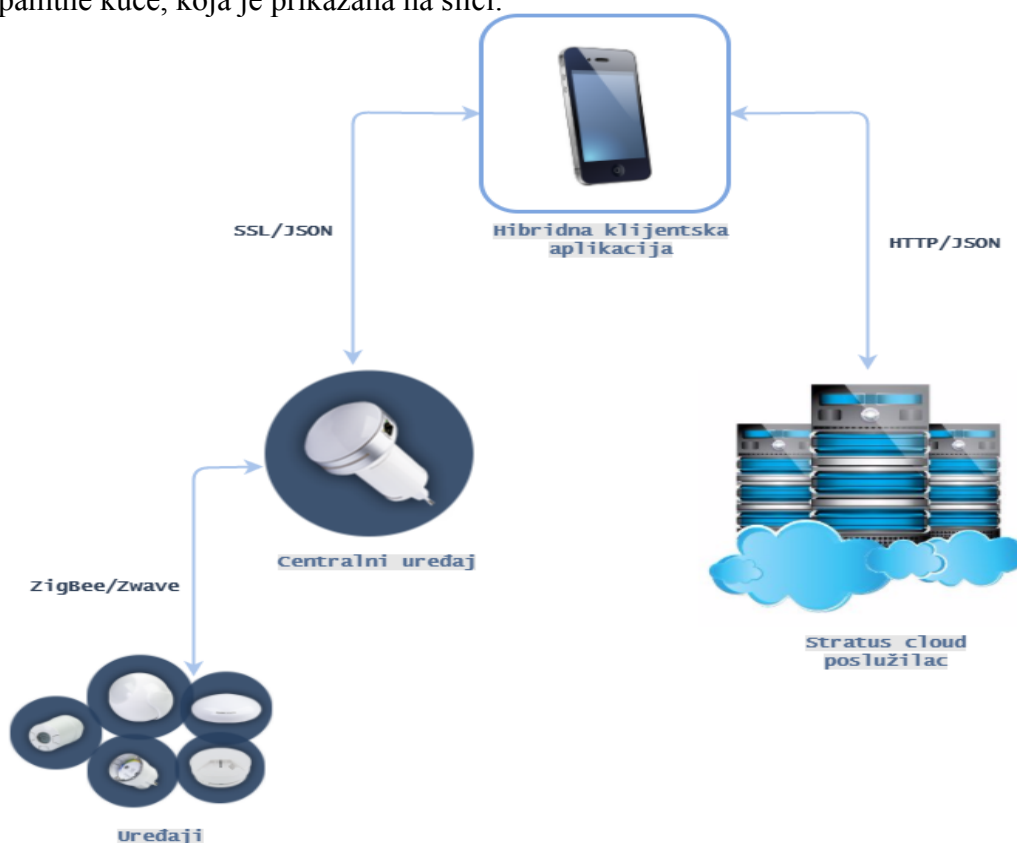
Slika 2.15: *plugin.xml* datoteka

Tag *platform* je jedan od najbitnijih elemenata. Osim što definiše podržane platforme, u njemu se definišu putanje do *source* datoteka *JavaScript* i *native* interfejsa. Putanja do *source* datoteke za *JavaScript* interfejs nalazi se između *js-module* tagova. *Source-file* tagovi označavaju putanje do natvnih *source* datoteka. Tag *header-file* se koristi za *iOS* operativne sisteme i putanje do *.h* datoteka.

Svaki *plugin* treba se pravilno navesti i opisati u *config.xml* datoteci od same aplikacije. Na taj način omogućava se pravilna upotreba *custom* dodatka (*plugina*).

3. Opis rešenja

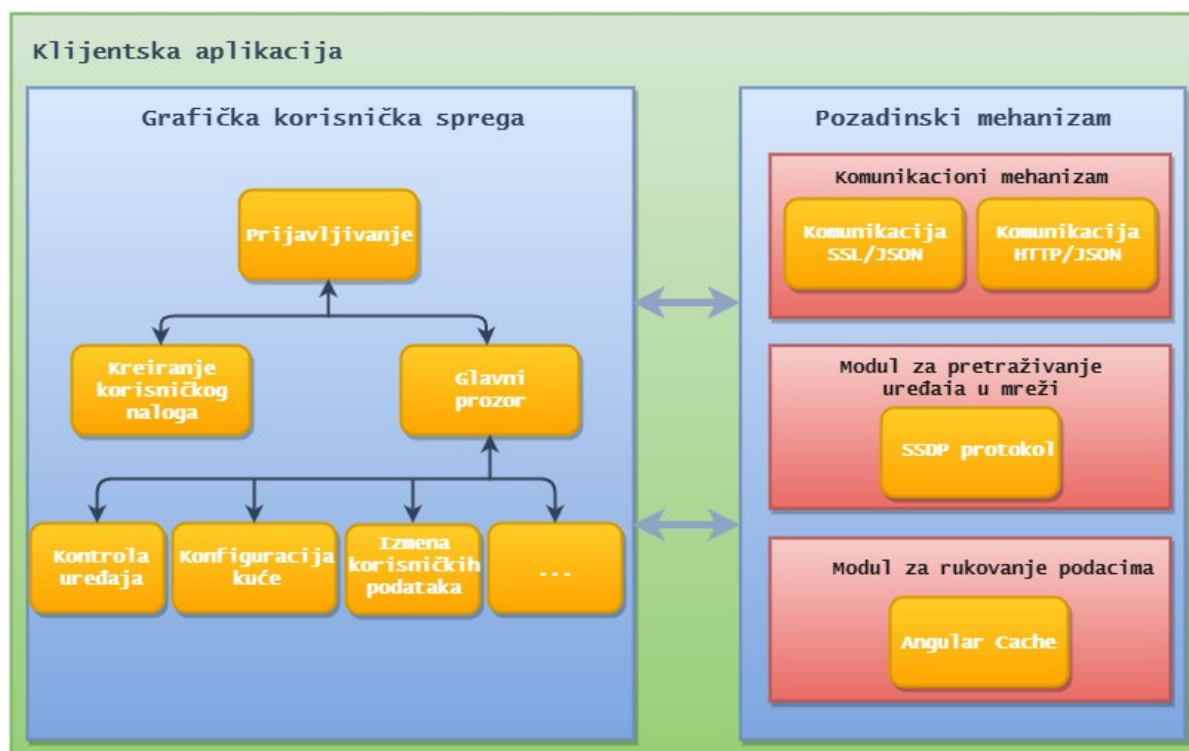
Za razvoj i realizaciju hibridne aplikacije korišteno je razvojnom okruženje *Visual Studio*[10] sa dodatkom za *Apache Cordova*. U ovom poglavlju biće prikazan koncept implementacije određenih funkcionalnosti. Najveća pažnja usmjerena je na funkcionalnosti: pronalaze centralnog uređaja u lokalnoj mreži, komunikacija klijentske aplikacije sa centralnim uređajem i kontrola uređaja. Programski jezici korišteni za realizaciju rešenja su *JavaScript* i programski jezici podržane mobilne platforme. Polazna tačka je arhitektura kompletnog sistema pametne kuće, koja je prikazana na slici:



Slika 3.1: Arhitektura celokupno sistema pametne kuće

Sistem se sastoji iz dva dela: klijentskog i poslužiteljskog. Klijentski dio čini hibridna aplikacija, a poslužiteljski dio čine centralni uređaj pametne kuće i *Stratus* poslužilac. Centralni uređaj predstavlja centralni dio pametne kuće, a *Stratus* poslužilac služi za upravljanje pametnom kućom sa udaljene lokacije.

Klijentska aplikacija se sastoji od dva glavna modula. Glavni moduli su Grafička korisnička sprega i Pozadinski mehanizam. Arhitektura klijentske aplikacije prikazana je na slici 3.2:



Slika 3.2: Arhitektura klijentske aplikacije

3.1 Pozadinski mehanizam

Pozadinski mehanizam čine tri pod modul:

- Komunikacioni pod modul
- Pod modul za pronalazak uređaja u mreži
- Pod modul za rukovanje podacima

Pravilan rad aplikacije najviše zavisi od ovih pod modula i zbog toga čine osnovu kompletne aplikacije.

3.1.1 Komunikacioni pod modul

Komunikacioni pod modul čini jednu zasebnu celinu koja služi za komunikaciju sa centralnim uređajem pametne kuće. Podržava dve vrste komunikacije:

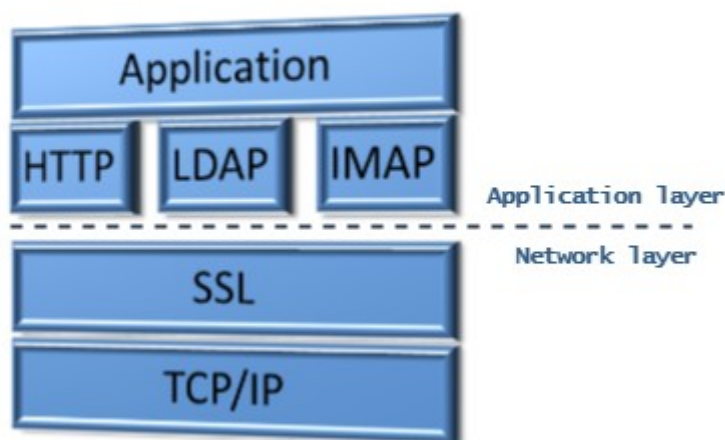
- komunikacija putem *SSL/JSON* protokola
- komunikacija putem *HTTP/JSON* protokola

Komunikacija putem *SSL/JSON* protokola omogućava komunikaciju sa centralnim uređajem putem *SSL[11]* osiguranog komunikacionog kanala. Komunikaciju je potrebno realizovati kao asinhroni zadatak. Aplikacija sa centralnim uređajem razmenjuje poruke u *JSON* formatu. Sadržaj poruke zavisi od komande koja se šalje. Zbog velikog broja komandi, zahteva se modularnost prilikom implementacije komunikacionog pod modula. Modularnost omogućava *AngularJS framework* preko njegovih modula. U ovom slučaju modula *serivice*. Svaka komanda je realizovana kao zasebna funkcija servisa. Komunikacioni pod modul će pozivati odgovarajuću funkciju servisa u zavisnosti od potrebne komande.

Komunikacija putem *HTTP/JSON* protokola omogućava komunikaciju sa centralnim uređajem putem *HTTP* protokola. Komunikacija se odvija preko *Stratus* poslužioca i njena organizacija je slična kao organizacija prethodne komunikacije.

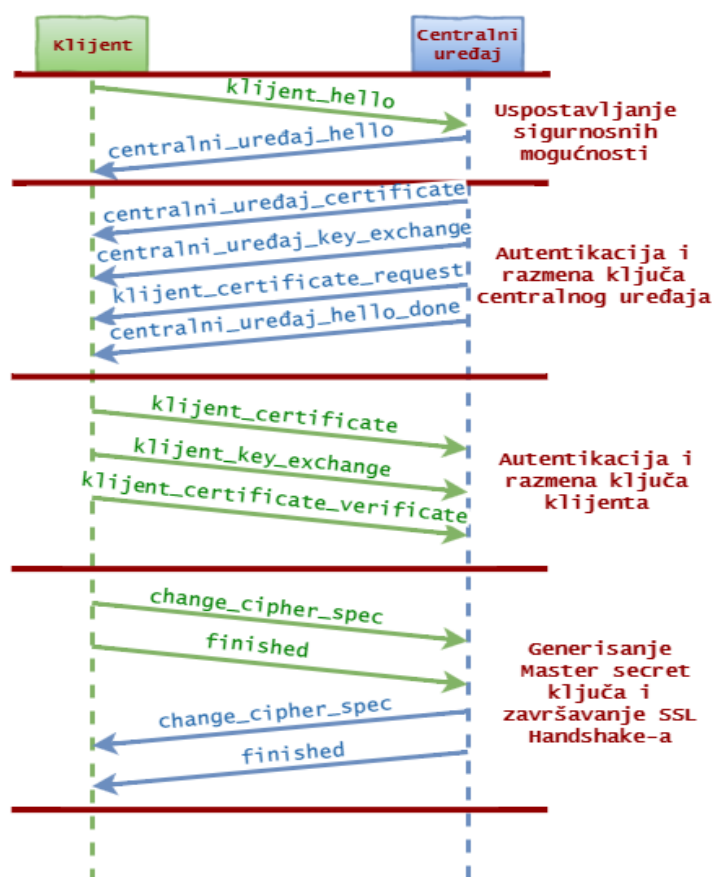
3.1.1.1 Realizacija modula za komunikaciju

Pod modul za komunikaciju predstavlja osnovni dio ovog modula. Komunikacioni modul realizovan je kao pozadinski mehanizam. Veza prema centralnom uređaju realizovana je upotrebom *SSL (Secure Socket Layer)* mrežnog protokola.



Slika 3.3: Nivoi protokola komunikacije

SSL protokol sadrži dva pod protokola: *SSL record* protokol i *SSL handshake* protokol. Protokol *SSL record* definiše format za prenos podataka. A *SSL handshake* protokol podrazumeva upotrebu *SSL record* protokla za razmenu niza poruka između servera i klijenta kada prvi put uspostave SSL konekciju. SSL pruža dodatnu sigurnost. Postupak prijave na centralni uređaj je prikazan na slici 3.4:



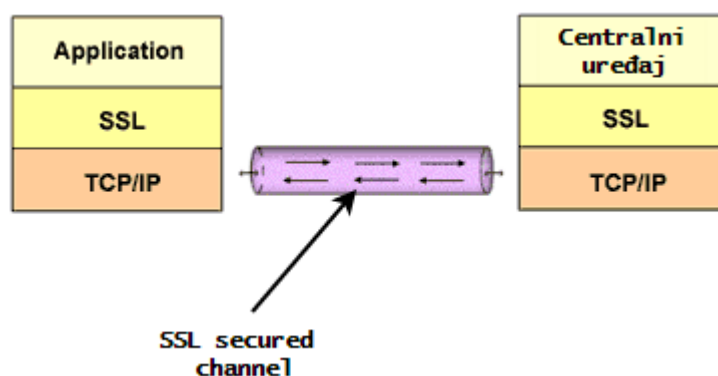
Slika 3.4: Uspostavljanje veze sa centralnim uređajem

Klijent i centralni uređaj sadrže svoje privatne i javne ključeve. Koji će im biti potrebni u ostvarivanju komunikacije i enkripcije. Postupak prijave na centralni uređaj podijeljen je u četiri faze:

- Uspostavljanje sigurnosnih mogućnosti - ovu fazu čine dve poruke koje uspostavljaju sigurnosne mogućnosti. Mnogi pogreške prilikom tumačenja značaja poruke dobijene od klijenta. Sadržaj poruke *klijent_hello* čini lista mnogih polja. Najbitnija polje je *Cipher Suite*. Vrednost polja predstavlja lista čiji elementi su kombinacije algoritama. Klijent predlaže centralnom uređaju nekoliko kombinacija algoritama, jer se ne koristi samo jedan algoritam za celokupnu enkripciju. Jedan algoritam se koristi za *key exchange* i autentikaciju, a drugi algoritam za enkripciju. Sadržaj polja *random_bytes* je takođe bitan i biće iskorišten u narednim fazama.

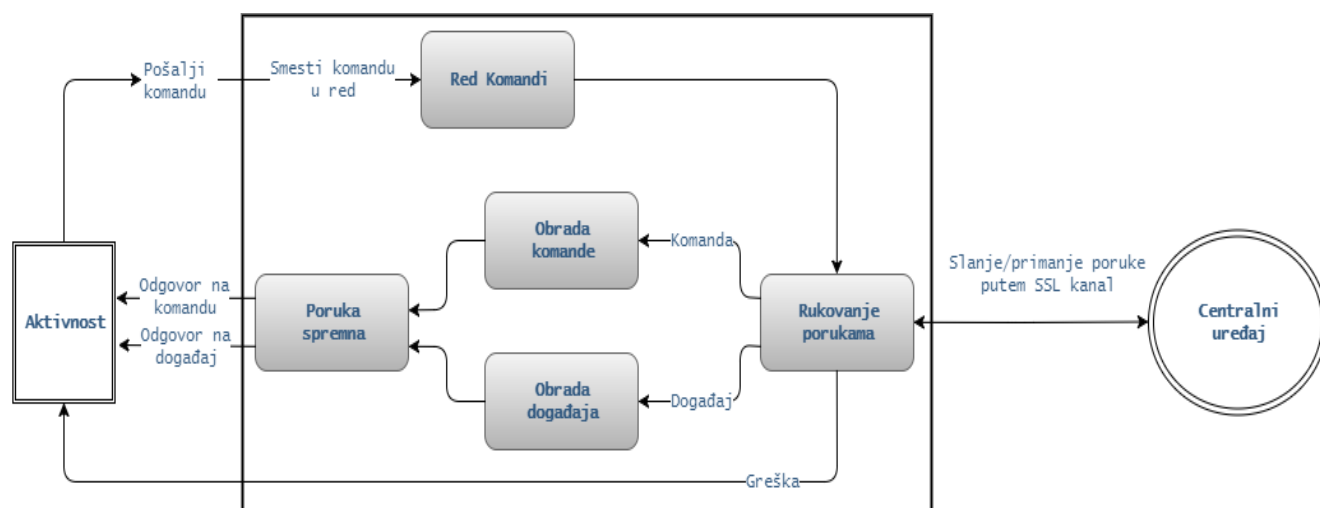
Centralni_uređaj_hello poruka takođe sadrži polje *random_byte* čiju je vrednost izražena u brojevima. *Cipher Suite* polje bira jednu kombinaciju algoritama iz liste ponuđenih algoritama od strane klijenta.

- Autentikacija i razmena ključa centralnog uređaja - nakon preuzimanja kombinacije algoritama centralni uređaj šalje njegov sertifikat i na taj način prosleđuje klijentu javni ključ. Zatim centralni uređaj zahteva klijentski sertifikat u kome će biti javni ključ klijenta. Nakon primanja sertifikata, klijent se uvjerava u ispravnost dobijenog sertifikata. Nakon ovog procesa klijent dobija vrednost javnog ključa centralnog uređaja i *random number* centralnog uređaja. Vrednost *random number-a* i polja *random_bytes* je ista i odnosi se i na klijenta i na centralni uređaj. Sa druge strane, centralni uređaj takođe sadrži njegov i klijentov *random number*.
- Autentikacija i razmena ključa klijenta - u ovoj fazi se realizuje sličan proces kao u prethodnoj. Sada su uloge drugačije, jer se klijent identifikuje kod centralnog uređaja. Klijent šalje sertifikaciju i javni ključ centralnom uređaju. Sledi validacija sertifikata klijenta po odgovarajućim poljima i preuzimanje njegovog javnog ključa. Da bi klijent obezbedio vezu samo između njega i centralnog uređaja. Prilikom prijema sertifikata centralni uređaj vrši dekripciju pomoću javnog ključa klijenta, dobijajući privatni ključ klijenta. Nakon dekripcije centralni uređaj ima sve informacije o klijentu.
- Generisanje *Master secret* ključa i završavanje *SSL Handshake-a* - potrebno je da obe strane prethodno generisu *Pre-Master secret*. Prvo se *Pre-Master secret* generiše na strani klijenta. Nakon čega se prosleđuje centralnom uređaju sa javnim ključem centralnog uređaja. Nakon dekripcije *Pre-Master Secret-a*, pomoću privatnog ključa centralnog uređaja, počinje navedeno generisanje *Master secret* ključa na strani centralnog uređaja. Podaci se šalju enkriptovanim *Master Secret* čime je završen proces *SSL Handshake*. Obezbeđen je *SSL kanal* za siguran prenos podataka.



Slika 3.5: SSL kanal za prenos podataka

Mehanizam slanja komandi ka centralnom uređaju realizovan je kao pozadinski mehanizam u cilju postizanja efikasnije i bržeg odziva sistema. Struktura modula za komunikaciju je prikazana na slici 3.6:



Slika 3.6: Struktura modula za komunikaciju

Kada neki modul aplikacije želi poslati komandu ka centralnom uređaju poziva kontroler koji preko funkcija servisa prosleđuje komandu komunikacionom modulu. Za primer iskoristiće se funkcije iz servisa *OHMServis*: *setPropertyValue()*, *getPropertyValue()*, *getDevice()* itd. Svaka funkcija realizovana je upotrebom *\$http* i *\$q AngularJS* servisa. Jedna od prednosti upotrebe *\$q* servisa su *deferred* i *promise* objekti. Čime je omogućeno asinhrono izvršavanje komandi. Svaka od navedenih funkcija implementira različite komande u vidu poruke u *JSON* formatu. Primer *JSON* poruke:

```
{
  "msg_type": "command",
  "command": "get_object_status",
  "id": 27
}
```

Prototip servisa	• \$http
Metode servisa	• post (\$http.post)
Opis metode	• Svaka komanda treba da redefiniše metodu <i>post</i> . Namena metode je omogućavanje komunikacije i prosledjivanje zahteva komande centralnom uređaju.
Parametri metode	• url – relativna ili apsolutna adresa odredišta • data – zahtev komadne, odnosno sadržaj JSON poruke • config – konfiguracija
Povrtana vrednost metode	• <i>promise</i> objekat
Prototip servisa	• \$q
Metode servisa	• defer (\$q.defer)
Opis metode	• Omogućava asinhrono izvršavanje komandi i kreiranje <i>deferred</i> objekta, koji omogućava pristup povezanom <i>promise</i> objektu. Koristi se u modulu za obradu komande i obradu događaja, odnosno koriste se <i>resolve</i> i <i>reject</i> metode <i>deferred</i> objekta.
Parametri metode	• Nema parametre
Povrtana vrednost metode	• <i>deferred</i> objekat

Tabela 3.1: Ugrađeni servisi *AngularJS framework-a*

- *Deferred API*:
Nova instanca se kreira pozivom *\$q.defer()* metode. Njegova svrha se ogleda u omogućivanju pristupa *promise* objektu. Bitna metoda koja se koristi je:
 - *resolve(value)* - metoda se koristi u modulu *Poruka spremna* i kreira instancu *promise* objekta. Na osnovu kreirane instance funkcija za *odgovor na komandu* ili *odgovor na događaj* ima vrednosti poruke tj. *value*-u.
- *Promise API*:
Nova instanca *promise-a* nastaje kada je kreirana *deferred* instanca. Upotreba *promise* objekta je omogućena kroz povratnu vrijednost *\$http.post* metode. Metoda koja se koristi je:
 - *then* - metoda se koristi u funkcijama *odgovor na komandu* ili *odgovor na događaj*, kroz koju aktivnost dobija vrednosti odgovora na komandu ili događaj. Primjer je upotrebe *then* metode prikazan je na slici 3.7:

```

.controller("AddDeviceController", function (OHMService, $scope, $state, CacheFactory) {
    $scope.getNetworkTimeout = function () {
        var promise = OHMService.getNetworkTimeout();
        promise.then(function (data) {
            $scope.networkTimeout = data.network_timeout;
            $scope.copyValue = data.network_timeout;
        });
    };
});

```

Slika 3.7: Upotreba *then* metode

Koraci koje komunikacioni modul treba obaviti nakon prosleđene komande od strane *Aktivnosti* su:

- Obrati se pod modulu *Red Komandi* koji prosleđenu komandu smesti u red. Prosleđenu komandu prosledi u modul *Rukovanje porukama*.
- Pod modul *Rukovanje porukama* šalje poruku putem *SSL* kanala ka centralnom uređaju. *Centralni uređaj* obradi poruku i preko istog kanala pošalje odgovor. Dobijeni odgovor se u modulu *Rukovanja porukama* razvrstava. Ako se u toku slanja ili primanja poruke desi greška, modul obaveštava *Aktivnost* o toj grešci. Ukoliko je poruka odgovor na poslatu komandu, poruka se prosleđuje modulu za obradu komande. U ostalim slučajevima poruka se prosleđuje modulu za obradu događaja.
- Pod modul *Obrada komande* vrši obradu prosleđene komande. Vrsta obrade komande zavisi od poslate komande i njenog zadatka.
- Obradena poruka se prosleđuje modulu *Poruka spremna*. Instanca *deferred* objekta se kreira u ovom modulu. Koji preko funkcije *odgovor na komandu* obavesti aktivnost o statusu ili povratnim podacima poslate komande. Povratni podaci se regulišu upotrebom *promise* objekta koji je povezan sa *deferred* objektom. Nakon čega su podaci dostupni aktivnosti.
- *Centralnog uređaja* može slati poruke koje nisu komande i one se smatraju događajima. Pod modul *Rukovanje porukama* putem *SSL* kanala dobavi poruku, koju prosledi modulu za *Obradu događaja*. Obradena poruka se prosleđuje modulu *Poruka spremna* koji obaveštava aktivnost pomoću funkcije *odgovor na događaj*. Princip rada modula za obradu i pakovanje poruke za događaje je isti kao i za komande.

U prethodnom tekstu pomenuto je da funkcije u servisu predstavljaju implementaciju komande. Funkcije primaju parametre u zavisnosti od zadatka komande. Svaka funkcija za njenu implementaciju koristi ugrađene servise *AngularJS framework-a*. Primjer jedne komande je prikazan na slici 3.8:

```

.service('OHMService', function ($http, $ionicPopup, $q, $rootScope, CacheFactory, $timeout) {
    var gatewaysListHome = CacheFactory('GatewaysListHome');
    var allDevicesListHome = CacheFactory('AllDevicesListHome');
    var devicesHome = CacheFactory('DevicesHome');

    var deferred = $q.defer();

    /**-- setPropertyValue --*/
    function setPropertyValue(valueProperty, deviceId, serviceName, propertyName, gatewayId) {
        var key = gatewayId + deviceId;

        $http.post(path,
            {
                "msg_type": "command",
                "device_id": deviceId,
                "command": "set_device_property_value",
                "property_name": propertyName,
                "property_value": valueProperty,
                "service_name": serviceName
            },
            {
                headers: {
                    "Content-Type": "application/json",
                }
            })
        .success(function (data, status, headers) {
            if (status == 200) {
                getDevice(deviceId, gatewayId);
                // $rootScope.startUpdateInterval();
                deferred.resolve(status);
            }
            else {
                console.log("Status not OK" + status);
                if (status == "-1") {
                    var alertPopup = $ionicPopup.alert({
                        title: 'ALERT',
                        template: '<p style= "text-align: center;">Please check internet connection.</p>'
                    });
                }
                deferred.reject("Error");
            }
        })
        .error(function (data, status) {
            console.log("Error" + status);
            deferred.reject("Error");
        });
    }
    return deferred.promise;
}

```

Slika 3.8: Primer implementacije komande

Komunikacioni model je realizovan modularno. Što nam omogućava vrlo lako i brzo dodavanje nove komande u sistem. Za dodavanje nove komande potrebno je u određeni servis kreirati funkciju koja će implementirati komandu. Sve prethodno navedene pogodnosti omogućava *AngularJS framework*. Iz prethodnih nekoliko primjera može se primetiti zbog čega se koristi *AngularJS* u implementaciji celog sistema, pre svega ogleda se njegova modularnost koju koristimo kroz modul kontrolera i modul servisa.

3.1.2 Pod modul za pronalazak uređaja u mreži

Pod modul za pronalazak uređaja u mreži obezbeđuje pronalazak centralnog uređaja u lokalnoj mreži. Pronalazak svih dostupnih uređaja u lokalnoj mreži predstavlja osnovni cilj ovog modula. Modul obezbeđuje slanje poruka određenog formata ka svim uređajima u lokalnoj mreži i obradu povratne poruke od uređaja. Filtriranje samo centralnih uređaja u lokalnoj mreži je još jedna mogućnost koju ovaj modul omogućava. Protokol za realizaciju ovog modula je *SSDP*[12] protokol.

3.1.2.1 Realizacija modula za pronalazak uređaja u mreži

Pronalazak uređaja u mreži vrši se upotrebom *SSDP* protokola. Za realizaciju *SSDP* protokola iskorištena je mogućnost koju pruža *Apache Cordova*, a to je kreiranje *custom plugin-a*. *ServiceDiscovery plugin* implementiran je upotrebom dva programska jezika: *JavaScript* i *Java*. Aplikacija preko *JavaScript* interfejsa poziva metode iz *Java* klase. Prilikom pokretanja *SSDP* mehanizma šalju se tri *M-SEARCH* grupne poruke koje su poropisane standardom. Poruke se šalju na grupni kanal koji je definisan *239.255.255.250:1900* vrednostima adrese i porta. *SSDP* je zasnovan na *UDP* protokolu kod kojeg postoji mogućnost gubitka poruke u mreži. Da bi se *SSDP* zaštitio od gubitka poruke, šalje prethodne tri *M-SEARCH* poruke. Način razmene poruka sa centralnim uređajem prikazan je na slici 3.9:



Slika 3.9: Razmena poruka putem *SSDP* protokola

Primljena poruka od strane centralnog uređaja se parsira. Bitno polje u poruci je *LOCATION*, jer sadrži *URL* adresu. *URL* adresa sadrži putanju do *XML* datoteke. Datoteka predstavlja opis samog uređaja i sadrži polja definisana *SSDP* standardom. Primer jedne takve *XML* datoteke je prikazan na slici 3.10:

```

SSDP mehanizam
ssdp.xml

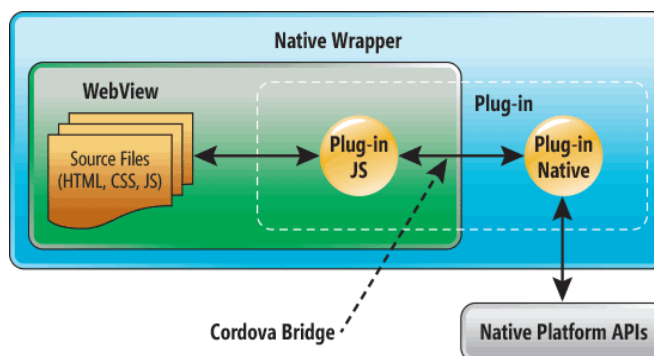
<?xml version="1.0" encoding="utf-8">
<root
  xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>1</minor>
  </specVersion>
  <device>
    <deviceType>urn:dtvresearch-com:device:SchwaigerHomeGateway:1</deviceType>
    <presentationURL>https://192.168.0.104:1443/api</presentationURL>
    <friendlyName>SchwaigerHome HA</friendlyName>
    <manufacturer>DTVResearch</manufacturer>
    <manufacturerURL>http://www.dtvresearch.com/</manufacturerURL>
    <modelDescription>Home automation gateway</modelDescription>
    <modelName>HA101</modelName>
    <modelName>HA101</modelName>
    <serialNumber>00070B00DA82</serialNumber>
    <UDN>uuid:7bbf5796-e675-6f0d-742b-00070b00da82</UDN>
    <iconlist>...</iconlist>
  </device>
</root>

```

Slika 3.10: *XML* datoteka

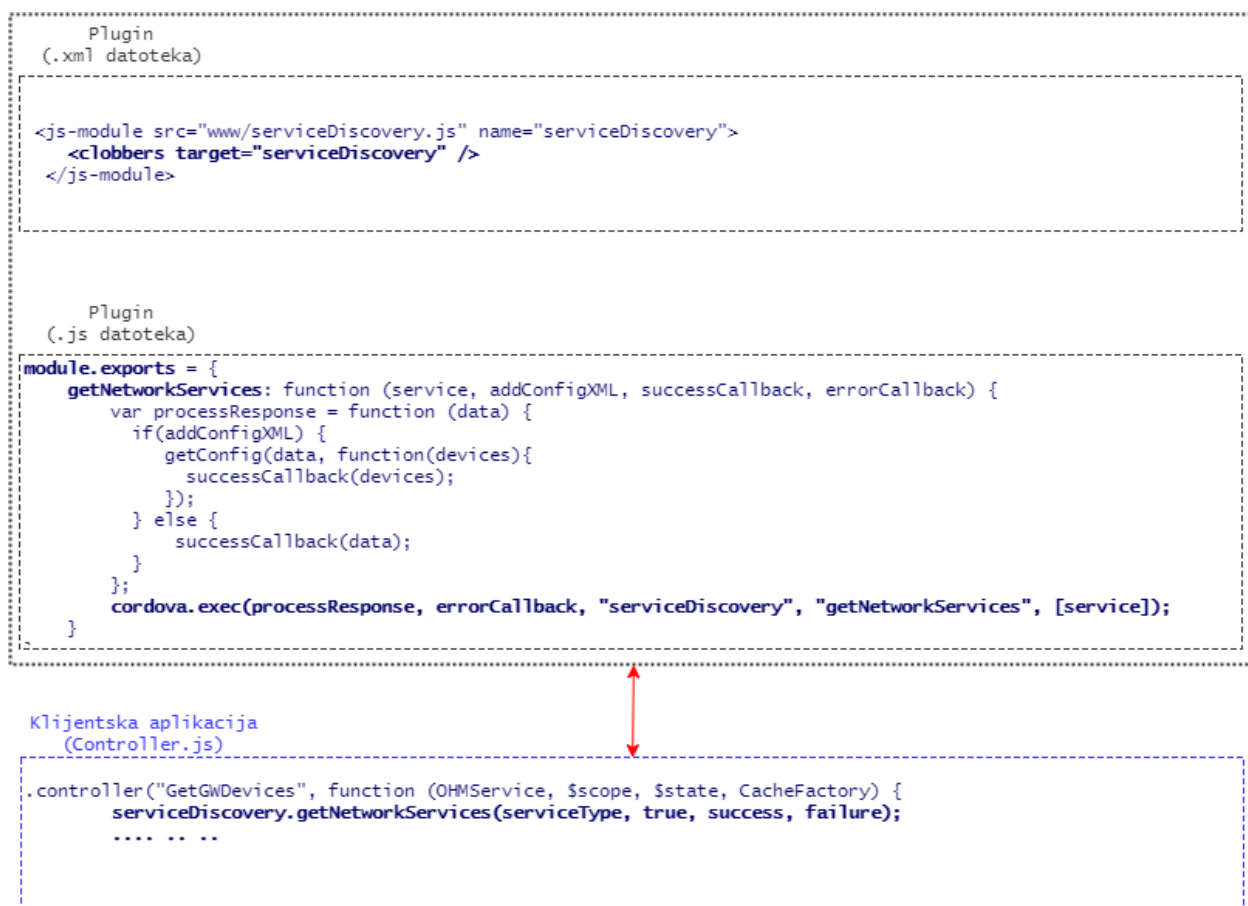
Bitna polja iz datoteke su: *serialNumber*, *friendlyName*, *modelName*, *deviceType*. Vrednost polja koristimo u daljem procesu, koji se odnosi na dodavanje centralnog uređaja u listu dostupnih centralnih uređaja. U daljem tekstu detaljno se prolazi kroz implementaciju *plugin-a* za realizaciju *SSDP* mehanizma.

Plugin se sastoji od *plugin.xml* datoteke, dva interfejsa. Dva interfejsa čine *native bridge* koji omogućava *Cordova* u cilju uspostavljanja komunikacije između *JavaScript* koda klijentske aplikacije i *Java* koda platforme. Pomenuta dva interfejsa su: *JavaScript* interfejs i *native* interfejs. Oni predstavljaju *API*e preko kojih klijentska aplikacija iskorištava mogućnosti mobilne platforme. Slikovito objašnjenje bi glasilo da *JavaScript* interfejs predstavlja paravan iza koga se izvršavaju metode napisane u programskom jeziku podržane platforme. *Cordova native bridge* je prikazan na slici 3.11:



Slika 3.11: Cordova bridge

Pristup nativnoj strani iz *JavaScript* interfejsa omogućava metoda *cordova.exec*, čije definisanje je obavezno. Implementacija metoda se vrši unutar *module.exports property-a* kako bi se obezbedili pozivi metode izvan interfejsa. *Modul.exports* predstavlja listu metoda. Svaka lista je poseban *statemant*, koji se poziva uz prefiks definisan u *clobber* polju *plugin.xml* datoteke. Na slici je prikaza poziv jedne takve metode:

Slika 3.12: Poziv *plugin* metode

JavaScript interfejs u ovom zadatku predstavlja *serviceDiscovery.js* datoteka. Bitne metode interfejsa su:

1. Prototip metode	<code>cordova.exec(processResponse, errorCallback, serviceDiscovery, getNetworkServices, [service])</code>
Opis metode	Najbitnija metoda za komunikaciju sa nativnom stranom. Preko parametara u konstruktoru dostavlja informacije <i>native bridge-u</i> kako bi se izvršila metoda implementirana na nativnoj strani. Metoda <i>exec</i> se nalazi u <i>getNetworkService</i> metodi.
Parametri metode	• processResposne - <i>success callback</i> pod pretpostavkom da se metoda na nativnoj strani uspešno završi, ova funkcija se izvršava zajedno sa svim njenim parametrima • errorCallback - <i>error callback</i>, ukoliko se metoda na nativnoj strani ne izvrši ispravno, izvršava se ova funkcija sa njenim parametrima • serviceDiscovery - naziv servisa koji se poziva na nativnoj strani. Odgovara nazivu native klase. • getNetworkServices - naziv akcije koja se poziva na nativnoj strani. Odgovara nazivu metode iz prethodno pomenute klase • [service] - argument ili lista argumenata koja se proselđuje u nativno okruženje
Povratna vrednost metode	<i>success</i> i <i>error callback</i>
2. Prototip metode	<code>getNetworkServices(service,addConfigXML,successCallback, errorCallback)</code>
Opis metode	Metoda postavlja vrednost na osnovu koje će <i>SSDP</i> protokol izvršiti pretragu uređaja. Pronaći će se uređaji koji podržavaju pomenutu vrednosti. Nakon poziva <i>exec</i> metode i uspešnog izvršavanja metode na nativnoj strani, dobavlja povratnu vrednost, odnosno listu uređaja u mreži.
Parametri metode	• service - vrednost <i>ST(search target)</i> polja <i>M-SEARCH</i> poruke. • addConfigXML - dodavanje polja <i>config</i> u <i>JSON</i> objekat sa podacima <i>XML</i> datoteke. • successCallback - ukoliko se funkcija izvrši uspešno, izvršava se ova funkcija sa njenim parametrima • errorCallback - ukoliko se funkcija ne izvrši uspešno, izvršava se ova funkcija sa njenim parametrima
Povratna vrednost metode	Metoda nema povratnu vrednost

Tabela 3.2: Metode *JavaScript* interfejsa

Native dio *plugin-a* pisan je u programskom jeziku *Java*. Sadrži dve *Java* klase koje proširuju klasu *CordovaPlugin*. Takođe bitno je dobiti sa interneta *apache cordova jar*, da bi se odgovarajuća biblioteka uvezla u projekat. Pomenute dve klase su: *simpleSD* i *serviceDiscovery*. Klasa *simpleSD* je zadužena za implementaciju *SSDP* mehanizma i omogućavanje komunikacije putem *SSDP* protokola. Klasa *serviceDiscovery* predstavlja interfejs koji poziva *JavaScript* strana. Pa se može reći da ta klasa predstavlja *API* nativne strane.

Klasa *serviceDiscovery* redefiniše *execute* metodu. Pomenuta metoda je bitna, jer preko nje se obezbeđuje pristup mogućnostima koje pruža podržana platforma uređaja. Svaki zahtjev koji je poslat kroz funkcija *cordova.exec* iz *JavaScript* interfejsa prosleđeno je u *execute* funkciju nativnog interfejsa.

1. Prototip metode	<code>execute(String action, JSONArray data, final CallbackContext callbackContext)</code>
Opis metode	Metoda kroz prva dva parametra dobija instrukcije o akciji i argumentima iz <i>JavaScript</i> interfejsa. Ako sve prođe kako treba <i>callback</i> dobija vrednost liste sa uređajima u mreži i njihovim podacima u <i>XML</i> datoteci.
Parametri metode	action - naziv akcije koja je pozvana iz <i>JavaScript</i> interfejsa. Vrednost ovog parametra je potpuno ista kao vrednost prvog parametra <i>exec</i> metode u <i>JavaScript</i> interfejsu. data - lista prosleđenih argumenata iz <i>JavaScript</i> interfejsa callbackContext - prosleđuje vrednost za <i>success</i> proces.
Povratna vrednost metode	Povratna vrednost <i>true</i> ukoliko postoji tražena metoda, <i>false</i> ako ne postoji.
2. Prototip metode	<code>parseHeaderValue(String content, String headerName)</code>
Opis metode	Parsiranje određenog polja poruke, koja je dobijena putem grupnog kanala. Drugi parametar metode se odnosi na naziv polja poruke (npr. <i>Id</i> , <i>Server</i> , <i>Location</i> i druga polja).
Parametri metode	content - poruka koja je dobijena sa grupnog kanala headerName - naziv polja poruke za parsiranje
Povratna vrednost metode	Metoda nema povratnu vrednost

Tabela 3.3 Metode *Native* interfejsa

Proces omogućavanja pretrage centralnog uređaja implementira se u *simpleSD* klasi. Bitne metode pomenutih klasa su:

1. Prototip metode	<code>search(String service, CallbackContext callbackContext)</code>
Opis metode	Razmena poruka putem <i>SSDP</i> protokola, dobavljanje <i>M-SEARCH</i> poruke i njeno parsiranje. Kreiranje <i>JSON</i> objekta koji sadrži polja: <code>id</code> , <code>serviceURL</code> , <code>type</code> , <code>service</code> . Za popunjavanje sadržaja <i>JSON</i> objekta koristi se <i>parseHeaderValue</i> metoda. Nakon kreiranja, objekat se smešta u listu. Objekat predstavlja jedan uređaj u mreži. Lista uređaja se prosleđuje u <i>success callback</i> .
Parametri metode	service - vrednost <i>ST(search target)</i> polja <i>M-SEARCH</i> poruke. callbackContext - <i>success callback</i> popunjava listom otkrivenih uređaja
Povrata vrednost metode	Metoda nema povratnu vrednost

Tabela 3.4: Metoda iz *Java* klase

3.2 Grafička korisnička sprega

Pod moduli Grafičke korisničke sprege implementiraju grafički prikaz aplikacije. Za prikaz svih grafičkih komponenti aplikacija koristi podatke koji se nalaze na centralnom uređaju pametne kuće, smešteni u lokalnoj bazi podataka. Za dobavljanje podataka sa centralnog uređaja potrebno je izvršiti niz odgovarajućih komandi u komunikaciji. Dobijeni podaci se obrađuju i skladište se u *cache* objekte, a zatim se po potrebi podaci čitaju iz odgovarajućeg *cache* objekta.

Cache objekti se koriste za čuvanje i preuzimanje podataka. *Cache* objekte prvenstveno koristi *AngularJS* servis *\$http*. Prilikom kreiranja *cache* objekat dobija *id*, *size* i sve ostale dodatne osobine koje podatak sadrži. Objekat sadrži skup metoda koje omogućavaju manipulaciju sa podacima i dobijanje željenih osobina podataka. Skup metoda čine: *info()*, *put(key, value)*, *get(key)*, *remove(key)*, *removeAll()*, *destroy()* metode.

Grafička korisnička sprega se sastoji od 6 pod modula:

- Pod modul za prijavljivanje korisnika u sistem
- Pod modul za kreiranje novog korisničkog naloga
- Pod modul za glavni prozora
- Pod modul za izmenu podataka korisnika
- Pod modul za kontrolu uređaja
- Pod modul za konfiguraciju kuće

Realizacija pod modula je ostvarena upotrebom grafičkih komponenti *Ionic framework-a*, uz podršku *Angular framework-a*. Grafička korisnička sprega realizuje vizuelnu prezentaciju stanja kompletnog sistema i upravlja kompletnim sistemom na osnovu unapred definisanih zahteva.

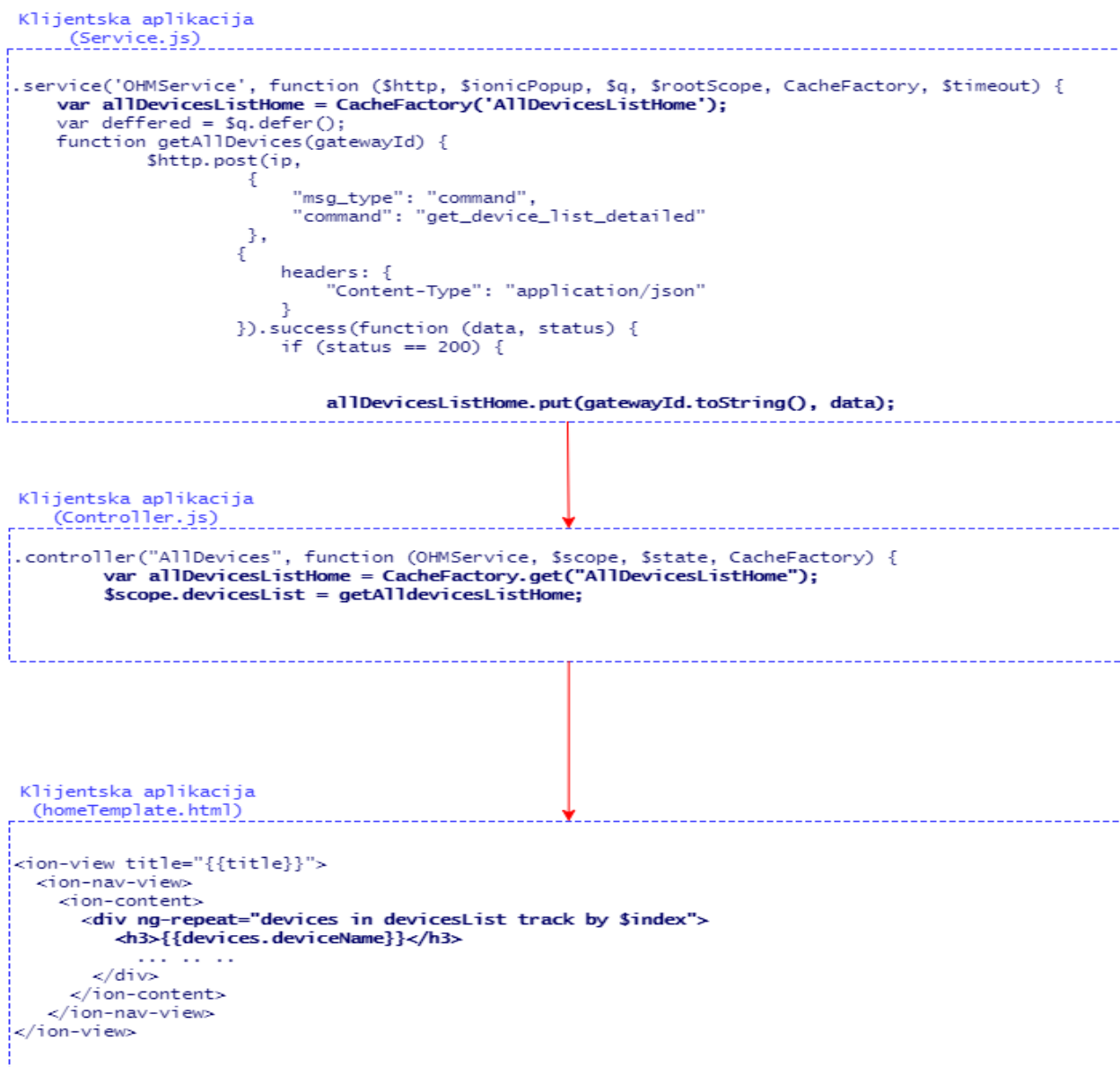
3.2.1 Pod modul za kontrolu uređaja

Pod modul za kontrolu uređaja namenjen je za prikaz dostupnih uređaja na centralnom uređaju. Sastoji se iz liste svih dostupnih uređaja zbog lakšeg manipulisanja. Omogućava korisniku da može dugo kliknuti na određeni uređaj i prikazuje prikaz osnovnih funkcija uređaja. Svaki uređaj ima detaljan prikaz njegovih osnovnih funkcija, koje nam prikazuju bitne informacije. Npr. bitna informacija kod *Dimmer* uređaja je trenutno stanje inteziteta svetlosti. Takođe potrebno je obezbediti da se na osnovu klika u prikazu uređaja korisnik usmeri na narednu stranicu, gde su prikazana napredna podešavanja tog uređaja.

3.2.1.1 Realizacija pod modula za kontrolu uređaja

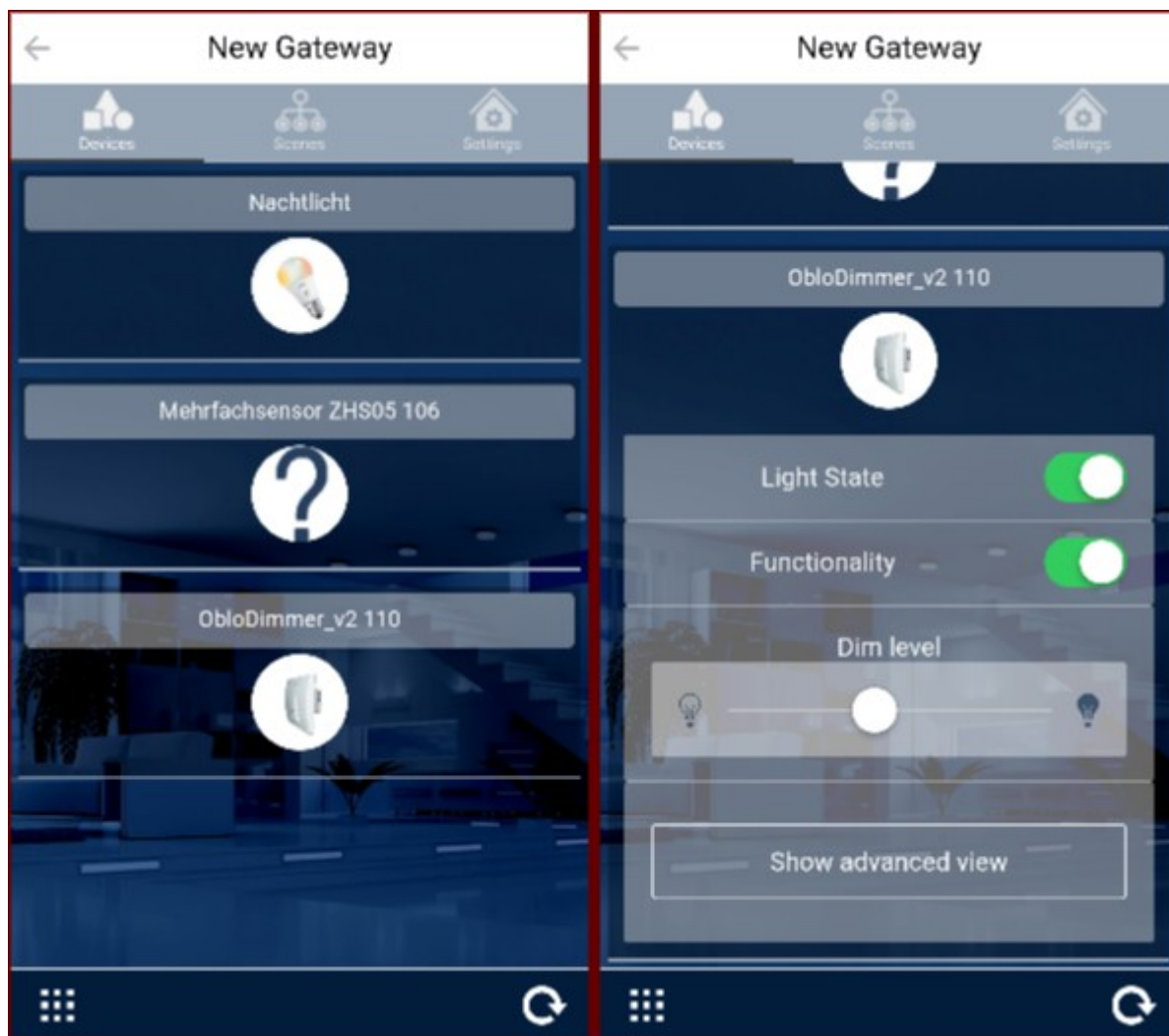
Interakcija između korisnika i aplikacije vrši se preko elementa aplikacije koji se naziva *Aktivnost* i koji ima kompleksan životni ciklus. Raspored elemenata na grafičkoj korisničkoj sprezi realizuje se u okviru *HTML* datoteke. Organizacija i realizacija *HTML* datoteka se vrši pomoću komponenti *Ionic framework-a*. Uz pomoć *AngularJS framework-a* *Ionic* pruža dinamičan prikaz i stil elemenata.

Lista svih dostupnih uređaja realizovana je upotrebom `<div>` elementa sa *ng-repeat* direktivom *AngularJS framework-a*. Detaljan prikaz funcionlanosti uređaja realizovan je upotrebom `<ul>` elementa sa *property-em class*, na osnovu kog *Ionic* prikazuje listu. Čuvanje svih podataka o uređajima u *cache* objekat realizuje se funkcijom *getAllDevices* u servisu. Lista uređaja se iščitavaju iz *cache* objekta u *\$scope* objekat, koji je takođe lista. Prosleđivanje podataka se vrši u kontroleru. Naredna slika 3.13 prikazuje prethodni proces:



Slika 3.13: Podaci za prikaz uređaja

Preko `$scope` objekta *HTML* datoteka ima pristup vrednostima podataka. Na taj način se prikaz uređaja popuni sa određenim podacima. Vizuelni prikaz uređaja i kontrole uređaja prikazan je na slici 3.14:



Slika 3.14: Izgled prikaza liste dostupnih uređaja na centralnom uređaju i kontrole odabranog dostupnog uređaja

Dugim pritiskom na ime uređaja korisniku se obezbeđuje prikaz osnovnih funkcija uređaja. Kao što prikazuje desni prozor. Na dugme *Show advanced view* realizovan je pritisak koji korisnika preusmerava na drugu stranicu koja sadrži sve funkcionalnosti odabranog uređaja. Prikazane dve stranice omogućavaju sledeće funkcije:

- Osvežavanje liste uređaja - vrši se slanjem komande *getDeviceListDetailde* na centralni uređaj pomoću modula za komunikaciju. Ako je komanda uspešno izvršena stari sadržaj *cache* objekta koji je čuvao podatke o uređajima se briše i popunjava se novim podacima. Funkcija *loadAllDevices* u *HomeController* kontroleru dodeljuje podatke iz *cache* objekta u *\$scope.deviceList* objekat.
- Promena parametra uređaja - vrši se slanjem komande *setDevicePropertyValue* na centralni uređaj pomoć komunikacionog modula. Komanda prima sledeće parametre: ***deviceId*** - jedinstveni identifikacioni broj uređaja, ***serviceName*** - određeni servis uređaja, ***propertyName*** - ime određene mogućnosti uređaja, ***propertyValue*** - vrednost na koju se postavlja određena mogućnost. Komanda se šalje preko funkcije *setPropertyValue* u *OHMSERVICE* servisu. Izgled ovog prozora je prikazan na slici 3.14.

4. Testiranje i rezultati

Kvalitet mobilne aplikacije ima presudan značaj kod krajnjih korisnika, čija očekivanja neprekidno rastu. Testiranje mobilnih aplikacija je ključno kod obezbeđivanja kvaliteta. Testiranje nije „jeftin“ proces i donosi dodatne izazove. Pažljivo odabrana strategija može višestruko vratiti uloženo u investiciju testiranja mobilnih aplikacija, smanjiti složenost i ukupno utrošeno vreme na razvoj. Nad realizovanim rešenjem izvršena su sledeća testiranja:

- testiranje funkcionalnosti
- testiranje prilikom opterećenja
- testiranje u realnim uslovima
- testiranje potrebnog napora prilikom razvoja

Testiranja su izvršena koristeći uređaj *Samsung GT-I9505* sa operativnim sistemom *Android* verzije *5.0.1*.

4.1 Testiranje funkcionalnosti

Testiranje funkcionalnosti predstavlja testiranje korisničkog interfejsa (*UI*). Testiranje korisničkog interfejsa se svodi na upotrebu aplikacije od strane korisnika. Testira se da li navedene karakteristike i funkcionalnosti iz prethodnog poglavlja izvršavaju prema očekivanjima. Proverava se aspekt iskorištenosti ekrana, dodir, da li je interakcija praćena odgovarajućom akcijom itd.

Glavna testiranja su fokusirana na funkcionalnost korisničke sprege i njen odziv na određene događaje, kao i otpornost aplikacije na greške prouzrokovane upotrebom aplikacije. Testiranja su izvršena manuelno. Opis određenih manualnih testiranja sa rezultatima prikazani su u tabeli 4.1:

Redni broj testa	Testa	Očekivani uslovi	Uspešnost
1.	<u>Opis:</u> Fizička promene stanja jednog od uređaja dostupnih na centralnom uređaju. <u>Početni uslovi:</u> Centralni uređaj i <i>Samsung</i> uređaj trebaju se povezati na istu lokalnu mrežu. Aplikacija se treba nalaziti u prozoru za prikaz dostupnih uređaja na centralnom uređaju sa prikazanim osnovnim funkcionalnostima uređaja ili u prozoru za prikaz svih funkcionalnosti. <u>Postupak:</u> Potrebno je fizički promeniti stanje uređaja preko prekidača na brznoj kontroli.	Promena stanja jednog od uređaja nakon fizičke promene stanja tog uređaja.	Uspešan.
2.	<u>Opis:</u> Dodavanje novog uređaja na centralni uređaj. <u>Početni uslovi:</u> Centralni uređaj i <i>Samsung</i> uređaj trebaju se povezati na istu lokalnu mrežu. Aplikacija se treba nalaziti u prozoru za prikaz dostupnih uređaja na centralnom uređaju. <u>Postupak:</u> Potrebno je kliknuti na dugme „Find Devices“. Nakon toga potrebno je ispoštovati određenu proceduru za dodavanje uređaja na centralni uređaj.	Da se u listi dostupnih uređaja pojavi novi uređaj. (slika 3.14)	Uspešan.

3.	<u>Opis:</u> Promena stanja uređaja koristeći mogućnost kontrole. <u>Početni uslovi:</u> Centralni uređaj i <i>Samsung</i> uređaj trebaju se povezati na istu lokalnu mrežu. Aplikacija se treba nalaziti u prozoru za prikaz dostupnih uređaja na centralnom uređaju sa prikazanim osnovnim funkcionalnostima uređaja ili u prozoru za prikaz svih funkcionalnosti određenog uređaja. <u>Postupak:</u> U pomenutim prozorima pritisnuti prekidač za promenu stanja uređaja.	Da se nakon klika na prekidač u prozoru upali/ugasi odgovarajući uređaj i promene stanja zavisnih uređaja. (slika 3.14)	Uspešan.
4.	<u>Opis:</u> Promena imena centralnog uređaja . <u>Početni uslovi:</u> Centralni uređaj i <i>Samsung</i> uređaj trebaju se povezati na istu lokalnu mrežu. Aplikacija se treba nalaziti u prozoru za prikaz liste dostupnih centralnih uređaja koji su povezani sa aplikacijom. <u>Postupak:</u> Potrebo je kliknuti na dugme „<i>Settings</i>“, a zatim na dugme „<i>Rename</i>“. Nakon unete nove vrednosti klinuti na dugme „<i>Save</i>“.	Da se nakon klika na dugme „ <i>Save</i> “ prikaze lista dostupnih centralnih uređaja sa novim imenom odgovarajućeg centralnog uređaja.	Uspešan.

5.	<u>Opis:</u> Dodavanje centralnog uređaja na korisnički nalog.	Da se dodati centralni uređaj pokaže u listi svih dostupnih centralnih uređaja za jednog korisnika.	Uspešan.
	<u>Početni uslovi:</u> Centralni uređaj i <i>Samsung</i> uređaj trebaju se povezati na istu lokalnu mrežu. Aplikacija se treba nalaziti u prozoru za dodavanje novog centralnog uređaja.		
	<u>Postupak:</u> Potrebo je uneti podatke za jedinstven bezbednosni broj, serijski broj i simboličko ime centralnog uređaja.		

Tabela 4.1: Manuelni testovi funkcionalnosti

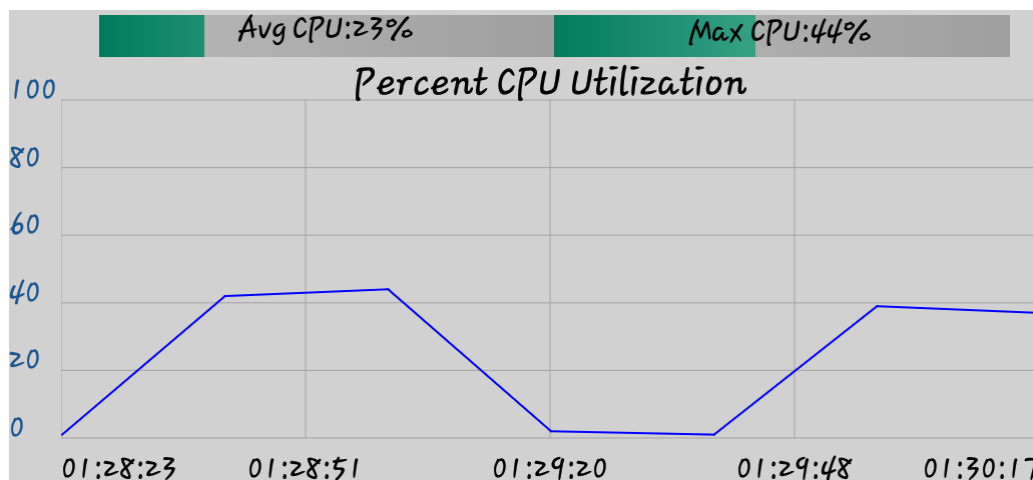
4.2 Testiranje prilikom opterećenja

Testovi prilikom opterećenja se odnose na merenja opterećenosti procesora i radne memorije uređaja tokom rada aplikacije. *Samsung* uređaj korišten za testiranje karakteriše procesor 1.89 GHz sa četiri jezgra i radna memorija od 1.78 GB. Uređaj se koristi za testiranje hibridne aplikacije (opisane u radu) i testiranje nativne aplikacije razvijene za *Android* platformu. Što omogućava poređenje ostavarnih rezultata testiranja hibridne i nativne aplikacije.

4.2.1 Testiranje iskorišćenosti procesora

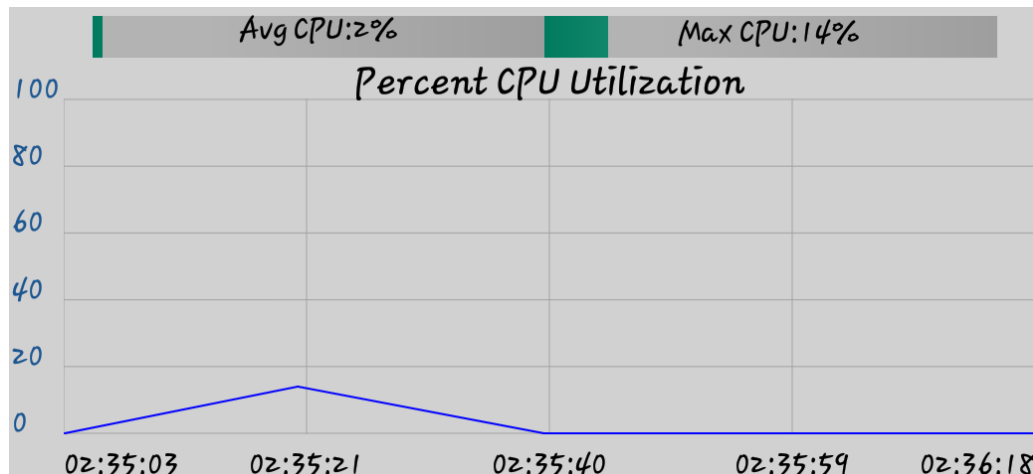
Testiranje je vršeno tokom korišćenja kompletnih aplikacija, odnosno korištene su sve dostupne mogućnosti koje nude aplikacije. Rezultati svake aplikacije prikazani su posebno preko grafičkog prikaza.

- Grafički prikaz rezultata hibridna aplikacije:



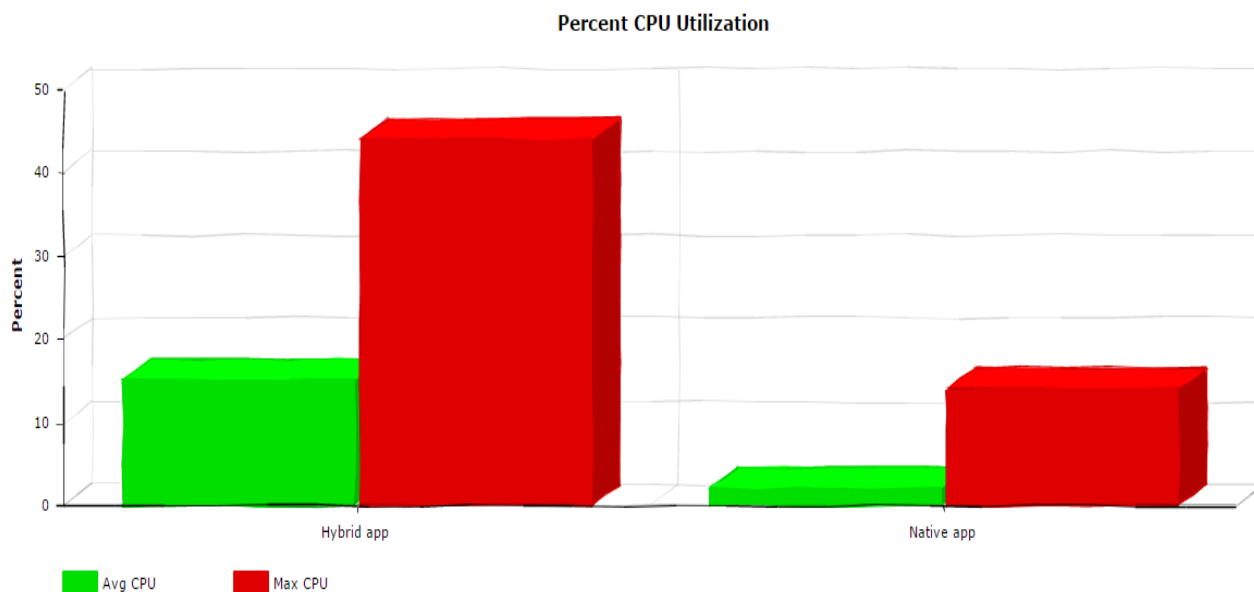
Slika 4.1: Grafički prikaz iskorištenosti procesora pri radu hibridne aplikacije

- Grafički prikaz rezultata *Android* native aplikacije:



Slika 4.2: Grafički prikaz iskorištenosti procesora pri radu native aplikacije

Na grafičkim prikazima treba primetiti da je prilikom rada *Android* native aplikacije opterećenje procesora mnogo manje u poređnju sa hibridnom aplikacijom. Rezultati su očekivani, jer native aplikacije su razvijene u cilju postizanja najboljih performansi za namenjenu platformu. Dok hibridne aplikacije iskorištavaju više resursa i očekuje se slabljenje performansi. Poređenje rezultata prosečne i maksimalne vrednosti prethodno pomenute dve aplikacije prikazano je na slici 4.3:

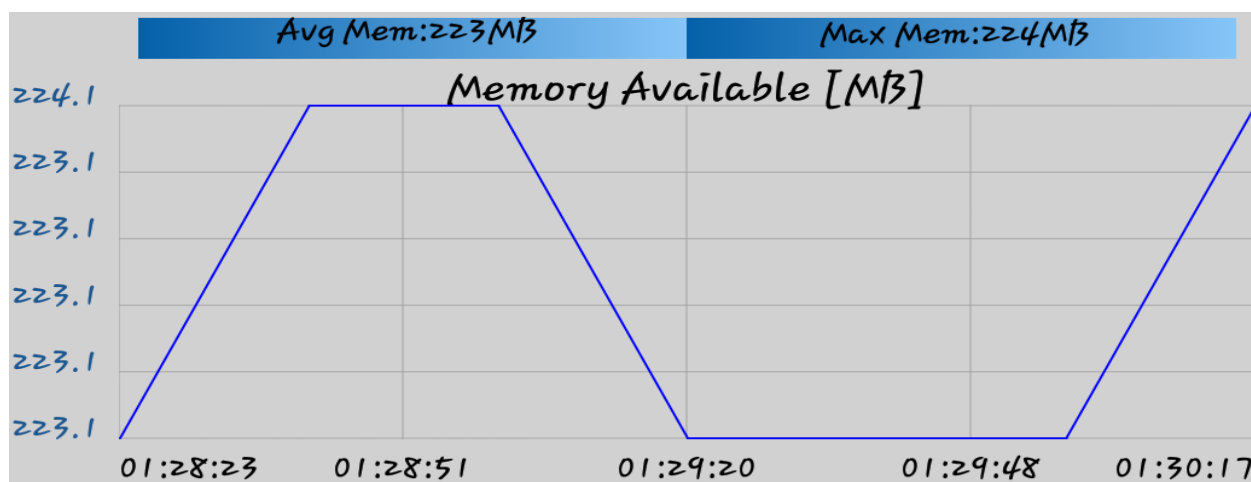


Slika 4.3: Poređenje ostvarenih rezultata hibridne i nativne aplikacije za testiranje opterećenja procesora

4.2.2 Testiranje zauzetosti radne memorije

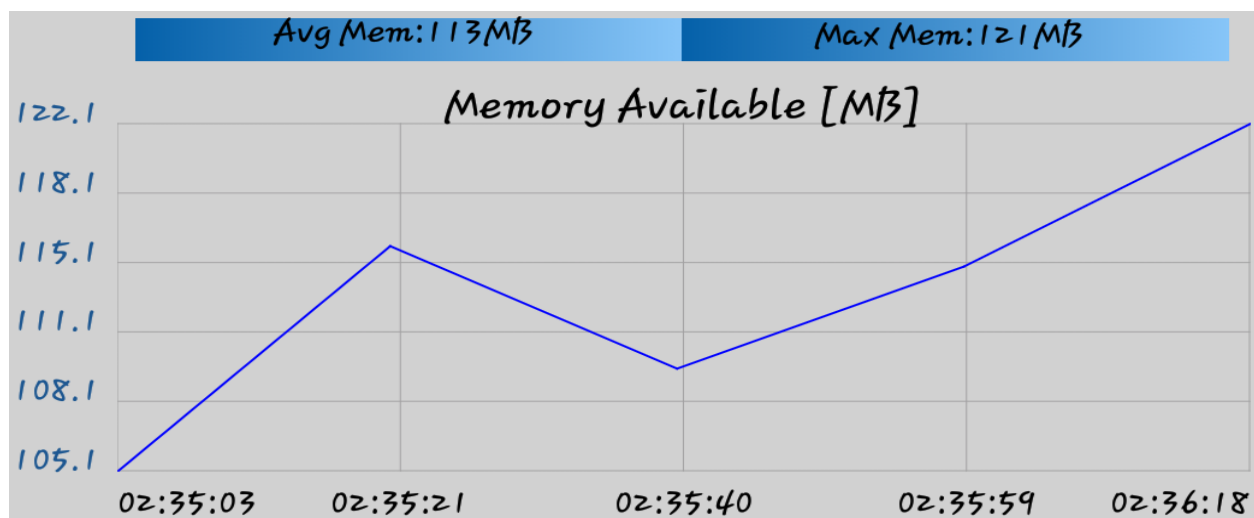
Mobilne aplikacije rade na malim fizičkim uređajima, zbog toga umerena količina upotrebe radne memorije predstavlja izazov. Testiranje je vršeno tokom korišćenja svih dostupnih mogućnosti aplikacije. Rezultati svake aplikacije prikazani su posebno preko grafičkog prikaza.

- Grafički prikaz rezultata hibridne aplikacije:



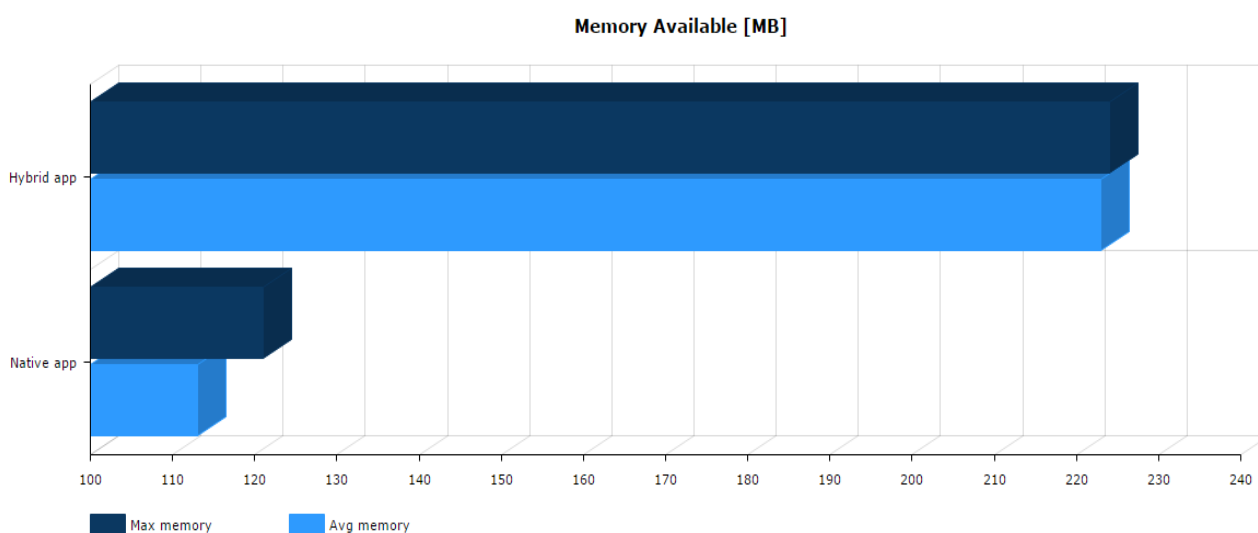
Slika 4.4: Grafički prikaz zauzetosti radne memorije pri radu hibridne aplikacije

- Grafički prikaz rezultata *Android* native aplikacije:



Slika 4.5: Grafički prikaz zauzetosti radne memorije pri radu native aplikacije

Grafički prikazi ukazuju na veću upotrebu radne memorije od strane hibridnih aplikacija. Rezultati koji to dokazuju su očekivani, jer hibridne aplikacije koriste *webview*, koji sam po sebi koristi značajnu količinu memorije.



Slika 4.6: Poređenje ostvarenih rezultata hibridne i native aplikacije za testiranje zauzetosti radne memorije

4.3 Testiranje u realnim uslovima

Testovi se odnose na to koliko će sistem brzo reagovati na određenu akciju pokrenutu od strane klijentske aplikacije. Kao primer je uzeta kontrola uređaja na centralnom uređaju. Takođe testirano je vreme potrebno za *start-up* aplikacija i vreme potrebno za tranziciju.

Vreme odziva sistema za kontrolu uređaja predstavlja trenutak kada je na klijentskoj aplikaciji kliknut prekidač do trenutka kada se svetlo upalilo. Merenje *start-up* vreme predstavlja vreme potrebno za prikaz prozora za prijavljivanje korisnika (*login*), a za tranziciju meri se vreme trajanja prelaska iz jedne stranice (prozora) u drugu. Tranzicija će imati minimalnu i maksimalnu vrednost. Minimalna vrednost se odnosi na stranice sa prostim sadržajem, a maksimalna na stranice sa kompleksnim sadržajem. Tabela 4.2 prikazuje rezultate:

Test	Vreme [ms] odziva za hibridne aplikacije	Vreme [ms] odziva za native aplikacije
Kontrola jednog uređaja	400 ms	400 ms
<i>Start-up</i> aplikacije	7800 ms	4400 ms
Tranzicija	min: 55 ms max: 503 ms	min: 40 ms max: 80 ms

Tabela 4.2: Prikaz testova odziva sistema

Rezultati prikazuju izjednačenost prilikom kontrole jednog uređaja na centralnom uređaju. Ali bolje rezultate ostvaruje nativna aplikacija za tranziciju i prilikom prvog startovanja aplikacije. Ostvareni rezultati su očekivani, jer hibridne aplikacije zahtevaju više vremena za učitavanje biblioteka koje koristi i za iscertavanje komponenti. Maksimalno vreme za tranziciju kod hibridnih aplikacija se povećava ukoliko naredna stranica sadrži kompleksan sadržaj ili su određene komponente animirane.

4.4 Testiranje potrebnog napora prilikom razvoja

Rezultati za potreban napor prilikom razvoja različitih aplikacija su prikupljeni na osnovu iskustva programera u razvoju nativnih i hibridnih aplikacija.

	Vreme potrebno za razvoj aplikacije (od početka - do kraja)
Nativna mobilna aplikacija	4 meseca samo za jednu platformu
Hibridna mobilna aplikacija	3 meseca za sve platforme

Tabela 4.3: Prikaz potrebnog vremena za razvoj aplikacije

Razvoj aplikacije za svaku platformu ili verziju platforme će uvesti dodatne zahteve za dizajnere i programere. Svaki dodatni zahtev kreira nove zahteve za testiranje. Testiranje se vrši za svaku verziju platforme. Ako aplikacija podržava tri mobilne platforme to prouzrokuje minimalno utrostručavanje napora potrebnog za razvoj aplikacije. Rezultati prikazuju da je prilikom razvoja hibridne aplikacije potreban manji napor (*effort*), što je bio i očekivani ishod ovog testiranja.

5. Zaključak

Zadatak rada bio je realizacija hibridne aplikacije za upravljanje pametnom kućom, koja ispunjava sve potrebe i zahteve prosežnog korisnika, sa ciljem doprinosa i napretka u razvoju mobilnih aplikacija. Evaluacija alata koji omogućava generisanje aplikacije iz jednog koda za više mobilnih platformi, odnosno rešavanje problema razvoja aplikacija za različite platforme je primarni cilj.

Realizacija aplikacije zahtevala je određene zadatke. Jedan od zadataka je omogućiti što jednostavniji izgled aplikacije, kako bi se omogućila laka i jednostavna svakodnevna upotreba aplikacije za korisnike svih uzrasta. Takođe i iskorištavanje novih internet tehnologije u cilju postizanja tečnijeg rada (optimizacije) aplikacije.

U sklopu aplikacije realizovane su funkcionalnosti:

- Rukovanje centralnim uređajem iz bilo kog dela pametne kuće gdje je dostupan pristup lokalnoj mreži
- Rukovanje centralnim uređajem sa bilo kog mesta preko *Stratus* udaljenog poslužioca
- Kontrola uređaja unutar pametne kuće

Mogući pravci razvoja i poboljšanja funkcionalnosti aplikacije su:

- Višejezična podrška na grafičkoj korisničkoj sprezi
- Rukovanje složenim radnjama unutar centralnog uređaja u vidu scena
- Kontrola grupe uređaja unutar pametne kuće

Aplikacija je testirana sistematično u realnom sistemu. Testovi koji su obavljeni su: testovi funkcionalnosti, testovi merenja performansi sistema, testovi odziva sistema. Sprovedena testiranja su pokazala da performanse aplikacije ne zavise samo od karakteristike mreže i od samog poslužioca. Dodeljivanje kompleksnijih funkcionalnosti hibridnim aplikacijama prouzrokuje slabljenje performansi. Rezultati sprovedenih testiranja pokazuju da je realizovana

aplikacija postigla željene mogućnosti, ali ne i željene karakteristike. Zaključuje se da inicijalno postavljani zahtevi nisu u potpunosti ispunjeni.

Nakon završenog razvoja i testiranja aplikacije mogu se primetiti prednosti i mane hibridnih aplikacije u odnosu na native aplikacije. Prednost je generisanje aplikacije za više platformi iz jednog koda, manji napor za realizaciju aplikacije. Najveći problem mobilnih aplikacija ovog tipa je optimizacija. Iz tog razloga hibridne aplikacije ostvaruju inferiornije rezultate performansama, u poređenju sa native aplikacijama. Potencijalne problemi mogu predstavljati dodatni zahtevi za dodatke (*3th party*[13]), kao npr. drajveri za IP kamere.

Performanse predstavljaju temelj kvalitetne aplikacije i obezbeđuju snažan *UX* (*user experience*). Aplikacija opisana u radu nije ostvarila pozitivne rezultate tokom testiranja, pre svega mislim na rezultate performansi. Zbog toga izabrao bih drugu tehnologiju za razvoj aplikacija sličnih opisanoj aplikaciji u radu.

Apache Cordova framework je pogodan za razvoj hibridnih aplikacija koje nisu veoma zahtevne i kompleksne. Za zahtevnije i kompleksnije aplikacije native aplikacije imaju prednost. Međutim *Apache Cordova* je ostvario jedan veliki korak i napredak u razvoju mobilne tehnologije.

6. Literatura

- [1] Daniel Burrus, founder and CEO of Burrus Research.
- [2] Dr. Ovidiu Vermesan, Dr. Peter Friess, „Internet of Things - From Research and Innovation to Market Deployment“.
- [3] Kevin Ashton, cofounder and executive director of the Auto-ID Center at MIT, 1999.
- [4] ZigBee Alliance official site specification ZigBee document 053474r13 version 1.1. , Acc.26.06.2015, <http://www.zigbee.org/>
- [5] ZW0301 Data Sheet
- [6] About VOCO, Acc.26.06.2015 <http://myvoco.com>
- [7] John M. Wargo, „Apache Cordova API Cookbook“, July 2014
- [8] Chris Smith, web developer, „Angular Basics by scripty (books)“
- [9] Arvind Ravulavaru, „Learning Ionic“, July 2015.
- [10] Visual Studio 2015, <https://www.visualstudio.com/docs/vs/overview>
- [11] Ivan Ristić, engineer, „OpenSSL Cookbook“, March 2015
- [12] Official documentation, page 15 <http://upnp.org/specs/arch/UPnP-archDeviceArchitecture-v1.1.pdf>
- [13] John M. Wargo, „Apache Cordova 4 Programming“, April 2015