

ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU



**IMPLEMENTACIJA I TESTIRANJE SIGURNE KLIJENT SERVER
KOMUNIKACIJE IZMEĐU SERVERA I ANDROID APLIKACIJE**

– Master rad –

Kandidat:

Ivana Arsenijević 2013/3214

Mentor:

doc. dr Zoran Čiča

Beograd, Jun 2015.

SADRŽAJ

SADRŽAJ.....	2
1. UVOD.....	3
2. SIGURNOST	4
2.1. ŠIFROVANJE.....	4
2.1.1. Algoritmi sa simetričnim ključem	4
2.1.2. Algoritmi sa asimetričnim ključem	5
2.1.3. Hash algoritmi	6
2.2. AUTENTIFIKACIJA.....	7
2.2.1. Sredstva autentifikacije.....	7
2.2.2. Šeme za utvrđivanje autentičnosti.....	7
2.2.3. Autentifikacija servera.....	7
2.2.4. Autentifikacija korisnika	8
2.3. SPECIFIKACIJA BANKAPP APLIKACIJE	9
2.3.1. Razvojno okruženje	9
2.3.2. Enkripcija i autentifikacija	9
2.3.3. Protokol razmene podataka	9
3. IMPLEMENTACIJA SERVERA.....	10
3.1. PROGRAMSKI JEZIK JAVA	10
3.1.1. JDK.....	10
3.2. RAZVOJNO OKRUŽENJE ECLIPSE.....	10
3.3. RAZVOJ SERVERA.....	11
3.3.1. ServerDB klasa	12
3.3.2. ConnectionHandler klasa	13
3.3.3. Worker klasa.....	13
3.3.4. DBHandler.....	15
3.3.5. Security klasa.....	17
4. IMPLEMENTACIJA KLIJENTA	22
4.1. ANDROID SDK	22
4.2. RAZVOJ KLIJENTA.....	23
4.2.1. AndroidManifest.xml.....	25
4.2.2. Implementacija grafičkog interfejsa	26
4.2.3. Implementacija funkcionalnosti klijenta	28
5. VERIFIKACIJA RADA APLIKACIJE	35
6. ZAKLJUČAK.....	42
LITERATURA.....	43
A.SKRAĆENICE.....	44

1. UVOD

Sa razvojem telekomunikacija i Internet tehnologija sve se više napreduje ka kreiranju multifunkcionalnih uređaja. S obzirom da je mobilni uređaj jedan od uređaja koje većina ljudi ima uvek sa sobom, postoji tendencija stalnog rasta broja dostupnih aplikacija, odnosno dodatnih usluga koje se mogu izvršiti sa mobilnog uređaja.

Predviđanja su da ćemo uskoro svoje ključeve i kartice moći da zamenimo odgovarajućim aplikacijama na mobilnom uređaju. S obzirom da je reč o poverljivim informacijama, javlja se pitanje sigurnosti ovakvih aplikacija.

U ovom radu ćemo se baviti problemom sigurnosti pri dizajnu i implementaciji jedne Android aplikacije. Pokušaćemo da ukažemo na eventualne pretnje od malicioznih trećih lica, potencijalne sigurnosne propuste, kao i na protokole zaštite poverljivih informacija.

U drugom poglavlju ćemo pokriti teorijsku osnovu koju je neophodno poznavati da bi se upustili u poduhvat kreiranja sigurne komunikacije. Reći ćemo par reči o najčešćim algoritmima šifrovanja, načinima autentifikacije, kao i kodovima koji se koriste za proveru ispravnosti prenesenih informacija.

Treće poglavlje će se baviti razvojem servera u programskom jeziku Java. Prvo ćemo dati kratak pregled programskog jezika, kao i alata koji će biti korišćeni za razvoj. Posebno ćemo naglasiti probleme pri implementaciji sa kojima smo se susretali, kao i eventualna drugačija rešenja od našeg.

Četvrto poglavlje treba prvo da predstavi Android platformu na kojoj ćemo razvijati svoju klijentsku aplikaciju, a zatim i samu klijentsku aplikaciju. Naglasićemo ograničenja koja donosi ova platforma, kao i načine na koje možemo da ih prevaziđemo.

U petom poglavlju ćemo testirati razvijeni softver i pokušati da ukažemo na potencijalna poboljšanja. Takođe ćemo predstaviti alate koje ćemo koristiti, kao i proces instalacije aplikacije. Potrudićemo se da istaknemo značaj testiranja i verifikacije kod ovakvih aplikacija.

U zaključku ćemo istaći prednosti i nedostatke razvijene aplikacije. Pokušaćemo da ukažemo na primene koje bi aplikacija mogla da ima. Takođe ćemo izneti prognoze budućnosti sigurnosnih sistema na osnovu urađenog istraživanja trenutnih trendova na tržištu.

2. SIGURNOST

Osnovni sigurnosni problemi koje VPN zaštita treba da zadovolji su:

- Tajnost - Da li neko nepozvan može da pristupi podacima?
- Autentičnost - Da li podaci zaista potiču od osobe od koje mislimo da potiču.
- Integritet - Da li su podaci koje smo primili modifikovani od strane trećeg lica?

Danas postoji veliki broj protokola koji nude svoje rešenje koje bi trebalo da odgovori na ova pitanja. Ne postoji optimalno rešenje. Zavisno od zahteva, potrebno je praviti kompromise.

2.1. Šifrovanje

Šifrovanje je proces kodiranja informacija sa ciljem zaštite podataka od osoba sa malicioznim namerama. Osnovni zadatak je da omogući komuniciranje dve osobe preko nesigurnog kanala, tako da treća osoba koja nadzire kanal ne može razumeti njihove poruke.

Poruka koja se šalje se naziva otvoreni tekst (*plaintext*). Otvoreni tekst se transformiše pomoću ključa i dobija se šifrat ili kriptogram.

Zavisno od vrste ključa razlikujemo:

- Algoritme sa simetričnim ključem
- Algoritme sa asimetričnim ključem

2.1.1. Algoritmi sa simetričnim ključem

Kod ovih algoritama isti ključ se koristi i za šifrovanje i za dešifrovanje dokumenata. Ključ se može dobiti samo upotrebom "grube sile", odnosno isprobavanjem raznih kombinacija. Ukoliko haker dođe u posjedstvo ključa, ima mogućnost da dešifruje, izmeni i ponovo šifruje podatke. Na taj način može da nanese veliku štetu neprimetno.

Takođe potreban je siguran kanal za prenos ključa od pošaljioaca do primaoca. Ovaj kanal takođe može biti hakovan i ključ ukraden.

Dolazi se do zaključka da je za optimalnu zaštitu potrebno povremeno menjati ključ. Ovo se osigurava protokolima za zaštitu kanala od kojih je najčešće korišćeni IPsec.

Najpoznatiji algoritmi sa simetričnim ključem su DES, AES i trostruki DES.

DES

DES (*Data Encryption Standard*) je razvijen od strane IBM. Američka vlada je 1977. godine prihvatila DES kao zvaničan standard za javne informacije. Algoritam je tako projektovan da se šifrovanje i dešifrovanje obavlja istim ključem.

Osnovna ideja je da se otvoreni tekst šifruje u blokovima od po 64-bita pomoću 56-bitnog ključa u 19 koraka, dajući na taj način 64-bitne blokove šifrovanog teksta. Prvi korak je transponovanje 64-bitnog osnovnog teksta bez upotrebe ključa, dok poslednji korak podrazumeva odgovarajuću inverziju ovog postupka. U pretposlednjem koraku se 32 bita na levom kraju zamenjuju sa 32 bita na desnom kraju. Preostalih 16 koraka su funkcionalno jednaki, ali vrednosti parametara

dobijaju korišćenjem različitih funkcija ključa. Pri dešifrovanju se prolaze isto koraci samo obrnutim redosledom.

DES je zasnovan na algoritmu Lucifer, koji je takođe razvio IBM, ali koji umesto 56-bitnog koristi 128-bitni ključ, što je bilo povod za razne rasprave o njemu. Navodno je ključ skraćen da bi NSA (National Security Agency) mogla da ga razbije, dok bi i dalje bio siguran za organizacije sa manjim budžetom.

Danas se više ne može smatrati sigurnim algoritmom i zbog male dužine ključa i zbog objavljenog niza algoritama za njegovo razbijanje tokom godina. Noviji algoritmi koji se koriste su AES i trostruki DES.

Trostruki DES

Ključ za DES je bio prekratak i samim tim je vremenom postao prevaziđen, pa je IBM smislio kako da ga produži trostrukim šifrovanjem. Umesto jednog koriste se dva ključa u tri koraka.

U prvom koraku osnovni tekst se šifrjuje algoritmom DES uz ključ K1. Zatim se u drugom koraku DES izvršava u režimu dešifrovanja uz ključ K2. A onda se opet šifrjuje kao i u prvom koraku ključem K1.

Sistem šifrovanja, dešifrovanja i ponovo šifrovanja je izabran zbog svoje kompatibilnosti sa postojećim DES sistemima sa jednim ključem. Ukoliko je potrebno komunicirati sa računarom koji koristi jednostruko šifrovanje jednostavno se izjednačava vrednost ključeva K1 i K2 i uspostavlja veza. Ovo je omogućilo postepeno uvođenje trostrukog šifrovanja.

AES

Kada je Nacionalni institut za standarde i tehnologiju (NIST) presudio da je potreban novi standard za šifrovanje, organizovano je javno nadmetanje. Na ovaj iznenađujući potez su se odlučili zbog nepoverenja među kriptanalitičarima prema NSA. Naime posle rasprava oko DES-a, pretpostavka bi bila da je NSA u novi algoritam ugradila „mala vrata“ sa namerom da čita šifrovane poruke. Stoga je jedan od uslova konkursa bio da ceo projekat mora biti javan, kao i algoritam.

Podneto je više ozbiljnih projekata i posle velikog broja konferencija izabran je Rijndael. Tvorci ovog algoritma su dva mlada Belgijanca - Joan Daemen i Vincent Rijmen.

Rijndael podržava ključeve i blokove veličina od 128 do 256 bitova u koracima po 32 bita, pri čemu se dužina ključa i bloka može birati nezavisno. Kao i kod DES-a koristi se supstituisanje i permutovanje i algoritam se sastoji iz većeg broja rundi. Broj rundi zavisi od veličine ključa i bloka, pa tako za 128-bitni ključ i 128-bitni blok je neophodno 10 rundi, dok je za 256-bitni ključ i blok potrebno 14 rundi. Algoritam je projektovan sa namerom da pored bezbednosti zadovolji i potrebe za brzom realizacijom šifrovanja.

AES je varijacija Rijndaela sa fiksnom dužinom bloka od 128 bita i dužinom ključa od 128 bita, 192 bita ili 256 bita.

2.1.2. Algoritmi sa asimetričnim ključem

Problem kod sistema sa simetričnim ključem je što se ključ mora distribuirati između korisnika nekom sigurnom vezom. Ukoliko bi se hakeri dokopali ključa mogli bi da čitaju i menjaju zaštićene informacije. Pri promeni ključa potrebno je ponovo slati svim korisnicima novi ključ i obezbediti da ga dobiju samo oni kojima je namenjen.

Imajući u vidu ove nedostatke Diffie i Hellman – istraživači sa Stanforda, došli su na ideju nove vrste kriptosistema gde će se koristiti različiti ključevi za šifrovanje i dešifrovanje. Algoritmi za šifrovanje E i dešifrovanje D bi morali da zadovoljavaju sledeće stavke:

- $D(E(P)) = P$
- D se teško izvodi iz E
- E se ne može provaliti napadom zasnovanim na šifrovanju osnovnog teksta

Ukoliko su ispunjeni ovi uslovi, ključ za šifrovanje može postati javan i to neće uticati na sigurnost sistema.

Dakle nema potrebe za sigurnim kanalom za distribuciju ključa, pošto u ovom slučaju tajni ključ nije poznat ni onome ko koristi javni ključ za šifrovanje ili dešifrovanje poruke u zavisnosti od primene algoritma sa asimetričnim ključem. Ukoliko se štiti tajnost, javnim ključem se šifrjuje, a tajnim dešifrjuje. Ukoliko se štiti autentičnost, tajnim ključem se šifrjuje, a javnim dešifrjuje.

Najpoznatiji algoritam sa asimetričnim ključem je RSA.

RSA

RSA algoritam se sastoji iz nekoliko koraka:

- Prvo je potrebno izabrati dva velika prosta broja p i q .
- Zatim se računaju brojevi n i z , tako da je $n = p * q$ i $y = (p - 1)(q - 1)$.
- Bira se e , tako da e i z nemaju zajedničkih faktora i $e < n$.
- Bira se d , tako da je $(e * d)^{mod Z} = 1$.
- Javni ključ je sada (n, e) , a tajni (n, d) .

Kada smo dobili javni i tajni ključ, može se izvršiti šifrovanje poruke m . Dakle potrebno je izračunati šifrat c kao:

$$c = m^e \text{ mod } n$$

Da bi se dešifrovala sekvenca treba izračunati:

$$m = c^d \text{ mod } n$$

Problem koji se javlja kod RSA je pronalazak velikog prostog broja. Trenutno napadima odolevaju 1024-bitne šifre, a preporučuje se korišćenje 2048-bitnih.

2.1.3. Heš algoritmi

U slučaju da je samo neopohodna provera identiteta, bez čuvanja tajnosti koristimo heš algoritme. Cilj je da se pomoću nekog jednostavnog mehanizma generiše sažetak poruke koji jednoznačno odgovara originalnoj poruci. Obrnut mehanizam nije moguć, odnosno nije moguće iz sažetka poruke rekonstruisati originalnu poruku. Dakle reč je o jednosmernom šifrovanju.

Dobijanje sažetka poruke iz osnovnog teksta mnogo je brže od šifrovanja tog teksta sa javnim ključem, pa se ovaj mehanizam često koristi za ubrzavanje digitalnog potpisa.

Najčešće korišćeni algoritmi su MD5 i SHA-2.

2.2. Autentifikacija

Provera identiteta, tj. autentifikacija je metoda kojom se utvrđuje da li je osoba sa kojom komuniciramo zaista osoba za koju se izdaje. Ova tehnika je neophodna da bi se izbegli napadi od strane „čoveka u sredini”. Reč je o slučaju u kojem napadač prima poruke, kopira ih ili menja i zatim ih prosleđuje strani kojoj su namenjene. Obe strane ostaju nesvesne da su podaci menjani i da ih neko prisluškuje.

Da bi se ovo izbeglo pre slanja bilo kakvih poverljivih podataka neophodno je da izvršiti autentifikaciju. Postoji više načina da se izvrši ova provera.

2.2.1. Sredstva autentifikacije

Mnoge metode mogu da se koriste za zadovoljavanje autentifikacije kao što su unos lozinke, čitanje RFID identifikacije kartice, pritisak prsta na skener gde se otisak nalazi u bazi podataka ili neki drugi podatak koji se može koristiti za identifikaciju. Različita sredstva autentifikacije se mogu klasifikovati na sledeći način :

- znanje nečega: lozinke .
- posedovanjem nečega: smart kartice, OTP (*One time password*) tokeni, fizički tasteri.
- fizička karakteristika: statička biometrija poput otiska prsta, glasa, mrežnjače.
- karakteristike ponašanja: dinamična biometrija kao rukopis, kucanje ili ritam hodanja.

2.2.2. Šeme za utvrđivanje autentičnosti

Između različitih bezbednosnih mehanizama moraju postojati kompromisi. Ukoliko se odlučimo za digitalni potpis koji je najjači način pružanja autentifikacije, moramo biti svesni da ćemo proći skuplje u odnosu na jednostavno korišćenje lozinke. Klasičan primer gde je potrebna autentifikacija na visokom nivou su banke. Stoga ćemo navesti neke opcije za autentifikaciju koje se koriste u bankama:

- Jednokratna lozinka preko SMS - server banke šalje šifru na mobilni telefon koji je prethodno registrovan. Korisnik dakle dobija šifru na svoj mobilni, a zatim je unosi pri logovanju na veb servis. Pretpostavlja se da je trenutni korisnik mobilnog telefona zapravo korisnik kome je namenjena šifra, što ne mora biti tačno.
- Jednokratna lozinka generisana na osnovu tokena - korisnik koristi token koji je prethodno dobio od banke da generiše jednokratnu lozinku, a zatim ga unosi u veb stranicu. Ponovo se uvodi slična pretpostavka da je osoba koja poseduje token, ona kojoj je namenjen.
- Jednokratna lozinka generisana od strane aplikacije - slučaj je sličan prethodnom, samo sa tom razlikom što je generator lozinke aplikacija, umesto tokena.
- Digitalni potpis - da bi potvrdio svoj identitet server šalje klijentu digitalni sertifikat. Reč je o digitalno potpisanom dokumentu koji povezuje podatke o identitetu sa javnim ključem koji se koristi za enkripciju. Digitalni sertifikat je izdat i digitalno potpisan od strane sertifikacionog tela kome veruju kome veruju svi učesnici u komunikaciji.

2.2.3. Autentifikacija servera

Autentifikacija servera sa zasniva na sistemima sa javnim ključem (PKI – *Public Key Infrastructure*). Da bi potvrdio svoj identitet server šalje klijentu digitalni sertifikat. Reč je o digitalno

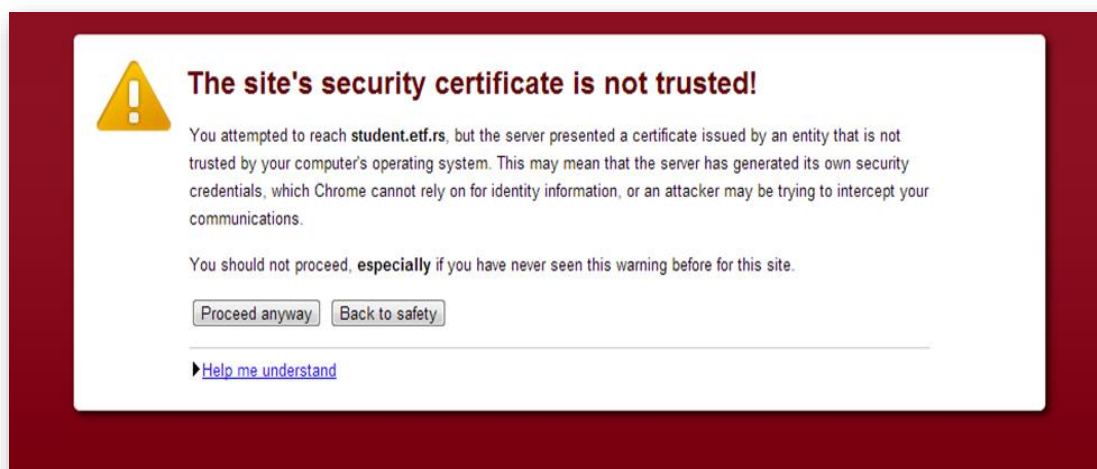
potpisanom dokumentu koji povezuje podatke o identitetu sa javnim ključem koji se koristi za enkripciju. Digitalni sertifikat je izdat i digitalno potpisan od strane sertifikacionog tela kome veruju svi učesnici u komunikaciji.

Najčešći format digitalnog sertifikata je opisan ITU-T X.509 standardom. Najčešće sadrži:

- Podatke o vlasniku sertifikata
- Javni ključ vlasnika sertifikata
- Period važenja sertifikata
- Ime izdavača (odnosno sertifikacionog tela)
- Serijski broj sertifikata
- Digitalni potpis izdavača

Zavisno od toga ko je sertifikaciono telo koje digitalno potpisuje sertifikat imamo samopotpisane i kvalifikovane sertifikate.

Samopotpisane sertifikate može da izda svako, pa tako i sam korisnik. Ovakvi sertifikati se koriste interno i baziraju se na poverenju. U slučaju da pokušavamo da pristupimo nekom sajtu koji je samopotpisan, naš brauzer će nas upozoriti da mu sertifikat nije poznat i da ulazimo na sopstveni rizik.



Slika 2.2.3.1 – Primer upozorenja na samopotpisani sertifikat u Google Chrome-u

Kvalifikovane digitalne sertifikate izdaje sertifikaciono telo koje mora da ispunjava određene zakonske uslove i da ima dozvolu za rad.

2.2.4. Autentifikacija korisnika

Kao i kod servera i kod korisnika se mogu koristiti digitalni sertifikati tako što će sertifikaciono telo podeliti svim korisnicima X.509 sertifikate sa javnim ključevima. Ovo je u većini slučajeva previše komplikovano i ne koristi se često.

Najjednostavniji i najčešći način autentifikacije je korišćenjem korisničkog imena i šifre. Ovo je takođe i nesiguran metod pošto šifra može biti i pogođena.

Umesto da posao proveravanja šifara radi sam server, moguće je unete podatke poslati RADIUS (*Remote Access Dial In User Service*) Serveru koji omogućava centralizovanu autentifikaciju. RADIUS je klijent/server protokol koji radi na aplikacionom sloju koristeći UDP saobraćaj. Ovaj server raspolaže sa više podataka o samom korisniku i njegovim privilegijama u okviru mreže.

2.3. Specifikacija BankApp aplikacije

Nakon što smo opisali neki osnovne tehnike zaštite informacije, u ovom poglavlju ćemo predstaviti specifikaciju aplikacije koju planiramo da implementiramo. Glavne tehničke karakteristike koje se tiču zaštite podataka i algoritama koje ćemo koristiti su već objašnjeni u prethodnom tekstu teze, tako da ćemo se prvenstveno fokusirati na procese koji će se odvijati između klijenta i servera.

BankApp je jednostavna aplikacija koja dozvoljava klijentu da proveri stanje svog bankarskog računa pomoću Android aplikacije na telefonu. Da bi ovakva aplikacija mogla da se distribuira korisnicima, neophodno je uvesti određene mere zaštite informacija koje se prenose, kao i autentifikaciju korisnika.

2.3.1. Razvojno okruženje

Aplikacija će biti realizovana korišćenjem programskog jezika Java. Kao razvojno okruženje koristićemo Eclipse, zajedno sa softverskim alatom za razvoj Android aplikacija Android SDK (*software development kit*). Podaci će biće skladišteni u MySQL bazi.

2.3.2. Enkripcija i autentifikacija

Da bi postigli optimalno rešenje u vidu brzine enkripcije podataka i jednostavnosti distribucije ključa, odlučili smo se za hibridno rešenje, odnosno kombinovanje dva algoritma enkripcije. Koristićemo AES algoritam sa 256-bitnim simetričnim ključem i RSA algoritam sa 2048-bitnim ključem.

AES algoritam će biti korišćen za enkripciju samih podataka koji će se prenositi između klijenta i servera. RSA algoritam će imati ulogu samo u enkripciji i prenosu samog AES ključa. Server će generisati privatni i javni RSA ključ i zatim proslediti klijentu javni ključ. Klijent će generisati svoj simetrični AES ključ, a zatim ga šifrovati dobijenim RSA javnim ključem i proslediti serveru. Svi podaci koje će potom server proslediti klijentu, će biti šifrovani AES protokolom, odnosno dobijenim simetričnim ključem.

Što se tiče autentifikacije klijenta koristićemo najčešće i najjednostavnije rešenje koje je danas u primeni, dakle PAP (*Password Authentication Protocol*) protokol. Drugim rečima klijent će svoj identitet potvrđivati pomoću korisničkog imena i šifre.

2.3.3. Protokol razmene podataka

Poruke će biti realizovane pomoću datagram soketa, odnosno komunikacija će se zasnivati na UDP protokolu.

3. IMPLEMENTACIJA SERVERA

U ovom poglavlju ćemo objasniti kod servera. Prvo ćemo se osvrnuti na programski jezik u kojem je server razvijan, kao i na alate koji su korišćeni. Zatim ćemo pokušati da približimo i kod samog programa sa posebnim akcentom stavljenim na klasu koja je zadužena za enkripciju.

3.1. Programski jezik Java

Java je objektno orijentisan programski jezik, nastao po uzoru na C++. Za razliku od C++ koji se donekle smatra hibridnim jezikom, Java je potpuno objektno orijentisan programski jezik. Sintaksa je slična programskom jeziku C, od koga su nastali i C++ i Java.

Java pokušava da reši neke probleme koji se javljaju pri korišćenju programskih jezika C ili C++. Tu se pre svega misli na sigurnost i portabilnost, zbog koje je Java danas jedan od najčešće korišćenih programskih jezika.

Način na koji je obezbeđena u isto vreme i portabilnost i sigurnost koda je prevođenje Java koda pomoću Java kompajlera u bajtkod, umesto u izvršni. Bajtkod se zatim pokreće, odnosno interpretira pomoću JVM (*Java Virtual Machine*). Sa obzirom da JVM kontroliše tok izvršavanja programa, postiže se određena sigurnost od širenja malicioznih programa izvan sistema. Takođe Java program se može izvršiti na svakom uređaju koji ima JVM.

3.1.1. JDK

JDK (*Java Development Kit*) je set alata za razvoj, debugovanje i nadgledanje Java aplikacija. U okviru JDK se nalazi JRE (*Java Runtime Environment*), koje opet sadrži implementaciju JVM, zajedno sa standardnim Java bibliotekama.

Da bi na uređaju bilo moguće razvijati Java aplikaciju, prvo je potrebno instalirati aktuelnu verziju JDK. Najnovija verzija JDK se može skinuti sa Oracle sajta besplatno:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

U vreme nastajanja ovog rada najnovija verzija je JDK 8. Ipak zbog kompatibilnosti sa Android uređajem mi ćemo koristiti prethodnu verziju JDK 7.

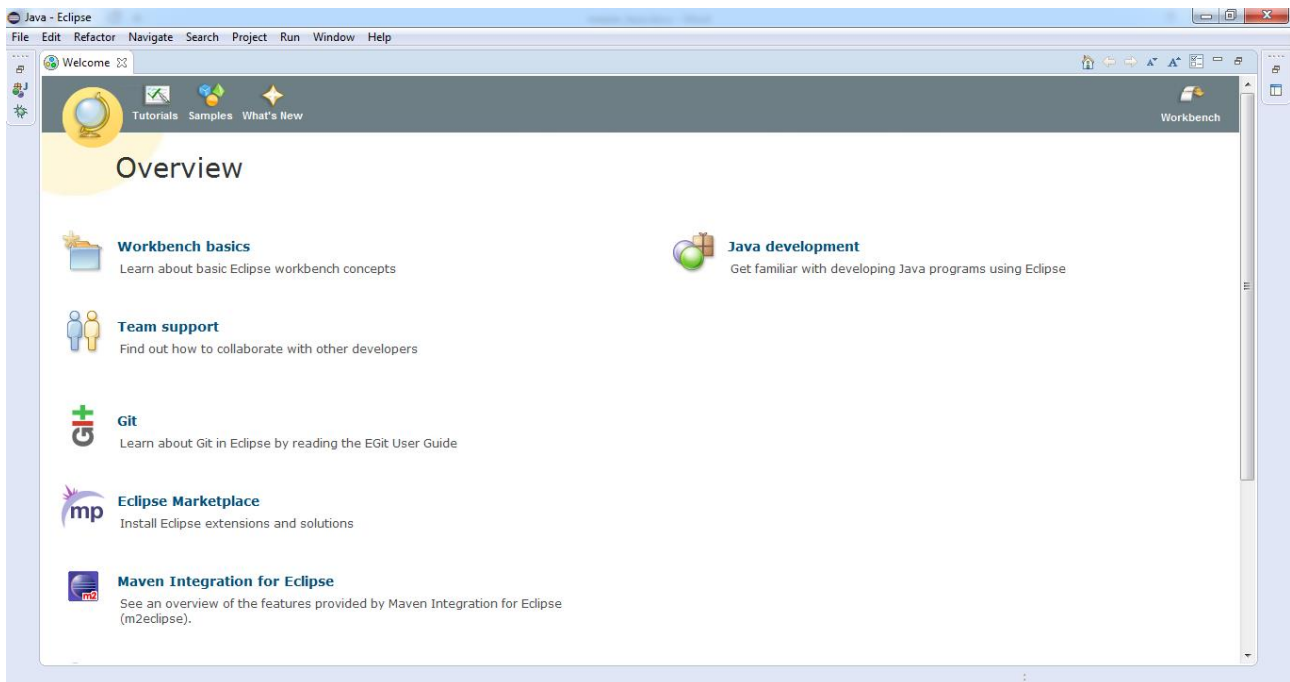
3.2. Razvojno okruženje Eclipse

Eclipse je jedno od najpoznatijih open sors razvojnih okruženja, koje poseduje dodatke za razvoj u većini programskih jezika. Sastoji se od osnovnog radnog okruženja, a zavisno od instaliranih dodataka, moguća je nadograditi osnovnu funkciju tako da zadovoljava potrebe programera za dodatnim alatima za modelovanje, testiranje, debugovanje itd. Dodatak koji ćemo koristiti u ovom radu, Android SDK je jedan od najčešće korišćenih.

Eclipse se može preuzeti besplatno sa sledeće stranice:

<https://eclipse.org/downloads/>

Zavisno od programskog jezika u kojem će se programirati, potrebno je izabrati distribuciju. Instaliranje podrazumeva samo raspakivanje Eclipse foldera i zatim pokretanje izvršnog fajla. Na slici 3.2.1 je prikazano osnovno radno okruženje pri prvom pokretanju aplikacije.



Slika 3.2.1 – Eclipse razvojno okruženje

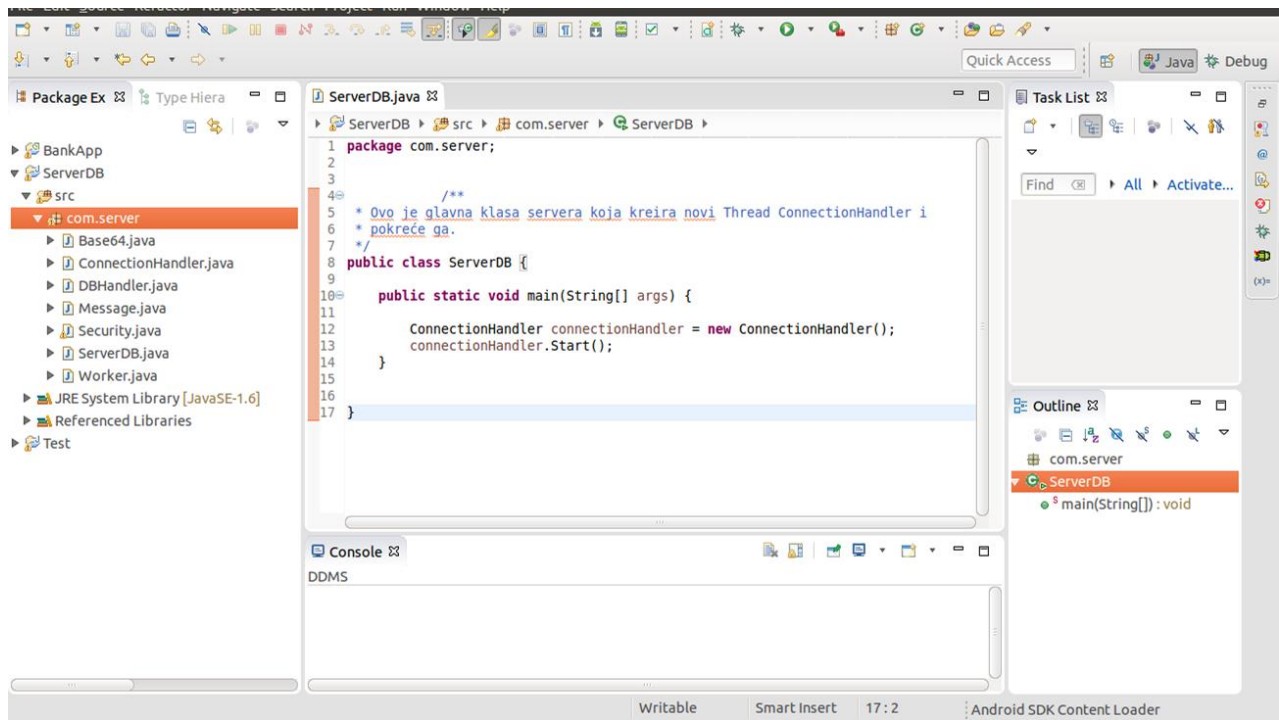
3.3. Razvoj Servera

Kao što je ranije pomenuto server je pisan u programskom jeziku Java. Reč je o objektno orijentisanom programskom jeziku, što znači da ćemo ponašanje servera implementirati korišćenjem različitih klasa, odnosno pozivanjem njihovih objekata. Što se tiče same enkripcije, Java ima već napisane biblioteke koje se bave ovom problematikom, tako da same algoritme enkripcije nećemo pisati, već ćemo koristiti gotova rešenja.

Ceo server se sastoji od šest klasa koje ćemo detaljno razmotriti u nastavku teze:

- ServerDB.java
- ConnectionHandler.java
- Worker.java
- DBHandler.java
- Security.java
- Message.java

Na slici 3.3.1 možemo videti izgled razvojnog okruženja Eclipse, sa otvorenim fajlom `ServerDB.java` u editoru.



Slika 3.3.1 – Projekat `ServerDB` u Eclipse razvojnem okruženju

Sa leve strane možemo da vidimo otvoren projekat **ServerDB**. Svaki projekat mora da sadrži **src** (source) folder u koji će se biti smeštene klase koje ćemo pozivati. Kao što možemo da vidimo na slici 3.3.1 u okviru **src** foldera se nalazi **com.server** paket (*package*). Pošto je Java veliki programski jezik u okviru kojega se nalazi veliki broj klasa, potrebno je na neki način ih klasifikovati, odnosno odvojiti. Za to nam služe paketi. U našem slučaju smo mogli da koristimo i **default.package**, pošto ne planiramo distribuciju koda, ali smo težili da se držimo dobre prakse u okviru Jave.

3.3.1. *ServerDB* klasa

Počecemo od glavne klase koja je ujedno i najjednostavnija, dakle sadrži samo dve linije koda u svojoj **main** metodi.

```
public class ServerDB {  
  
    public static void main(String[] args) {  
  
        ConnectionHandler connectionHandler = new ConnectionHandler();  
        connectionHandler.Start();  
    }  
  
}
```

Sve što klasa **ServerDB** radi je da kreira novu nit klase **ConnectionHandler** i zatim je pokreće. Ono što je bitno ovde pomenuti je da kada pokrenemo program **ServerDB**, on prvo pronalazi klasu koja se naziva kao program, odnosno klasu **ServerDB**. Kada je pronasao klasu, traži njenu **main** metodu i izvršava je. Naš program je kreiran tako da se ovim pokreće nova nit koja će dalje raditi sa podacima.

3.3.2. *ConnectionHandler* klasa

Objekat klase **ConnectionHandler** je nit koja prihvata konekcije od klijenta i kreira novu instancu klase **Worker** za dalje obavljanje zahteva od strane klijenta u okviru metode **Start()**. Dakle server otvara novi soket na 4444 portu i čeka na zahteve za konekciju od klijenta.

```
public void Start()
{

    try {
        serverSocket = new ServerSocket(4444);

        while(true)
        {

            Socket socket = serverSocket.accept();
            Worker worker = new Worker(socket);
            workerList.add(worker);

        }

    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Po uspostavljanju konekcije kreira se novi objekat tipa **Worker** i njemu se prosledjuje otvorena konekcija, odnosno soket.

Pošto u programskom jeziku Java svi objekti na koje ne postoje reference, budu uklonjeni od strane *Garbage Collector* mehanizma, morali smo da kreiramo listu koja će sadržati reference na sve **Worker** instance.

```
private List<Worker> workerList = new LinkedList<Worker>();
```

Dakle svaki put kada kreiramo novi objekat klase **Worker**, referencu na njega dodajemo u listu.

3.3.3. *Worker* klasa

Kada kreiramo novi objekat klase **Worker**, prvo se poziva posebna metoda koja se naziva konstruktor. Ovo je posebna metoda koja služi da se inicijalizuje vrednost objekta kada se on kreira. Treba napomenuti da ukoliko konstruktor nije definisan, klasa koristi difolt konstruktor koji

inicijalizuje sve vrednosti na nulu.

```
public Worker(Socket soc) {  
  
    socket = soc;  
  
    thread = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            Run();  
        }  
    });  
    thread.start();  
}
```

Konstruktor se prepoznaje po tome što nosi isto ime kao i klasa. Konstruktor u našem slučaju definiše novu nit tipa **Runnable**, koju zatim i pokreće. To znači da svaki put kada **ConnectionHandler** kreira objekat tipa **Worker**, stvara se nova nit na kojoj se dalje izvršava program, odnosno uslužuje klijent.

S obzirom na prethodno lako je zaključiti da se u okviru ove klase izvršava najveći deo posla koji server odrađuje.

Pored glavne metode **Run()**, **Worker** sadrži i dve pomoćne metode **SendMessage(Message msg)** i **ReceiveMessage()**. Kao što i samo ime kaže, prva metoda se koristi za slanje poruka i njoj prosledjujemo objekat tipa **Message**, koji je neophodno proslediti klijentu. Druga metoda služi za prijem poruka poslatih od strane klijenta.

Najbitniji deo koda koji podrazumeva implementaciju protokola komunikacije između klijenta i servera smešten je u metodu **Run()**. Dakle u okviru ove metode se primaju, a zatim i procesuiraju poruke primljene od klijenta. Rezultat se prosleđuje nazad klijentu. Na sledećem isečku koda ćemo objasniti kako je implementiran protokol. Reč je o početnoj razmeni poruka.

```
while (true)  
{  
    Message message = RecieveMessage();
```

Na samom početku koda vidimo da se ceo protokol komunikacije nalazi u okviru beskonačne petlje **while(true)**. Ovo je urađeno jer želimo da server stalno prima poruke naredbe, a zatim zavisno od toga koju je naredbu dobio, prelazi na odgovarajući **if** blok. Kada je komunikacija između klijenta i servera završena, server će dobiti naredbu **End**, koja će značiti da treba da iskoči iz petlje pomoću naredbe **break**.

```
if (message.message.equals("End"))  
{  
    System.out.print("End OK\n");  
    SendMessage(new Message("End OK"));  
  
    break;  
}
```

Server će proslediti klijentu odgovor na naredbu, kojom mu daje do znanja da je uspešno primio naredbu porukom **End OK**, a zatim izaći iz petlje. Razmotrimo još i početak komunikacije.

```

if (message.message.equals ("Start"))
{
    System.out.print ("Start OK\n");
    SendMessage (new Message ("Start OK"));

    SendMessage (new Message (security.getPublicKey()));
}

```

Server će po prijemu naredbe **Start**, odgovoriti sa **Start OK**, što obaveštava klijenta da je server uspešno primio naredbu. Server će odmah zatim poslati i svoj javni ključ, kojim klijent treba da šifruje simetrični ključ. Ovdje na scenu stupa **Security** klasa.

Pored **Security** klase, server u okviru **Run()** metode poziva i **DBHandler** klasu. Primer toga je proveravanje login podataka u bazi.

```

boolean logInSuccessful = DBHandler.LogInProcedure(username, password);

if(logInSuccessful)
{
    SendMessage (new Message ("Log in OK"));
    System.out.print ("Log in OK\n");
}
else
{
    SendMessage (new Message ("Log in FAIL\n"));
}

```

Zavisno od vrednosti koju promenljiva **logInSuccessful** dobije po pozivanju metode klase **DBHandler**, server će vratiti klijentu poruku o uspešnoj, odnosno neuspešnoj autentifikaciji.

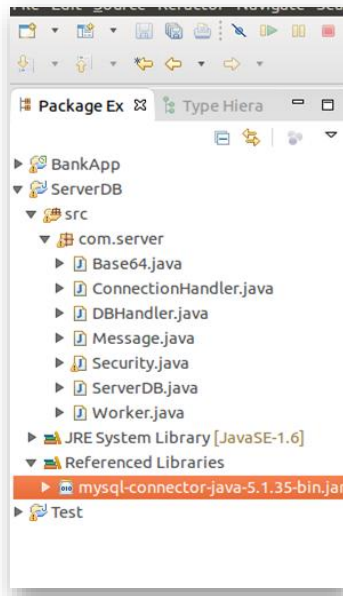
3.3.4. DBHandler

DBHandler klasa sadrži dve metode **LogInProcedure (String username, String password)** i **SendData (String username, String password)**. Obe metode se pozivaju iz **Worker** niti, zavisno od toga da li je potrebno samo potvrditi uspešnu autentifikaciju ili i proslediti informacije vezane za korisnika.

U obe metode potrebno je da se prvo server konektuje na bazu. Ranije je već pomenuto da je baza koju koristimo MySQL. Da bi mogli da pristupamo bazi iz našeg programa, potrebno je implementirati JDBC (*Java Database Connectivity*) drajv, koji obezbedjuje metode za povezivanje na bazu, odnosno slanje i čitanje upita. Koristićemo **Connector/J 5.1.35**, koji se može preuzeti besplatno:

<http://dev.mysql.com/downloads/connector/j/>

Sama implementacija drajva podrazumeva da ćemo odgovarajući .jar fajl dodati projektu.



Slika 3.3.4.1 – J connector unutar Eclipse projekta

Fajl se dodaje tako što kliknemo desnim tasterom na ime projekta i iz padajućeg menija izaberemo

Properties -> Java Build Path -> Libraries -> Add External JARs. Otvoriće nam se prozor u kojem možemo da izaberemo jar fajl koji ćemo da dodamo u projekat. Ukoliko je dodavanje drajva izvršeno uspešno moći ćemo da vidimo .jar fajl u okviru projekta kao na slici 3.3.4.1.

Pošto smo dodali drajv za JDBC, možemo koristiti gotove metode za povezivanje na bazu. U narednom delu koda, kreiramo konekciju i izvršavamo određeni upit (*query*).

```
//povezivanje na MySQL server
dbCon = DriverManager.getConnection(dbURL, dbUsername, dbPassword);
System.out.println("Connected\n");

//priprema upita
stmt = dbCon.prepareStatement(query);

//izvršavanje upita
rs = stmt.executeQuery(query);
```

Za povezivanje na bazu nam je potrebno da znamo URL (*Uniform Resource Locator*) baze na koju se povezujemo (**dbURL**), kao i korisničko ime (**dbUsername**) i šifru korisnika (**dbPassword**) koji ima dozvoljen pristup bazi koja nam je potrebna.

```
static String dbURL = "jdbc:mysql://localhost:3306/mybank";
static String dbUsername = "root";
static String dbPassword = "admin";
```

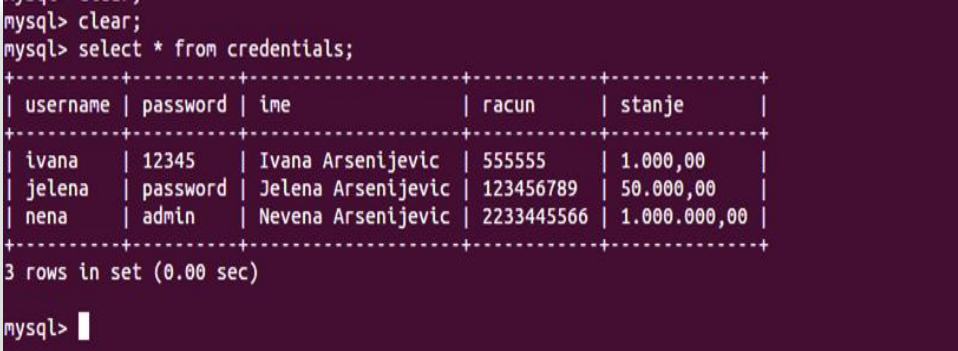
Iznad vidimo kako su definisane ove vrednosti u našem programu. Koristili smo **root**

korisnika, koji ima dozvole za čitanje i pisanje u bazi. Ukoliko želimo da zaštitimo neke podatke u okviru baze i ne dozvolimo korisniku da im pristupi, potrebno je kreirati novog korisnika i dodeliti mu samo određene privilegije tj. dozvole.

Na kraju ćemo još i razmotriti primer SQL upita u bazu. Posmatraćemo primer koji traži korisnika naše aplikacije prema zadatim vrednostima korisničkog imena i šifre, a zatim iščitava njegove podatke iz banke.

```
String query = "select * from credentials where password = '" +  
password + "' AND username = '" + username + "'";
```

Tabela **credentials** sadrži kolone: username, password, ime, račun, stanje.



```
mysql> clear;  
mysql> select * from credentials;  
+-----+-----+-----+-----+-----+  
| username | password | ime | racun | stanje |  
+-----+-----+-----+-----+-----+  
| ivana | 12345 | Ivana Arsenijevic | 555555 | 1.000,00 |  
| jelena | password | Jelena Arsenijevic | 123456789 | 50.000,00 |  
| nena | admin | Nevena Arsenijevic | 2233445566 | 1.000.000,00 |  
+-----+-----+-----+-----+-----+  
3 rows in set (0.00 sec)  
  
mysql> |
```

Slika 3.3.4.2 – Tabela *credentials* iz baze podataka

Na slici 3.3.4.2 možemo videti izgled **credentials** tabele, zajedno sa unetim vrednostima za testiranje. Naredbom "select * from credentials" iščitavamo sve vrednosti koje se nalaze u toj tabeli.

3.3.5. Security klasa

Klasu **Security** ćemo najdetaljnije predstaviti, pošto ona predstavlja samu srž naše aplikacije. Odnosno u okviru nje su smeštene metode koje koristimo da bi enkriptovali poruke koje se prosleđuju između klijenta i servera. Objekat klase **Security** se poziva u okviru metode **Run()** klase **Worker**.

Kao i u slučaju klase **Worker**, definisali smo konstruktor koji se poziva svaki puta kada se kreira neki objekat klase **Security**.

```
public Security() {  
  
    try {  
        generateKeyPair();  
    } catch (Exception e) {  
  
        e.printStackTrace();  
    }  
  
}
```

U okviru konstruktora ne radimo mnogo toga, sem što pozivamo metodu **generateKeyPair()**. Ova metoda je zadužena za generisanje javnog i privatnog ključa servera. Pošto je dovoljno samo jednom kreirati ove ključeve, odlučili smo se da metodu pozovemo iz konstruktora.

```
public void generateKeyPair() throws Exception {  
  
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");  
    SecureRandom random = new SecureRandom();  
  
    keyGen.initialize(1024, random);  
  
    KeyPair generatedKeyPair = keyGen.genKeyPair();  
  
    publicKey = generatedKeyPair.getPublic();  
    privateKey = generatedKeyPair.getPrivate();  
  
}
```

Prva stvar kojom se metoda bavi je kreiranje nove instance klase **KeyPairGenerator**, koja je zadužena za kreiranje javnog i privatnog ključa. Narednom naredbom takođe definišemo algoritam enkripcije za koji želimo da kreiramo par ključeva.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
```

Kada smo kreirali generator, potrebna nam je sekvenca nasumično izabranih brojeva. Koristićemo difolt verziju generatora.

```
SecureRandom random = new SecureRandom();
```

U slučaju da smo pisali server samo za operativni sistem Windows, mogli smo i da specificiramo algoritam koji će koristiti **SecureRandom** generator, pošto se po difoltu koristiti SHA1PRNG algoritam implementiran od strane Sun.

```
SecureRandom random = new SecureRandom.getInstance("SHA1PRNG", "SUN");
```

Međutim pošto naš server treba da radi i na Linux operativnom sistemu, odlučili smo se da pustimo sam sistem da odluči koji će algoritam koristiti.

Sledeći korak je inicijalizacija ključa na određenu dužinu, u našem slučaju 1024 bita.

```
keyGen.initialize(1024, random);
```

Kada smo dodelili sve parametre koje želimo da naš par ključeva poseduje ostalo je samo da ga i kreiramo.

```
KeyPair generatedKeyPair = keyGen.genKeyPair();
```

Klasa **KeyPair** služi samo za čuvanje naših ključeva i za njihovo pozivanje po potrebi. Kreiraćemo novi objekat **generatedKeyPair** ove klase i dodeliti mu generisane ključeve. Za generisanje ključeva se poziva metoda **genKeyPair()**. Sada je još ostalo da pozovemo metode

getPublic() i **getPrivate()** koje će vratiti vrednosti javnog i privatnog ključa respektivno.

```
publicKey = generatedKeyPair.getPublic();  
privateKey = generatedKeyPair.getPrivate();
```

Klasa Security sadrži metode **getPublicKey()** i **getPrivateKey()** koje vraćaju vrednosti odgovarajućeg ključa. Međutim, kada je reč o prosleđivanju vrednosti javnog ključa do klijenta, javlja se problem prenosa informacija pošto tip **Key** u kojem se čuvaju ključevi ne može da se pošalje korišćenjem soketa.

Da bi prevazišli problem, odlučili smo se da tip **Key** prebacimo u tip **String**, koji se najjednostavnije prosleđuje soketom. U te svrhe koristi se **Base64** koder.

```
String pubKeyStr = Base64.encodeToString(publicKey.getEncoded(), 0);
```

U okviru najnovije verzije JDK 8, ovaj koder je uključen kao standardna klasa. Međutim to nije slučaj sa JDK 7, koju mi koristimo zbog kompatibilnosti sa Android uređajem.

Postoji nekoliko implementacija ovog koda u okviru dodatnih biblioteka koje se mogu uvesti. Korišćenje najpoznatije Sun implementacije se smatra lošom praksom, pošto je reč o internom kodu, koji je podložan nedokumentovanim promenama. Drugo često rešenje je u okviru Apache commons klasa. U našem slučaju potrebno je voditi računa o tome i da Android ima svoju klasu **Base64** već implementiranu.

Da ne bi imali problema sa kompatibilnošću i prilagođavanjem koda, odlučili smo da Android **Base64** klasu, dodamo našem projektu i na taj način koristimo identičan kod i na klijentu i na serveru. Kada smo poslali privatni ključ klijentu, možemo da očekujemo da ćemo od njega dobiti enkriptovani AES ključ. Dakle, nije potrebno da u okviru servera implementiramo enkripciju RSA algoritmom, pošto se ona neće dešavati sa serverske strane, već samo sa klijentske. Ali je potrebno da implementiramo metodu za dekripciju dobijenog simetričnog ključa od strane klijenta.

Kao što smo već napomenuli poruke razmenjene između klijenta i servera će biti tipa **String**. To znači da će i enkriptovani simetrični ključ takođe biti primljen kao **String** tip. Primljeni ključ ćemo proslediti metodi **decryptRSA (String text)**, čiji je posao da izvuče simetrični ključ iz primljenog stringa.

```
public void decryptRSA(String text) throws Exception {  
    Cipher cipher=null;  
    byte[] decryptedData = null;  
    byte[] data = Base64.decode(text, 0);  
    cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");  
    cipher.init(Cipher.DECRYPT_MODE, getPrivateKey());  
    decryptedData = cipher.doFinal(data);  
    AESkeyb = decryptedData;
```

```

System.out.print("AES key:\n");
        System.out.print(Base64.encodeToString(AESkeyb, 0)+"\n");
    }

```

Prvo je neophodno primljeni String pretvoriti u tip sa kojim se može operisati. Za to koristimo ponovo **Base64** koder.

```
byte[] data = Base64.decode(text, 0);
```

Dakle, enkriptovani ključ, prevodimo iz tipa **String** u niz bajtova. Zatim iniciramo vrednost šifrata koji ćemo koristiti za dekripciju. Koristimo klasu **Cipher**.

```
cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
```

Definisali smo tip algoritma kao RSA, koji koristi ECB mod i format bloka PKCS1. Zatim inicijalizujemo šifrat sa odgovarajućim privatnim ključom i postavimo da radi u modu dekripcije.

```
cipher.init(Cipher.DECRYPT_MODE, getPrivateKey());
```

Na kraju pozivamo metodu **doFinal(data)** koja dešifruje poslani ključ, koji zatim smešta u promenljivu **AESkeyb**. Vrednost simetričnog ključa ćemo čuvati kao niz bajtova, zbog kasnijeg korišćenja.

```
decryptedData = cipher.doFinal(data);
AESkeyb = decryptedData;
```

Kada smo dešifrovali simetrični ključ, potrebno je još implementirati metode koje će se baviti enkripcijom odnosno dekripcijom AES algoritmom. Pošto će server morati i da enkriptuje podatke koje će slati klijentu, kao i da dekriptuje one koje dobija od klijenta, biće nam potrebne obe metode. U ovom slučaju ćemo obe metode imati i kod klijenta.

Prvo ćemo objasniti enkripciju, a zatim samo naglasiti razliku u slučaju dekripcije, pošto je proces u suštini sličan.

```

public String encrypt(String text) throws Exception {

    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        byte[] keyBytes = new byte[16];
        int len = AESkeyb.length;
        if (len > keyBytes.length)
            len = keyBytes.length;
    System.arraycopy(AESkeyb, 0, keyBytes, 0, len);
    SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
    IvParameterSpec ivSpec = new IvParameterSpec(keyBytes);
    cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec);

    byte[] results = cipher.doFinal(text.getBytes("UTF-8"));

        return Base64.encodeToString(results, 0);
}

```

Proces je sličan kao kod RSA algoritma. Prvo je neophodno da definišemo šifrat koji ćemo koristiti, odnosno njegove parametre postavimo na odgovarajuće vrednosti. U ovom slučaju to znači da ćemo imati AES algoritam, korišćen u CBC modu i prema PKCS5 standardu.

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Pošto AES radi u blokovima od 16 bajtova (odnosno 128bita), moramo prvo dobijeni ključ ograničiti na blok **keyBytes**. Prvo ćemo definisati **keyByte** kao niz od 16 bajtova, a zatim odrediti vrednost **len** koja će predstavljati dužinu našeg bloka za šifrovanje i ne sme prelaziti pomenutih 16 bajtova.

```
byte[] keyBytes = new byte[16];
int len = AESkeyb.length;
if (len > keyBytes.length)
    len = keyBytes.length;
System.arraycopy(AESkeyb, 0, keyBytes, 0, len);
```

Zatim ćemo koristeći funkciju **System.arraycopy** prekopirati niz bajtova iz **AESkeyb** promenljive u **keyBytes** promenljivu od početka niza do pozicije ograničene sa **len**. Zatim ćemo iz niza bajtova kreirati promenljivu **keySpec** tipa **SecretKeySpec**.

```
SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
IvParameterSpec ivSpec = new IvParameterSpec(keyBytes);
cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec);
```

Zatim kreiramo **ivSpec** promenljivu tipa **IvParameterSpec** koja predstavlja vektor za inicijalizaciju. CBC mod podrazumeva da se nad svakim blokom za enkripciju izvrši XOR operacija sa prethodnim blokom. Pošto u slučaju prvog bloka, nemamo prethodni blok, potrebno je kreirati sekvencu koja će omogućiti izvršavanje algoritma. Kada smo dodelili sve vrednosti, inicijalizujemo vrednost šifrata sa odgovarajućim promenljivama i podešavamo mod na enkripciju **Cipher.ENCRYPT_MODE**. Ostalo je još da ponovo pozovemo metodu **doFinal** da izvrši samu enkripciju.

```
byte[] results = cipher.doFinal(text.getBytes("UTF-8"));
```

Pošto je dobijeni rezultat niz bajtova, neophodno je da pomoću **Base64** kodera, prevedemo taj niz bajtova u **String**.

```
return Base64.encodeToString(results, 0);
```

U slučaju dekripcije postupak će biti sličan, sem što ćemo umesto **ENCRYPT_MODE** postaviti **DECRYPT_MODE** pri inicijalizaciji šifrata.

```
cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec)
```

4. IMPLEMENTACIJA KLIJENTA

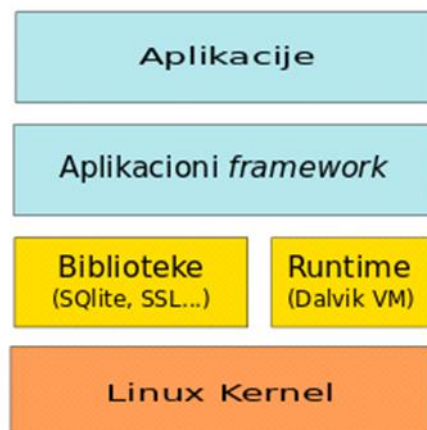
U ovom delu ćemo objasniti realizaciju klijenta. Pošto je naš klijent ustvari Android aplikacija, prvo ćemo reći par reči o operativnom sistemu Android kao i alatu za razvoj aplikacije Android SDK. Zatim ćemo obraditi implementaciju grafičkog interfejsa i funkcionalnosti aplikacije. Pošto se veliki deo koda poklapa sa već objašnjenim na strani servera, trudićemo se da naglasimo samo funkcionalnosti koje se ovde prvi put uvode.

4.1. Android SDK

Jedan primer SDK interfejsa je Android SDK. Može se preuzeti besplatno sa Interneta i obično se instalira u okviru Eclipse razvojnog okruženja na računaru. Razvoj aplikacija se vrši na samom računaru, a zatim se spušta na mobilni telefon sa Android operativnim sistemom.

Na primeru Android SDK možemo razmotriti praktičnu primenu SDK. Android je pre svega predviđen da se na njemu pišu aplikacije u Javi, tako da je i Android SDK kreiran u tu svrhu.

Android je baziran na Linux kernelu. Projekat koji je zadužen za razvoj Android sistema se naziva Android Open-Source Project i prvenstveno ga vodi Google.



Slika 4.1.1 – Nivoi Android platforme

Nivoi koje možemo uočiti odozgo na dole su:

- Aplikacije
- Aplikacioni *framework* – API koji omogućava interakciju između Android sistema i aplikacija na njemu.
- Biblioteke i runtime
- Linux kernel - sloj za komunikaciju sa hardverom.

Programere koji treba da razvijaju aplikacije za Android interesuju samo najviša dva sloja. To im omogućava SDK.

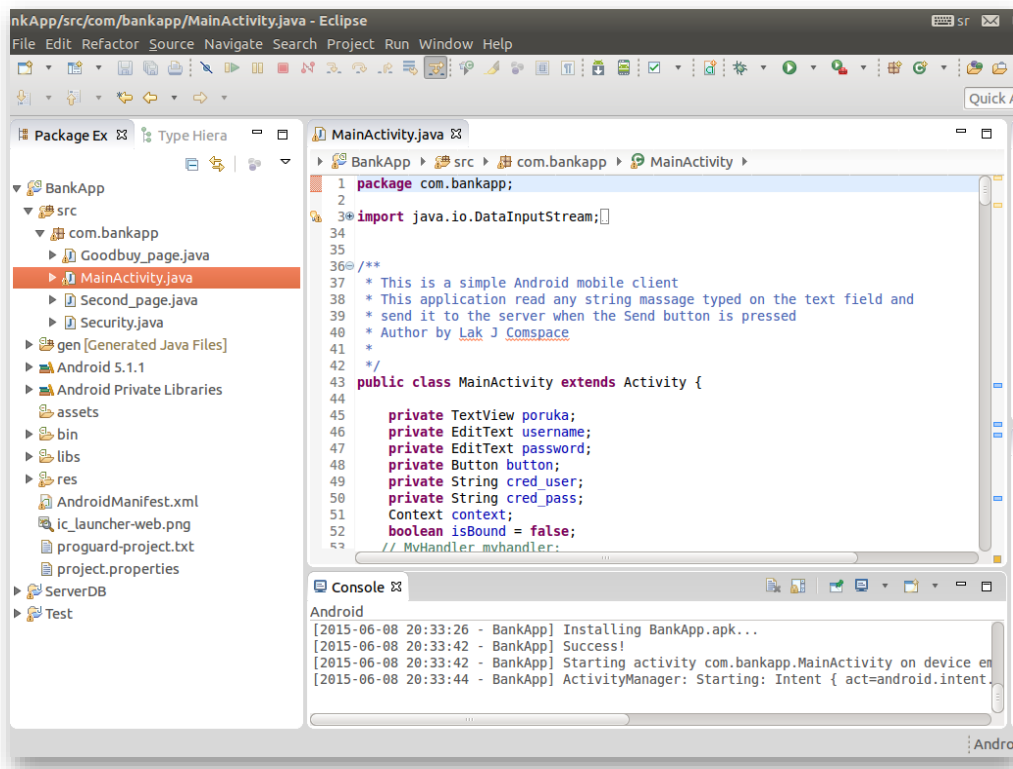
Kao što možemo videti na slici 4.1.1 Android koristi *Dalvik Virtual Machine* da bi pokretao aplikacije napisane u Javi. Iako radi slično kao *Java Virtual Machine* koristi drugačiji *bytecode*, tako da se klase napisane u Javi ne mogu direktno pokrenuti na Androidu. Mora se vršiti konverzija. Tu na scenu stupa Android SDK koji sadrži alat *dx*, koji vrši konverziju Java klasa u *.dex* (Dalvik Executable) fajl.

Da bi razvoj aplikacija bio jednostavniji Android SDK sadrži i emulator raznih Android uređaja. U okviru emulatora se može izabrati *Android Virtual Device* (AVD) koji sadrži emulacije raznih vrsta Android mobilnih telefona. Aplikacija se na ovaj način može testirati u AVD-u pre puštanja na sam mobilni uređaj.

4.2. Razvoj Klijenta

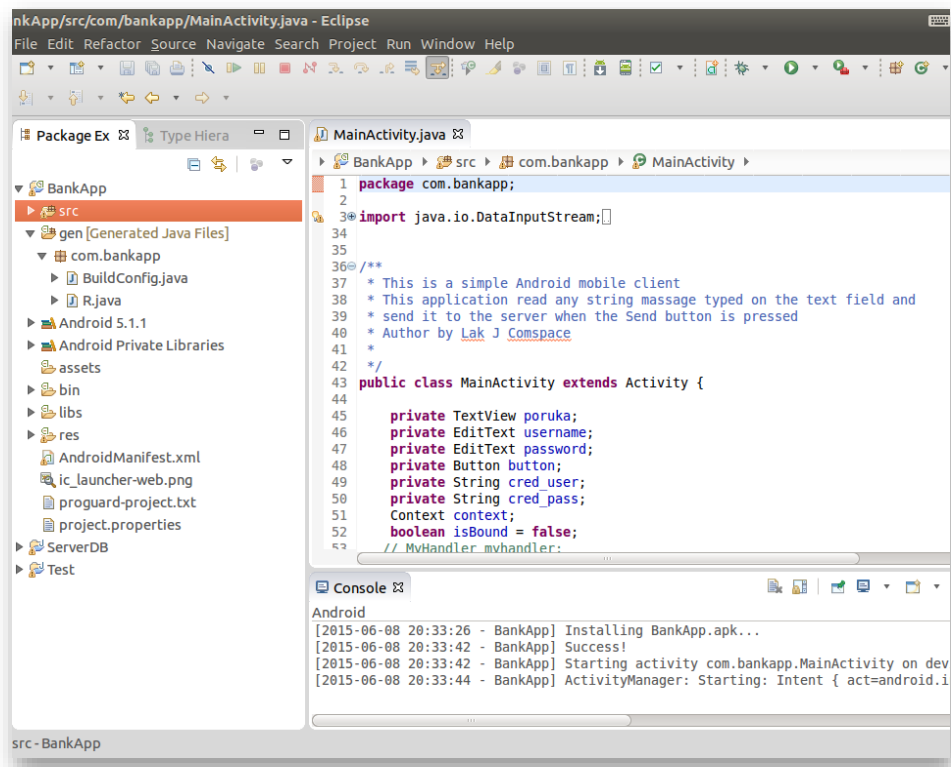
Da bi objasnili klijentsku aplikaciju, prvo moramo predstaviti koncept jednog android projekta. Ovde je reč o aplikaciji koja ima i grafički korisnički interfejs, pored standardnih *.java* fajlova pa ćemo imati i *xml* fajlove koji će definisati izgled.

Počecemo sa objašnjenjem kako je organizovan projekat.



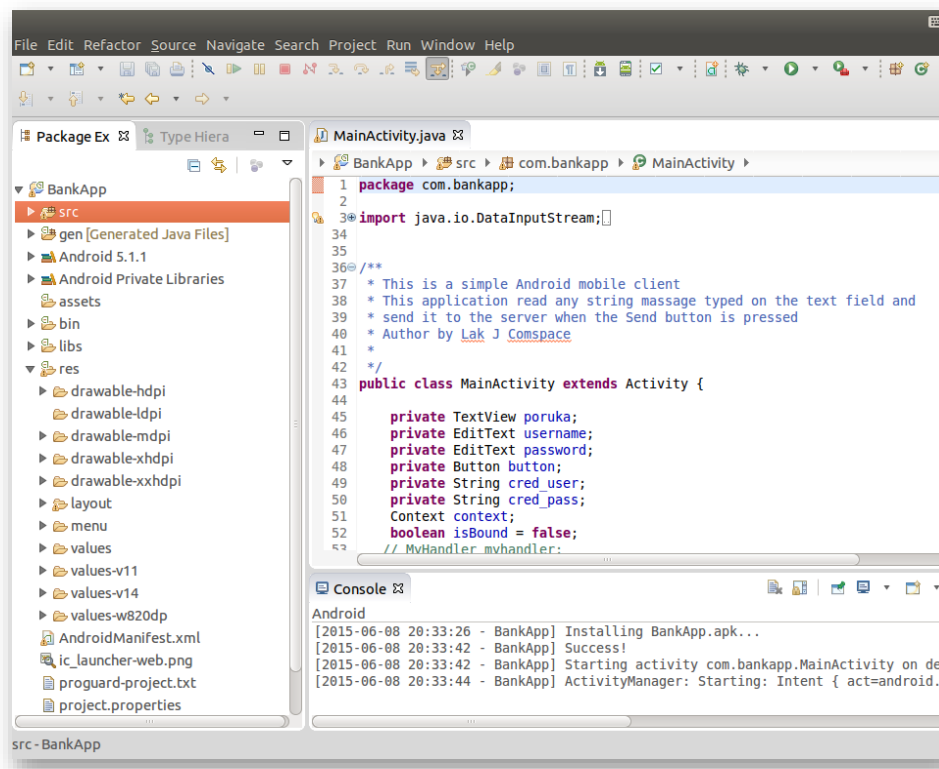
Slika 4.2.1 – Src folder u Eclipse projektu

Kao što možemo da vidimo na slici 4.2.1 pored standardnog *src* foldera u kojem će biti smešteni naši sors fajlovi, postoji još nekoliko dodatnih foldera, od kojih ćemo predstaviti samo one od značaja za našu aplikaciju.



Slika 4.2.2 – Gen folder u Eclipse projektu

Prvi folder na koji treba obratiti pažnju je folder pod imenom **gen**. Ovde program smešta generisane java fajlove, koji su potrebni za pravilno funkcionisanje aplikacije. U ovaj folder se ne sme ništa dodavati, odnosno brisati. Ukoliko je sve podešeno kako treba, program će sam generisati ove fajlove. Ovde ga ipak pominjemo, pošto se može desiti da ukoliko Android SDK ili JDK nisu dobro instalirani ili kompatibilni, pa generisanje ovih fajlova izostane. U tom slučaju program neće izvršavati neke osnovne Android naredbe, tako da nije loše imati ovo u vidu.



Slika 4.2.3 – Res folder u Eclipse projektu

Sledeći bitan folder se naziva **res** i služi za skladištenje resursa, odnosno slika, xml fajlova, video fajlova i tako dalje. Ovaj folder se sastoji iz više podfoldera, koji raščlanjuju malo naše resurse. Za nas su od značaja podfolderi **layout** i **values**. U **layout** podfolderu smešteni su xml fajlovi koji definišu izgled aplikacije, dok se u **values** podfolderu čuvaju vrednosti određenih promenljivih u xml fajlu koje pozivamo po referenci. U slučaju naše aplikacije ovo će se odnositi pre svega na duže stringove, koje je jednostavnije čuvati odvojeno, u slučaju da odlučimo da ih modifikujemo.

4.2.1. *AndroidManifest.xml*

U **AndroidManifest.xml** fajlu se nalaze generalni podaci o aplikaciji, koji su potrebni mobilnom uređaju za izvršavanje ove aplikacije. U trenutku kada napravimo projekat za aplikaciju, **AndroidManifest.xml** se automatski generiše. Kako dodajemo nove komponente našoj aplikaciji, sve standardne informacije će biti automatski upisane u ovaj fajl. Ukoliko ipak želimo da aplikaciji dozvolimo neke posebne permisije, potrebno je ručno ih dodati u fajl.

U slučaju naše aplikacije dodali smo još dve posebne permisije.

```
<uses-permission android:name="android.permission.INTERNET" >
</uses-permission>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" >
</uses-permission>
```

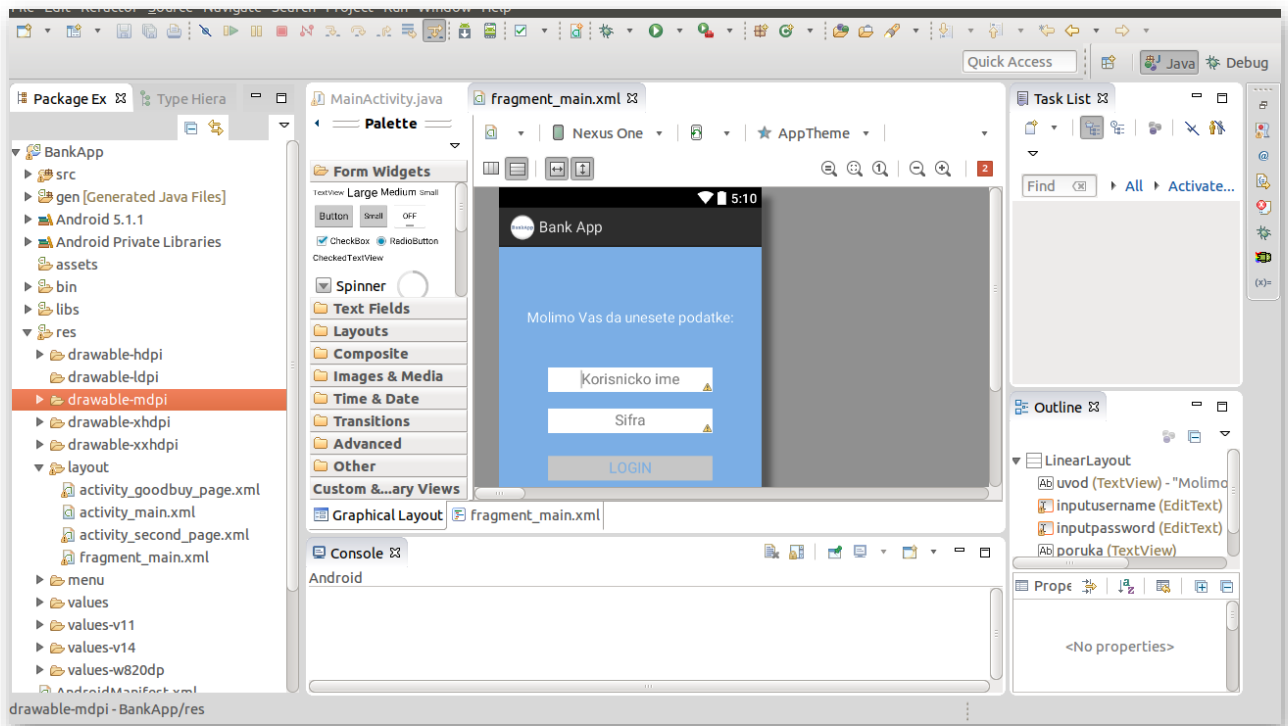
U prvom slučaju dozvoljavamo aplikaciji da pristupi Internetu, dok u drugom dozvoljavamo

aplikaciji da dobije podatke o mreži.

4.2.2. Implementacija grafičkog interfejsa

Aplikacija koju razvijamo imaće tri stranice. Početnu na kojoj će se unositi kredencijali, stranicu sa iščitanim podacima i stranicu koja obaveštava o završetku sesije. Za svaku od ovih stranica mora postojati xml fajl koji određuje kako će biti raspoređeni elementi na stranici.

Koristićemo samo standardne Android elemente koji se već nalaze u okviru SDK, mada postoji mogućnost skidanja dodatnih komplikovanijih elementata.



Slika 4.2.2.1 – Izgled aplikacije u Eclipse razvojnom okruženju

Na slici 4.2.2.1 možemo videti grafički prikaz xml fajla. Korisnik može da izabere tab sa grafičkim prikazom ili xml prikazom. Sa leve strane se mogu videti razni elementi koji se mogu dodati na okvir.

Prvo je neophodno da definišemo **layout** koji ćemo koristiti. U našem slučaju će to za sve stranice biti linearni. To podrazumeva da se elementi linearno postavljaju jedan za drugim.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    android:background="#79BAEC"
    >
```

```
<!-- ovde definišemo elemente
-->
</LinearLayout>
```

Možemo imati vertikalnu ili horizontalnu orijentaciju, zavisno od toga da li želimo da se elementi postavljaju jedan ispod drugog ili jedan do drugog. Mi ćemo koristiti vertikalnu.

Kada smo definisali **layout**, potrebno je da dodamo i elemente. Za realizaciju aplikacije potrebna su nam samo tri elementa:

- TextView
- EditText
- Button

Kombinovanjem i doradivanjem difolt verzija ovih elemenata, dobićemo stranicu kao što se može videti na slici 4.2.2.1.

```
<TextView
    android:id="@+id/uvod"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_marginTop="75dp"
    android:text="@string/uvod"
    android:textSize="17sp"
    android:textColor="#FFFF"
/>
```

Potrebno je da odmah definišemo **id** elementa, da bi kasnije mogli da ga pozivamo. Sistem će odmah po dodavanju elementa na stranicu, dodeliti elementu difolt id, ali je poželjno da ga promenimo na nešto što konkretnije označava o kom je elementu reč. Tačnije funkcionalno je svejedno, ali je lakše snaći se pri pozivanju elemenata kasnije.

U okviru elementa možemo definisati dužinu, visinu, orijentaciju, boju, margine u odnosu na ostale predmete, boju itd.

Kada je reč o TextView elementu, obično se podrazumeva neki tekst u okviru elementa. Ovaj tekst se može hardkodovati u okviru definisanja elementa, ali je bolja praksa da se definiše samo pokazatelj na tekst u okviru drugog xml fajla koji se naziva strings.xml.

```
android:id="@+id/uvod"
```

U ovom xml fajlu se čuvaju vrednosti svih stringova na jednom mestu, zbog jednostavnijeg menjanja ukoliko je to potrebno.

```
<resources>
...
<string name="uvod">Molimo Vas da unesete podatke:</string>
...
</resources>
```

Na sličan način se definiše i element Button, dok u slučaju EditText elementa postoji još par opcija koje nije loše pomenuti. Pošto je ovaj tip elementa tekstualno polje u koje je potrebno nešto uneti, možemo definisati kakav tip unosa želimo. U našem slučaju to će biti tekst.

```
android:inputType="text"
```

Takođe je korisno kada u samom polju možemo da sugerišemo korisniku šta treba uneti. Vizuelno je bolje rešenje od standarnog natpisa iznad polja i jednostavnije za implementaciju

```
android:hint="Korisnicko ime"
```

Na kraju je potrebno i pozabaviti se fokusom u trenutku kada nam se uključi aplikacija. Pošto podrazumevamo da će korisnik po pokretanju aplikacije želeti da unese svoje kredencijale, definisaćemo fokus tako da pokazuje na prvo tekstualno polje. Potrebno je postaviti tag

```
<requestFocus />
```

unutar odgovarajućeg elementa da bi se ovo postiglo.

4.2.3. Implementacija funkcionalnosti klijenta

Za svaki definisani xml fajl stranice postoji odgovarajući java fajl koji implemmentira funkcionalnost. U okviru Androida glavna klasa koja mora da postoji za svaku stranicu je **Activity**. Pri tome treba voditi računa da se prvo uvek poziva **MainActivity**, pa zatim druge klase odatle.

Svaka klasa **Activity** se sastoji od metoda koje opisuju njen životni ciklus. Ciklus počinje pozivanjem metode **onCreate()** i završava se metodom **onDestroy()**. Mogu se implementirati i dodatne metode kao što su **onStart()**, **onStop()**, **onPause()**, **onRestart()** itd. S obzirom da je akcenat pri razvoju aplikacije stavljen na implementaciji sigurne komunikacije, korišćićemo samo prve dve metode. Ovde treba naglasiti da se **onCreate()** metoda poziva pri pokretanju aplikacije, dok metodu **onDestroy()** treba eksplicitno pozvati iz same aplikacije.

Sve ono što se izvršava u okviru **Activity** klase, se izvršava na glavnoj UI (*user interface*) niti. Za optimalne performanse nije poželjno izvršavati nikakve komplikovane operacije.

Objasnićemo malo detaljnije kako je realizovana MainActivity.java klasa aplikacije koju razvijamo. Prva stvar koju treba da odradimo je da povežemo elemente definisane u okviru fragment_main.xml fajla sa promenljivim u našoj aplikaciji da bi mogli da manipulišemo unesenim podacima i vraćamo rezultat.

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.fragment_main);

    username = (EditText) findViewById(R.id.inputusername); // reference
to
```

```

the username field

password = (EditText) findViewById (R.id.inputpassword); //reference to
the password field

poruka = (TextView) findViewById (R.id.poruka); //reference to the
poruka field

button = (Button) findViewById(R.id.button1); // reference to the send
button

```

Postavljanje sadržaja stranice će takođe biti smešteno u **onCreate()** metodi, pošto se ona poziva odmah po pokretanju aplikacije. Dakle prvo ćemo metodom **setContentview** postaviti sadržaj stranice iz `fragment_main.xml` fajla.

Zatim se metodom **findViewById** vezuje element definisan u xml fajlu sa promenljivom u okviru klase. Element se identifikuje pomoću **id** vrednosti.

Pošto smo definisali elemente, između kojih i Button, treba da implementiramo ponašanje aplikacije. To ćemo uraditi tako da se pritiskom na Button (dugme) pokreću određeni procesi.

```

button.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {

        cred_user = username.getText().toString(); // get the
text message on the text field
        username.setText(""); // Reset the text field to blank
        cred_pass = password.getText().toString(); // get the
message on the text field
        password.setText(""); // Reset the text field to blank
        credentials = cred_user + ":" + cred_pass;

        if (!(credentials.equals(":")) ){

            Thread service = new Thread (new ServiceTest());
            service.start();
            dialog=new ProgressDialog(context);
            dialog.setMessage("Sacekajte...");
            dialog.setCancelable(false);
            dialog.setInverseBackgroundForced(false);
            dialog.show();

        }
        else {
            poruka.setText("Niste uneli podatke");
        }

    }

});

```

Prvo ćemo pozvati interfejs iz Android klase **View**, čija je uloga da osluškujе da li je pokrenut odgovarajući događaj. U ovom slučaju to će biti **onClick**, odnosno ukoliko se pritisne odgovarajući osluškivani element, dolazi do izvršavanja naredbi unutar metode.

Dakle kada je pokrenuta metoda **onClick**, iščitavaju se login kredencijali metodom **getText().toString()** iz odgovarajućih polja, prebacuju u string i čuvaju u odgovarajućim promenljivim. Ukoliko je dobijena promenljiva prazna, korisniku se ispisuje poruka da nije unео podatke, i aplikacija ostaje u fazi čekanja ponovnog pritiska na Button.

Ukoliko promenljiva nije prazna, odnosno podaci su uneti potrebno je obraditi podatke i vratiti korisniku odgovarajuće podatke. Kao što smo već pomenuli nije poželjno nikakve duge procese izvršavati na UI niti, tako da ćemo komunikaciju sa serverom smestiti na posebnu nit koja se nalazi u okviru **MainActivity** klase. Dakle ova nova nit će biti unutrašnja klasa klase **MainActivity**. Na ovaj način nit odnosno klasa **ServiceTest()** može da koristi sve promenljive klase **MainActivity**, kao i obrnuto.

Odmah po definisanju i pokretanju unutrašnje niti, pokrenućemo i prozor koji će korisnika obavestiti da treba da sačeka. Metoda **dialog.setMessage()** će postaviti poruku u okviru prozora koju želimo da ispišemo, dok će metoda **dialog.show()** prikazati prozor. Ovaj prozor će ostati aktivan sve dok ne pređemo na drugu stranicu.

ServiceTest klasa

ServiceTest klasa je kao što smo već objasnili unutrašnja klasa koja implementira **Runnable** interfejs. Po pozivanju klase pokreće se metoda **run**, koja kreira socket i otvara konekciju ka serveru.

```
public class ServiceTest implements Runnable{
    DataOutputStream dataOutputStream;
    DataInputStream dataInputStream;
    private Socket client;

    public void run() {

        // otvaranje socketa i povezivanje na server
        if (client == null) {
            try{
                client = new Socket("10.0.2.2", 4444);
            } catch (UnknownHostException e) {

                e.printStackTrace();

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Ova klasa odgovara **Worker** klasi sa strane servera i koristi pomoćnu klasu **Security** koja sadrži metode za enkripciju.

```
Security security = new Security();
    SendMessage("Start");
```

Program kreira novi objekat klase **Security** i zatim pokreće razmenu podataka sa serverom komandom **Start**. Program zatim ulazi u **while** petlju kao i u slučaju sa serverom, gde prima poruke a zatim zavisno od dobijene komande skače na odgovarajući deo koda.

```
while(true)
{
    String response = RecieveMessage();

    if (response.equals("Start OK")){

        //uradi nešto

    if (response.equals("Key OK")){

        //uradi nešto

    if (response.equals("Key OK")){

        // uradi nesto

    if (response.equals("Key OK")){

        //uradi nesto

    if (response.equals("LogIn OK")){

        //uradi nesto

    if (response.equals("Log in FAIL")){

        //uradi nesto

    if (response.equals("Log in OK")){

        //uradi nesto

    if (response.equals("ClientData OK")){

        //uradi nesto

    if (response.equals("End OK")){

        //uradi nesto
```

Pošto je kod dosta dugačak, a prilično sličan već objašnjenom kodu u okviru servera koncentrisaćemo se samo na one delove gde je kod specifičan za Android klijenta. To se dešava u dva slučaja.

Ukoliko smo zaključili da se kredencijali koje je korisnik uneo ne poklapaju sa zapisima koje imamo u bazi, potrebno je izbrisati unete vrednosti i vratiti korisniku poruku.

```
if (response.equals("Log in FAIL")){

    mActivity.runOnUiThread(new Runnable() {
```

```

public void run() {
    dialog.dismiss();
    poruka.setText("Uneli ste pogrešne podatke");
    button.setClickable(true);
}
});
break;

```

Dakle ukoliko je odgovor od servera negativan potrebno je vratiti se na glavnu UI nit, ispisati grešku i čekati novi unos. Povratak na glavnu nit postizemo **mActivity.runOnUiThread** naredbom. Ovde treba imati u vidu da smo ranije u okviru glavne niti definisali vrednost promenljive **mActivity** i u praksi ova promenljiva predstavlja referencu na određenu aktivnost.

Kada smo se vratili na UI nit, potrebno je da uklonimo prozor koji obaveštava korisnika da treba da sačeka metodom **dialog.dismiss()**. Zatim metodom **setText()** ispisujemo poruku o pogrešnim podacima korisniku i čekamo novi unos.

Drugi specifični slučaj je ukoliko je autentifikacija prošla uspešno i potrebno je da korisniku ispišemo odgovarajuće podatke.

```

if (response.equals("End OK")){

    Intent intent = new Intent(MainActivity.this, Second_page.class);
    intent.putExtra("message", result);
    startActivity(intent);
    finish();
break;

}

```

Podaci će biti ispisani u okviru druge stranice aplikacije, odnosno to je drugi **Activity**. Da bi pozvali drugi **Activity** koristimo **Intent**. **Intent** govori aplikaciji šta program namerava sledeće da uradi. Pošto nova aktivnost, znači i nova klasa, moramo nekako da prosledimo rezultate dobijene od servera toj klasi. Za to se koristi metoda **intent.putExtra(id poruke, poruka)**. Kada smo definisali nameru da pokrenemo novu aktivnost, pokrećemo je pomoću **startActivity(intent)** naredbe.

Na kraju je ostalo još da pozovemo metodu **finish()**. Metoda **finish()** pokreće metodu **onDestroy()** u okviru životnog ciklusa aplikacija.

```

protected void onDestroy() {
    super.onDestroy();

    credentials=null;
    result=null;

}

```

U našem slučaju **onDestroy()** metoda samo briše unete promenljive, pored izvršavanja difolt procesa pri gašenju aktivnosti.

Second_page klasa

Second_page je druga stranica razvijene aplikacije, koja korisniku treba da prikaže stanje računara. Stranica se sastoji od nekoliko polja u koja aplikacija ispisuje odgovarajuće informacije dobijene od servera. Odnosno, klasa prima informacije od **MainActivity** klase i zatim ih ispisuje na ekranu. Da bi se dobile ove informacije potrebno je definisati **Bundle** objekat, u koji će biti smešteni podaci.

```
Bundle extras = getIntent().getExtras();
String ret = extras.getString("message");
```

Zatim ćemo iz ovog objekta izvući samo string koji ima ranije definisani **id**. Program će zatim ispisati podatke na sličan način kao u prvoj aktivnosti i ostati aktivan 10 sekundi posle čega će pokrenuti treću pozdravnu stanicu.

U tu svrhu potrebno je da definišemo objekat **Timer** u okviru aktivnosti i da ga postavimo na 10 sekundi.

```
Timer t = new Timer();
t.schedule(task, 10000);
```

Kada istekne predviđeno vreme, pokreće se metoda **run** definisana u okviru interfejsa **TimerTask**.

```
TimerTask task = new TimerTask() {

    @Override
    public void run() {
        Intent intent = new Intent(Second_page.this, Goodbye_page.class);
        startActivity(intent);
        finish();
    }
};
```

Drugim rečima poziva se treća aktivnost, odnosno **Goodbye_page** klasa koja će ispisati pozdravnu poruku korisniku i posle 5 sekundi se ugaziti, na sličan način kao i prethodne dve. Dakle, otvara se treća stranica koja sadrži samo jedno polje u koje program ispisuje poruku korisniku da mu je sesija istekla. Ovo je još jedan vid zaštite poverljivih informacija od trećih lica. U praksi je period trajanja sesije malo duži i obično iznosi par minuta. Veoma je jednostavno u kodu razvijene aplikacije promeniti dužinu trajanja sesije, a radi jednostavnijeg testiranja je izabran kratak period od 5 sekundi.

Security klasa

Na kraju je ostalo još da pomenemo pomoćnu klasu **Security**. Ova klasa odgovara istoimenoj klasi sa server strane. Metode za enkripciju i dekripciju AES protokolom su identične, dok je metoda dekripcije RSA protokolom, ovde zamenjena metodom enkripcije. Razlike između metoda dekripcije i enkripcije su u velikoj meri već objašnjene kod AES protokola, tako da se nećemo time baviti.

Objasnićemo samo klasu koja se bavi kreiranjem AES ključa, pošto je to posao koji odrađuje samo klijent.

```
public byte[] createKeyForAES()
    throws NoSuchAlgorithmException, NoSuchProviderException {
```

```
SecureRandom random = new SecureRandom();
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(256, random);
AESKeyb = generator.generateKey().getEncoded();
Log.d("aes key", Base64.encodeToString(AESKeyb, 0));
return AESKeyb;
}
```

Klasa se naziva **createkeyforAES**. Potrebno je definisati generator slučajnih brojeva koji se ćemo nazvati **random** i generator ključa **generator** kod kojeg ćemo definisati algoritam na AES. Zatim ćemo postaviti dužinu ključa na 256 bita, i dodeliti generatoru niz slučajnih brojeva naredbom **generator.init(broj_bita, niz_slučajnih_brojeva)**. Na kraju naredbom **generator.generateKey()** kreiramo AES ključ.

5. VERIFIKACIJA RADA APLIKACIJE

U ovom poglavlju ćemo se baviti verifikacijom rada aplikacije. U okviru ovog poglavlja biće dato jednostavno uputstvo za korišćenje aplikacije. Koristićemo emulator za prikaz korišćenja aplikacije, a zatim ćemo predstaviti i izgled aplikacije na jednom Android uređaju.

Pre svega je potrebno da instaliramo aplikaciju na emulator. Prvo ćemo proveriti da li je emulator uključen, naredbom **adb devices**.

```
ivana-Inspiron-3542: ~/workspace/BankApp/bin
ivana@ivana-Inspiron-3542:~/workspace/BankApp/bin$ adb devices
List of devices attached
emulator-5554 device

ivana@ivana-Inspiron-3542:~/workspace/BankApp/bin$
```

Slika 5.1 – Prikaz adb device funkcije

Možemo videti da je uređaj emulator-5554 priključen i da ga naredba prepoznaje. Sada možemo da instaliramo aplikaciju na njega. Prvo ćemo promeniti direktorijum na onaj u kome se nalazi instalacioni fajl za Android.

```
ivana-Inspiron-3542: ~/workspace/BankApp/bin
ivana@ivana-Inspiron-3542:~/workspace/BankApp/bin$ ll
total 1684
drwxrwxr-x 5 ivana ivana 4096 Jun 8 19:56 ./
drwxrwxr-x 8 ivana ivana 4096 May 21 21:34 ../
-rw-rw-r-- 1 ivana ivana 1449 Jun 8 19:22 AndroidManifest.xml
-rw-rw-r-- 1 ivana ivana 440641 Jun 8 19:56 BankApp.apk
drwxrwxr-x 3 ivana ivana 4096 Jun 5 00:59 classes/
-rw-rw-r-- 1 ivana ivana 1216592 Jun 8 19:56 classes.dex
drwxrwxr-x 2 ivana ivana 4096 May 24 16:52 dexedLibs/
-rw-rw-r-- 1 ivana ivana 120 Jun 5 00:58 jarlist.cache
drwxrwxr-x 3 ivana ivana 4096 May 24 16:52 res/
```

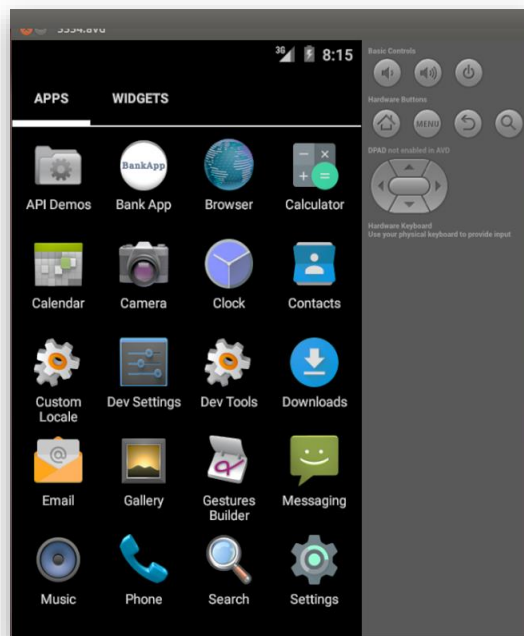
Slika 5.2 – Prikaz navigacije ka instalacionom fajlu BankApp.apk

Kao što možemo da vidimo na slici 5.2 fajl se naziva BankApp.apk. Sada ćemo naredbom **adb install -r ime_fajla** instalirati aplikaciju na emulator.

```
ivana@ivana-Inspiron-3542:~/workspace/BankApp/bin$ adb install -r BankApp.apk
390 KB/s (440641 bytes in 1.101s)
pkg: /data/local/tmp/BankApp.apk
Success
ivana@ivana-Inspiron-3542:~/workspace/BankApp/bin$
```

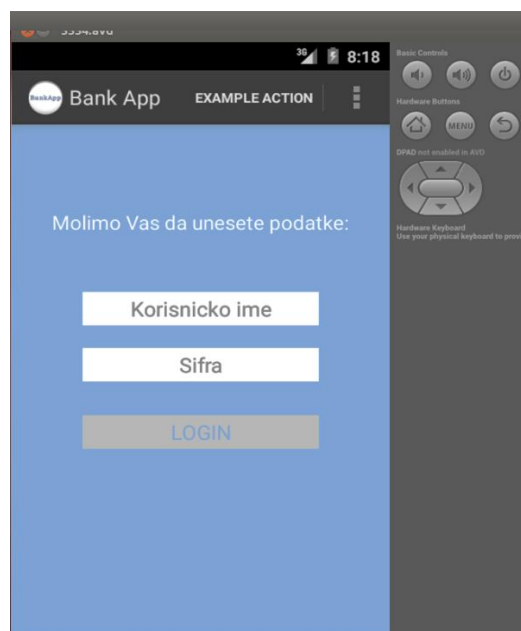
Slika 5.3 – Prikaz uspešne instalacije BankApp

Ukoliko je aplikacija uspešno instalirana, dobićemo poruku "Success" od sistema. Sada možemo da pređemo na emulator i vidimo da li je aplikacija zaista instalirana.



Slika 5.4 - Prikaz ikonice BankApp aplikacije na emulatoru

Kao što možemo videti na slici 5.4, među aplikacijama se nalazi Bank App ikonica. Kliknućemo dva puta na ikonu i pokrenuti aplikaciju.

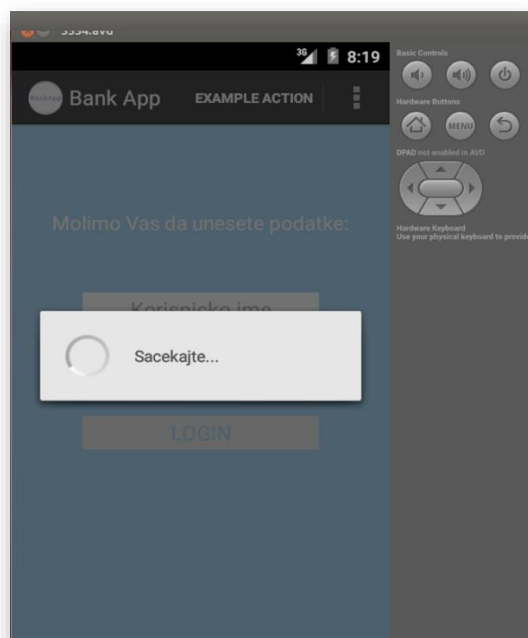


Slika 5.5 – Log in stranica BankApp aplikacije

Prvo će se otvoriti login stranica, gde je neophodno uneti validne kredencijale u odgovarajuća polja. Zatim je potrebno pritisnuti **Login** dugme, da bi se poslali podaci aplikaciji. Aplikacija ove

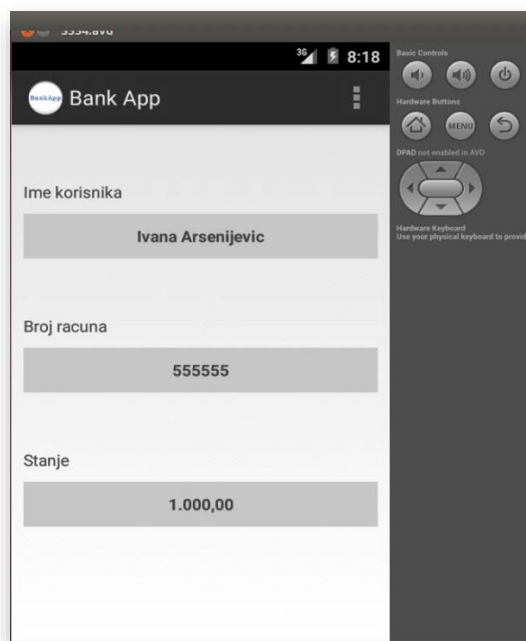
podatke šalje serveru, a zatim dobija pozitivan ili negativni odgovor od njega. U slučaju da dobije pozitivan odgovor prikazuje korisniku stanje računa, dok u slučaju negativnog odgovora vraća grešku.

Pošto su podaci koje smo uneli validni, sistem treba da vrati stanje računa korisnika.



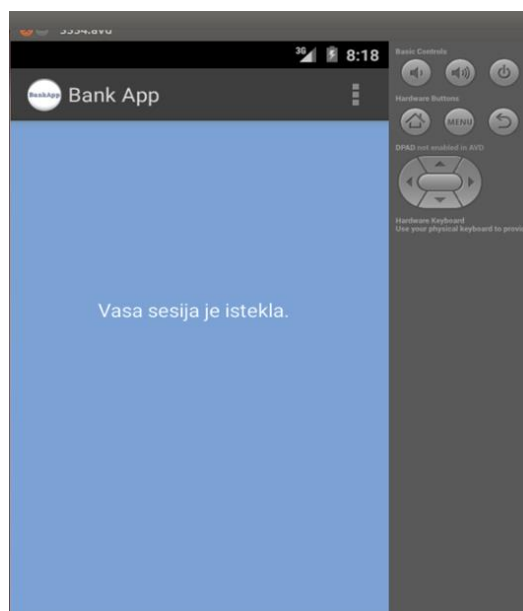
Slika 5.6 – Prikaz prozora koji obaveštava korisnika da sačeka

Dok sistem obrađuje podatke, korisniku iskače prozor koji ga upozorava da treba da sačeka da se podaci obrade.



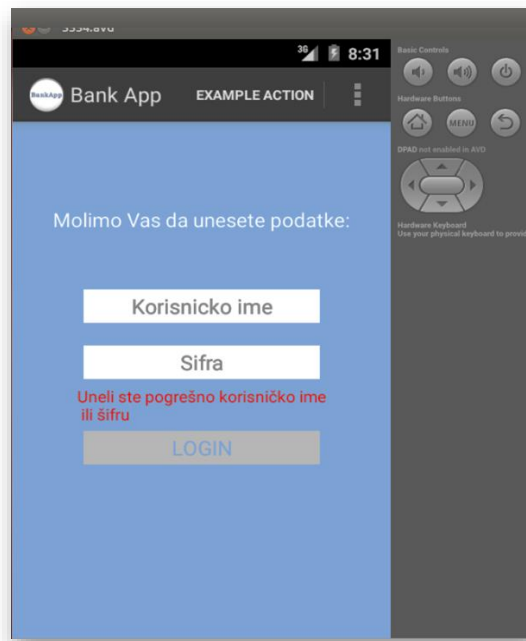
Slika 5.7 – Prikaz računa korisnika

Kada završi sa obradom, sistem vraća novu stranicu sa prikazom stanja računa. Ova stranica je aktivna svega 5 sekundi, posle čega korisnik dobija poruku o isteku sesije.



Slika 5.8 – Pozdravna strana aplikacije

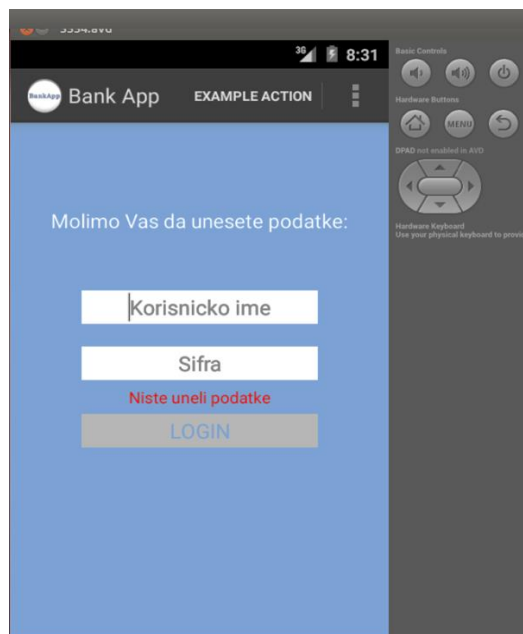
Ovim se aplikacija gasi i za ponovno proveravanje podataka potrebno je ponovo pokrenuti aplikaciju BankApp. Kada smo razmotrili šta se dešava u pozitivnom slučaju, potrebno je da razmotrimo i negativan slučaj. U negativnom slučaju korisnik je uneo pogrešne kredencije i server je poslao negativan odgovor. I u ovom slučaju korisniku je prikazan prozor koji ga obaveštava da treba da sačeka, posle čega se vraća na početni ekran.



Slika 5.9 – Poruka neuspešne autentifikacije

Ovaj put početni ekran prikazuje i poruku o neuspešnom logovanju, odnosno pogrešnim kredencijalima. Korisnik ipak može da ponovo pokuša da unese validne kredencijale.

Treba još da razmotrimo slučaj u kojem je korisnik ostavio oba polja prazna i pritisnuo dugme. U ovakvim slučajevima nema potrebe za kontakiranjem servera, već aplikacija odmah izbacuje poruku.



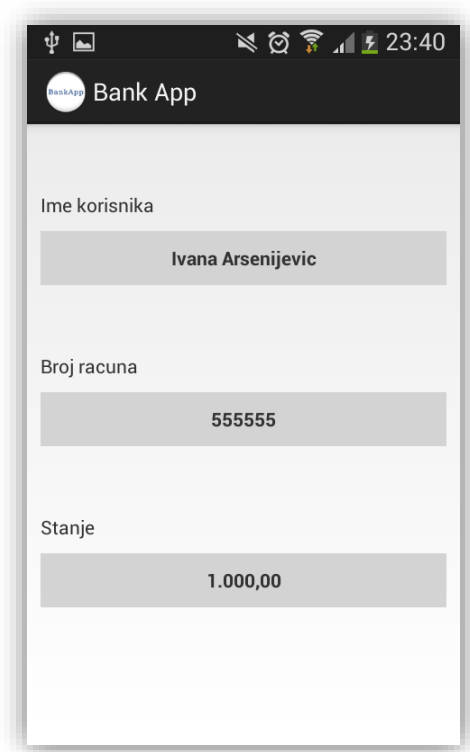
Slika 5.10 – Poruka korisniku da nije uneo podatke

Korisnik je ponovo u mogućnosti da unese validne podatke, posle čega će biti uspešno ulogovan na sistem.

Na kraju je još ostalo da prikazemo kako aplikacija izgleda na Android uređaju. U tu svrhu ćemo koristiti Samsung Galaxy Core Plus uređaj sa verzijom Androida 4.2.2.



Slika 6.11. – Login stranica BankApp aplikacije na Samsung uređaju



Slika 6.12 – Račun korisnika BankApp aplikacije na Samsung uređaju

Kao što možemo videti na slikama 6.11 i 6.12 najveća razlika u odnosu na emulator je u predstavljanju boja na ekranu. Elementi su raspoređeni srazmerno veličini ekrana. Pored Samsung uređaja, aplikacija je uspešno testirana i na HTC Desire X uređaju.

6. ZAKLJUČAK

U ovom radu smo pokušali da pokrijemo jedan mali deo opsega koji danas dostiže zaštita podataka. Ovaj opseg se razvija svakodnevno, pokušavajući da nadjača napore hakera da dođu do poverljivih podataka. Sa migriranjem ka mobilnim platformama, sa stalnim izlazom na Internet kao što su Android uređaji treba biti posebno pažljiv.

Danas ljudi svoje mobilne uređaje koriste i kao fotoaparate, za povezivanje na društvene mreže, email itd. Samo smo korak od trenutka kada će se mobilni uređaji koristiti i za plaćanja umesto kreditnih kartica, kao ključevi ili za iščitavanje identifikacionih podataka.

Naša aplikacija pruža jedno rešenje za siguran prenos podataka ka i od Android uređaja. Aplikacija je jednostavna za korišćenje i proces enkripcije je transparentan sa strane korisnika. Kombinovanjem AES i RSA protokola postigli smo da aplikacija ima zadovoljavajući stepen zaštite, sa rešavanjem problema transporta ključa između klijenta i servera. Sa druge strane odziv aplikacije je svega par sekundi, pa korisnik ne trpi posledice komplikovane obrade podataka.

Aplikacija je u potpunosti funkcionalna, mada bi bilo neophodno implemenirati dodatne funkcionalnosti da bi se povećala upotrebna vrednost. Na osnovu istraživanja Internet resursa, zaključak je da je trenutno ovo najčešći vid implementacije zaštite. Ostaje da vidimo kako će se kretati trendovi u toj oblasti sa daljim razvojem tehnologije.

LITERATURA

- [1] A. S. Tanenbaum, *Computer Networks*, Pearson, 2010.
- [2] Wei-Meng Lee, *Beginning Android 4 Application Development*, Wrox, 2012.
- [3] Bruce Eckel, *Thinking in Java*, Prentice Hall, 2006.
- [4] Elliotte Rusty Harold, *Java Network Programming*, O'Reilly Media, 2013.
- [5] <http://www.vogella.com/tutorials>
- [6] <http://developer.android.com/guide/index.html>

A.SKRAČENICE

AES – Advanced Encryption Standard
AVD - Android Virtual Device
DES – Data Encryption Standard
GUI - Graphical User Interface
IPSec – IP Security
JDBC - Java Database Connectivity
JDK - Java Development Kit
JRE - Java Runtime Environment
JVM - Java Virtual Machine
NIST – National Institute of Standards and Technology
NSA – National Security Agency
OTP - One Time Password
PAP – Password Authentication Protocol
PKI - Public Key Infrastructure
RADIUS - Remote Access Dial In User Service
RFID - Radio frequency identification
SDK - Software Development Kit
SQL - Structured Query Language
TCP - Transmission Control Protocol
UDP - User Datagram Protocol