# MOBILNE APLIKACIJE

## Vežbe 10

## Mobilne komunikacije

2022/2023

# Sadržaj

# 1. JSON

JSON je format za lakšu razmenu podataka. Podaci se zapisuju kao parovi ključ/vrednost. Ključ se navodi kao tekst pod duplim navodnicima nakon čega sledi vrednost.

"firstName":"John"

JSON vrednosti mogu biti:

- number (integer or floating point)
- string (in double quotes)
- boolean (true or false)
- array (in square brackets)
- object (in curly braces)
- Null

JSON objekat se zapisuje u parovima ključ/vrednost koji se nalaze unutar vitičastih zagrada:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Array sadrži ključ, nakon čega sledi niz elemenata u uglasim zagradama:

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName":"Jones"}
]
```

Na adresi http://www.jsonschema2pojo.org/ mogu se izgenerisati Java klase kopiranjem JSON sadržaja.

Potrebno je ispratiti par koraka:

- U *Package* delu napisati pun naziv paketa u projektu
- Za *Class* name dati naziv klasi koju generišemo
- Za *Source* type izabrati JSON
- Za *Annotation style* izabrati Gson
- [Opciono] *Preview* daje prikaz generisanih klasa
- Zip opcija preuzima zip fajl sa generisanim klasama i paketima koje možete importovati u projekat prostim *Copy-Paste* mehanizmom
- [Opciono] Ostala svojstva (*check boxovi*) mozete izmeniti po potrebi

## 2. REST web servis

Fokus RESTful servisa je na resursima i kako omogućiti pristup tim resursima. Resurs može biti predstavljen kao objekat, datoteka na disku, podaci iz aplikacije, baze podataka itd.

Prilikom dizajniranja sistema prvo je potrebno da identifikujemo resurse i da ustanovimo kako su međusobno povezani. Ovaj postupak je sličan modelovanju baze podataka.

Kada smo identifikovali resurse, sledeći korak je da pronađemo način kako da te resurse reprezentujemo u našem sistemu. Za te potrebe možemo koristiti bilo koji format za reprezentaciju resursa (npr. JSON).

Da bismo poslali zahtev nekom *web servisu* da dobijemo neki sadržaj, moramo napraviti jedan HTTP zahtev. HTTP zahtev se sastoji od nekoliko elemenata:

- <VERB> GET, PUT, POST, DELETE, OPTIONS itd. ili opis šta želimo da uradimo
- <URI> Putanja do resursa nad kojim ce operacija biti izvedena

Nakon svakog HTTP zahteva sledi i HTTP odgovor od servera klijentu tj. onome ko je zahtev poslao. Klijent kao odgovor dobija sadržaj i statusni kod. Ovaj kod nam govori da li je naša operacija izvršena uspešno ili ne. Kodovi su reprezentovani celim brojevima i to:

- *Success* 2xx - sve je prošlo ok.
- *Redirection* 3xx - desila se redirekcija na neki drugi servis.
- *Error* 4xx, 5xx - javila se greška.

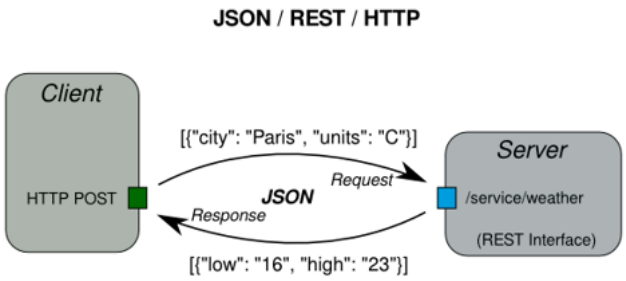
Svaki servis mora imati neku adresu na koju šaljemo HTTP zahtev.

Primer: http://MyService/Persons/1

Ako želimo da izvedemo nekakav upit nad našim servisom, to možemo da uradimo tako što dodamo **?** simbol na kraj putanje. Nakon specijalnog simbola, slede parovi ključ-vrednost spojeni **&** simbolom, ako tih parametara ima više od jednog.

Primer: http://MyService/Persons/1?format=json&encoding=UTF8

Servisi koje pozivamo preko interneta imaju već definisanu strukturu (tačnu putanju, metod kojim se pozivaju, podatke koje očekuju, kako se pretražuju). Jedino što je potrebno da uradimo je da nađemo konkretan servis i pogledamo kako on tačno očekuje da vršimo komunikaciju sa njim.




Slika 1. Primer komunikacije između klijenta i servera

## 2.1 Poziv REST web servisa koristeći Retrofit

Da bismo instalirali *Retrofit* biblioteku potrebno je dodati nekoliko biblioteka u *build.gradle* datoteci.

```
49    implementation 'com.google.code.gson:gson:2.8.7'
50    implementation 'com.squareup.retrofit2:retrofit:2.3.0'
51    implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
52    implementation 'com.squareup.okhttp3:logging-interceptor:3.12.1'
53}
```

Slika 2. Dodavanje biblioteka

Kada su biblioteke instalirane, potrebno je definisati *retrofit* instancu koja će vršiti HTTP zahteve ka određenom REST servisu (slika 3).

```
39    /*
40     * Prvo je potrebno da definisemo retrofit instancu preko koje ce komunikacija ici
41     * */
42    public static Retrofit retrofit = new Retrofit.Builder()
43            .baseUrl(SERVICE_API_PATH)
44            .addConverterFactory(GsonConverterFactory.create(gson))
45            .client(test())
46            .build();
```

Slika 3. Kreiranje *retrofit* instance

Za svaki servis koji želimo da pozovemo moramo da definišemo metod sa kojim se poziva (GET, POST, ...), ali i dodatne parametre upita, dodatne putanje, a u slučaju POST zahteva i sadržaj koji se šalje (u našem primeru String vrednost).

Na slici 4 nalazi se primer *MessageService* interfejsa koji sadrži POST i GET metodu. Metode se označavaju u vidu @POST i @GET, navodi se i koje podatke šaljemo i šta očekujemo kao rezultat.

```
17  * */
18  public interface MessageRestService {
19
20      @Headers({
21              "User-Agent: Mobile-Android",
22              "Content-Type:application/json",
23              "Accept: application/json"
24      })
25      @POST(".")
26      Call<Message> sendMessage(@Body String message);
27
28      @Headers({
29              "User-Agent: Mobile-Android",
30              "Content-Type:application/json",
31              "Accept: application/json"
32      })
33      @GET(".")
34      Call<List<Message>> getMessages();
35  }
```

Slika 4. Interfejs *ReviewerService*

Spisak dostupnih anotacija, koje *retrofit* podržava, se nalazi na slici 5.

| Annotation | Description |
|---|---|
| `@Path` | variable substitution for the API endpoint (i.e. username will be swapped for `{username}` in the URL endpoint). |
| `@Query` | specifies the query key name with the value of the annotated parameter. |
| `@Body` | payload for the POST call (serialized from a Java object to a JSON string) |
| `@Header` | specifies the header with the value of the annotated parameter |

Slika 5. Anotacije i njihova značenja

Ukoliko neki servis, koji želimo da pozovemo, zahteva dodatne elemente, koje treba poslati unutar zaglavlja HTTP zahteva, to možemo definisati koristeći *Header* anotaciju. Svi elementi koji se navode su parovi ključ-vrednost (slika 6).

```
19
20      @Headers({
21              "User-Agent: Mobile-Android",
22              "Content-Type:application/json",
23              "Accept: application/json"
24      })
25      @POST(".")
26      Call<Message> sendMessage(@Body String message);
```

Slika 6. Primer zaglavlja

Servis koji pozivamo izgleda ovako:

`http://<service_ip_adress>:<service_port>/messages`

Konkretno u našem primeru, definišemo putanju servisa kao na slici 7.

```
16  public class ServiceUtils {
17  //    http://<service_ip_adress>:<service_port>/messages
18      //EXAMPLE: http://192.168.43.73:8081/messages
19      public static final String SERVICE_API_PATH = "http://" + BuildConfig.IP_ADDR + ":8081/messages/";
20
```

Slika 7. Definisan server

Nakon definisanja putanje servera, kreiranja *retrofit*-a i definisanja REST servisa, potrebno je izvršiti i instanciranje REST servisa (slika 8).

```
48       * Definisemo konkretnu instancu servisa na intnerntu sa kojim
49       * vrsimo komunikaciju
50       * */
51      public static MessageRestService messageRestService = retrofit.create(MessageRestService.class);
```

Slika 8. Instanciranje MessageRestService-a

U metodi *onCreate* klase *MessageFragment* nalazi se kod koji obrađuje odgovor REST servisa nakon GET poziva (slika 9). Ovim pozivom tražimo listu svih poruka. Nakon prijema poruka, pomoću adaptera, prikazujemo poruke unutar *ListView*-a.

```
59      @Override
60      public void onCreate(Bundle savedInstanceState) {
61
62          super.onCreate(savedInstanceState);
63          Call<List<Message>> messagesList = ServiceUtils.messageRestService.getMessages();
64          messagesList.enqueue(new Callback<List<Message>>() {
65              @Override
66              public void onResponse(Call<List<Message>> call, Response<List<Message>> response) {
67                  messages = (List<Message>) response.body();
68                  Log.d( tag: "REZ", String.valueOf(messages));
69                  Log.i( tag: "REZ",  msg: "PORUKE: " + messages);
70
71                  getActivity().runOnUiThread(new Runnable() {
72                      @Override
73                      public void run() {
74                          adapter = new MessageAdapter(getContext(), messages);
75                          lw.setAdapter(adapter);
76                      }
77                  });
78              }
79
80              @Override
81              public void onFailure(Call<List<Message>> call, Throwable t) {
82                  Log.d( tag: "REZ", msg: "onFailure");
83
84              }
85          });
86
87      }
```

Slika 9. Odgovor REST servisa nakon GET poziva

Sam poziv REST servisa se odvija u pozadini i mi ne moramo da vodimo računa o tome, samo je potrebno da registrujemo Šta da se desi kada odgovor stigne do nas. Taj deo se implementira dodavanjem *Callback<List<Message>>* unutar *enqueue* metode.

U metodi *onResume* klase *MessageFragment* implementiran je odgovor REST servisa na poziv POST metode. POST metodom se šalje tekst poruke koju korisnik kuca unutar *EditText* polja (slika 10).

Kao i kod prethodnog poziva unutar *enqueue* metode nalazi se deo koda koji se poziva nakon odgovora REST servisa. U prikazanom primeru, nakon dobijanja poruke kao odgovora, metoda *notifyDataSetChanged* ažurira adapter kako bi se korisniku osvežila lista prikazanih poruka.

```
89      @Override
90      public void onResume() {
91          super.onResume();
92          View vi = getView();
93          sendMessage.setOnClickListener(v -> {
94              editMessage = vi.findViewById(R.id.editMessage);
95              Log.d( tag: "REZ",editMessage.getText().toString());
96
97              Call<Message> call = ServiceUtils.messageRestService.sendMessage(editMessage.getText().toString());
98              call.enqueue(new Callback<Message>() {
99                  @Override
100                 public void onResponse(Call<Message> call, Response<Message> response) {
101                     Log.d( tag: "REZ", response.body().getText());
102                     messages.add(((Message) response.body()));
103                     adapter.notifyDataSetChanged();
104
105                 }
106
107                 @Override
108                 public void onFailure(Call<Message> call, Throwable t) {
109                     Log.d( tag: "REZ",t.getMessage().toString());
110                 }
111             });
112
113     });
```

Slika 10. Odgovor REST servisa nakon POST poziva

Server je implementiran unutar projekta *server-demo* kao *Spring Boot* aplikacija. Projekat sadrži klasu *Message* sa poljima *_id, text* i *sender*.

Na slici 11. vidi se primer REST kontrolera. Kontroler poseduje implementacije GET i POST metoda.

```
@RestController
@RequestMapping("/messages")
@CrossOrigin
public class MessageController {
    @Autowired
    private MessageService messageService;

    @GetMapping
    public ResponseEntity<?> getMessage(){
        return new ResponseEntity<>(messageService.getMessages(), HttpStatus.OK);
    };

    @PostMapping
    public Message sendMessage(@RequestBody String message){
        System.out.println(message);
        return messageService.sendMessages(message);
    };
}
```

Slika 11. REST kontroler implementiran u Java projektu

Slika 12 prikazuje definisane servise.

```
public class MessageService {
    private static List<Message> messages = new ArrayList<>();
    private static Integer iterator = 0;

    public List<Message> getMessages(){
        return messages;
    }

    public Message sendMessages(String message){
        iterator += 1;
        Message m = Message.builder()._id(Long.valueOf(iterator)).text(message).sender(false).build();
        messages.add(m);
        return m;
    }
}
```

Slika 12. *MessageService* unutar Java projekta

Web aplikacija nalazi se na putanji *http://<service_ip_adress>:<service_port>/index.html* pomoću koje je moguće takođe pozvati definisane endpointe. Aplikacija izgleda kao na slici 13 ispod.


Slika 13. Web aplikacija

## 4. Web socket

*Web Socket* je komunikacijski protokol koji omogućava dvosmernu komunikaciju između klijenata i servera putem jedne TCP veze. U upotrebi sa HTTP protokolom koji koristi zahtev-odgovor arhitekturu, *Web Socket* omogućava stalnu vezu koja omogućava da obe strane šalju podatke jedna drugoj u stvarnom vremenu.

Web Socket-i omogućavaju mobilnoj aplikaciji ažuriranje u stvarnom vremenu. To je korisno za različite vrste aplikacija poput chat aplikacija, igrica i aplikacija za razmenu podataka u stvarnom vremenu.

U aplikaciji *socket-server*, koja je kreirana u *node.js-u*, implementiran je jednostavan *WebSocket* server koji koristi *Express.js* i *Socket.IO* biblioteke.

Način instalacije biblioteka i pokretanje objašnjen je u *README.md* datoteci.

Na slici 14 se u 2 liniji importuje biblioteka *socket.io* a u 10 liniji se inicijalizuje *socket server*.

server.js > ...

```
1   const express = require('express');
2   const socket = require('socket.io');
3   const fs = require('fs');
4   const app = express();
5   const port = 3000;
6
7   const server = app.listen(port);
8   app.use(express.static('public'));
9   console.log('Server is running');
10  const io = socket(server);
11  var count = 0;
12
13  var List = require("collections/list");
14
15  class Message {
16      constructor(_id, text, sender) {
17        this._id = _id;
18        this.text = text;
19        this.sender = sender;
20      }
21    }
22
23  var messages = new List([]);
```

Slika 14: Inicijalizacija socket servera

Sledeći deo koda (slika 15) izvršava se svaki put kada se uspostavi nova *Web Socket* veza sa serverom. U ovom bloku koda definišu se događaji (eventi) koje server osluškuje od klijenata.

```
25
26  io.on('connection', (socket) => {
27      console.log("New socket connection: " + socket.id);
28
29      socket.on('counter', () => {
30          count++;
31          console.log("counter " + count);
32          io.emit('counter', count);
33      })
34
35      socket.on('message', (messageText) => {
36          var randomBoolean = Math. random() >= 0.5;
37          messages.add(new Message(messages.length, messageText, randomBoolean));
38          console.log(messages.toJSON());
39          io.emit('message', messages.toJSON());
40      })
41  })
42
```

Slika 15: Osluškivanje događaja od klijenata na serveru

Prvi događaj je *counter***.** Kada se događaj *counter* emituje sa klijenta, server povećava varijablu *count* za 1 i šalje događaj svim povezanim klijentima, šaljući im trenutnu vrednost *count* varijable.

Drugi događaj je *message*. Kada se događaj *message* emituje sa klijenta, server stvara novu instancu *Message* klase sa primenjenim tekstom poruke i nasumično generisanoj boolean vrednosti polja *sender*. Poruka se dodaje u listu *messages*, a zatim se emituje message događaj svim povezanim klijentima, šaljući im trenutno stanje liste *messages* pretvorenu u JSON format.

Kreirana je mobilna aplikacija *Vezbe10_2* koja predstavlja socket klijenta.

Kako bi klijent (mobilna aplikacija) i server (*node.js* aplikacija) mogli da ostvare komunikaciju, potrebno je povezati ih na istu mrežu. Pronaći IP adresu uređaja gde je server pokrenut. Preko IP adrese i porta možemo da pristupimo serveru. Primer: http://192.168.0.19:3000 .

Kako IP adresu ne želimo da postavimo na git potrebno je da je sačuvamo u *local.properties* datoteku koju ne postavljamo na git repozitorijum (slika 16).

```
## This file is automatically generated by Android Studio.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#
# This file should *NOT* be checked into Version Control Systems,
# as it contains information specific to your local configuration.
#
# Location of the SDK. This is only used by Gradle.
# For customization when using a Version Control System, please read the
# header note.
sdk.dir=/home/svetlana/Android/Sdk
ip_addr=192.168.0.19
```

Slika 16: *local.properties* datoteka

Kako bismo dalje u kodu mogli da koristimo ovu *ip_addr* varijablu moramo da izvršimo sledeću konfiguraciju u *build.gradle* datoteci. Linije koda od 5-9 i 20 (slika 17).

```
plugins {
    id 'com.android.application'
    id 'com.google.gms.google-services'
}
def getIpAddress() {
    Properties properties = new Properties()
    properties.load(project.rootProject.file('local.properties').newDataInputStream())
    return properties.getProperty("ip_addr");
}
android {
    namespace 'com.example.vezbe10_2'
    compileSdk 32

    defaultConfig {
        applicationId "com.example.vezbe10_2"
        minSdk 30
        targetSdk 32
        versionCode 1
        versionName "1.0"
        buildConfigField "String", "IP_ADDR", "\""+getIpAddress()+"\""
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

```

Slika 17: *build.gradle* datoteka

Nakon toga možemo da pristupimo našem parametru preko *BuildConfig* klase na sledeći način *BuildConfig.IP_ADDR* (slika 18)**.** Na slici je takođe prikazana *SocketHandler* klasa, u kojoj je kreirana statička promenljiva *socket* i implementirane *getSocket* i *setSocket* metode za tu

promenljivu. Metoda *setSocket()* koristi *IO.socket* metodu za stvaranje *Socket.IO* instance. U ovom slučaju, koristi se IP adresa servera i port. Koristi se try-catch blok kako bi se uhvatila moguća greška.

```
9   public class SocketHandler {
10      static Socket socket;
11
12      public static void setSocket(){
13          try {
14              socket = IO.socket( uri: "http://" + BuildConfig.IP_ADDR + ":3000");
15          }catch (Exception e){
16
17          }
18      }
19
20      public static Socket getSocket() { return socket; }
23
24  }
```

Slika 18: *SocketHandler* klasa

Na slici 19 je data *MainActivity* klasa, gde su u *onCreate* metodi pozvane *setSocket* i *getSocket* metode iz klase *SocketHandler*.

Od 36 do 46 linije je socket postavljen da osluškuje događaje koji se dešavaju, ukoliko se desi događaj *counter***,** pročitaće se nulti argument i smestiće se u neko tekstualno polje na UI niti.

U liniji 32, 33, 34 postavlja se *setOnClickListener* na dugme *button2*, kada se klikne dugme emitovace se događaj *counter*.

```
16  public class MainActivity extends AppCompatActivity {
17      public static Socket socket;
18
19      @Override
20      protected void onCreate(Bundle savedInstanceState) {
21          super.onCreate(savedInstanceState);
22          setContentView(R.layout.activity_main);
23
24          TextView tw2 = findViewById(R.id.textView2);
25          Button button = findViewById(R.id.button1);
26
27          SocketHandler.setSocket();
28
29          socket = SocketHandler.getSocket();
30          socket.connect();
31
32          button.setOnClickListener(v -> {
33              socket.emit( event: "counter");
34          });
35
36          socket.on( event: "counter", args -> {
37              if(args[0] != null){
38                  Integer s = (Integer) args[0];
39                  runOnUiThread( new Runnable() {
40                      @Override
41                      public void run() {
42                          tw2.setText(s.toString());
43                      }
44                  });
45              }
46          });
47
48          FragmentTransition.to(MessageFragment.newInstance(), activity: this, addToBackstack: true, R.id.fragmet1);
49
```

Slika 19: *MainActivity* klasa

Na slici 20 prikazano je kako se na *setOnClickListener* emituje *message* i salje serveru poruka u tekstualnom formatu (linija 62) koja je preuzeta sa elementa *editMessage*.

Na slici 21 postavljen je socket da osluškuje događaje koji se dešavaju, ukoliko se desi događaj *message***,** pročitaće se nulti argument, transformisace se u listu poruka i smestiće se u *listView* element preko *MessageAdaptera*.

```
50      @Override
51  @   public View onCreateView(LayoutInflater inflater, ViewGroup vg, Bundle data) {
52          setHasOptionsMenu(true);
53          View vi = inflater.inflate(R.layout.fragment_message_list, vg,  attachToRoot: fal
54
55          adapter = new MessageAdapter(getContext(), new ArrayList<Message>());
56          editMessage = (EditText) vi.findViewById(R.id.editMessage);
57          sendMessage = vi.findViewById(R.id.button2);
58          lw = vi.findViewById(R.id.messages_list_view);
59          lw.setAdapter(adapter);
60
61          sendMessage.setOnClickListener(v -> {
62              MainActivity.socket.emit( event: "message", editMessage.getText());
63          });
64
65          return vi;
66      }
67
68
```

Slika 20: Emitovanje message socket-a

```
70      @Override
71      public void onCreate(Bundle savedInstanceState) {
72          super.onCreate(savedInstanceState);
73          MainActivity.socket.on( event: "message", args -> {
74              if(args[0] != null){
75                  Log.i( tag: "SOCKET", args[0].toString());
76
77                  List<Message> messages = convertJsonToListOfObject(args[0]);
78                  Log.i( tag: "SOCKET",  msg: "PORUKE: " + messages);
79
80                  getActivity().runOnUiThread(new Runnable() {
81                      @Override
82                      public void run() {
83                          adapter = new MessageAdapter(getContext(), messages);
84                          lw.setAdapter(adapter);
85                      }
86                  });
87              }
88          });
89      }
```

Slika 21: Postavljanje socket-a da osluškuje događaje vezane za message

## 5. Domaći

Domaći se nalazi na *Canvas-u (canvas.ftn.uns.ac.rs)* na putanji *Vežbe/10 Zadatak.pdf*

Primer *Vezbe10_1 i Vezbe10_2* možete preuzeti na sledećem linku:
https://gitlab.com/antesevicceca/mobilne-aplikacije-sit

Za dodatna pitanja možete se obratiti asistentima:

- Svetlana Antešević (svetlanaantesevic@uns.ac.rs)
- Jelena Matković (matkovic.jelena@uns.ac.rs)