DA: Retrofit - Connecting with the Backend

Applications for mobile devices - Theory - Unit 5

Didac Florensa Cazorla

Any: 2021-2022
Curs: 102386
Institut: University of Lleida (Campus Igualada)
Titulació: Bachelor's degree in Digital Interaction and Computing Techniques (GTIDIC)

Agenda

Introduction

Connecting with the Backend

Retrofit

HandsOn: Trivial

Introduction

Warm-up

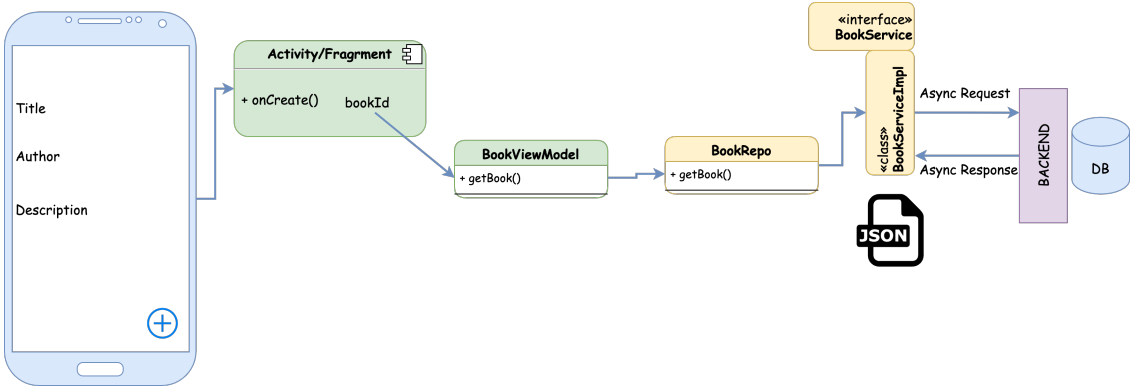
- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

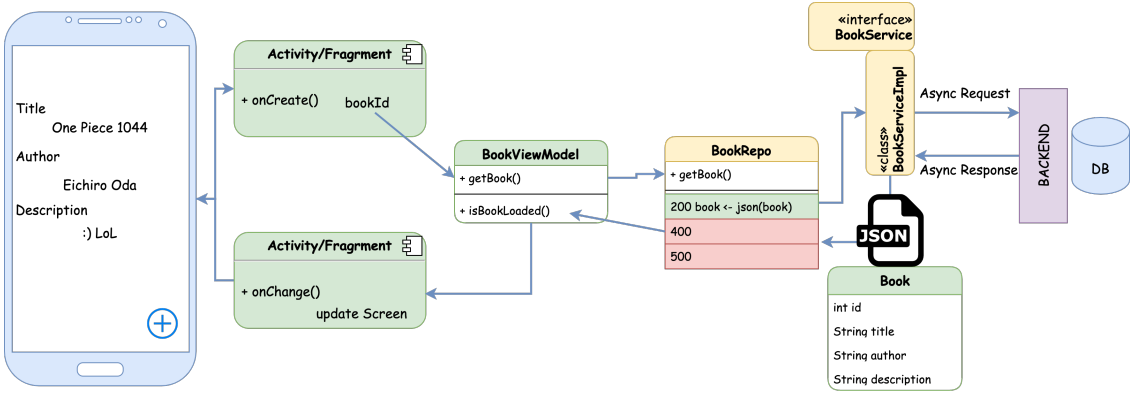
Connecting with the Backend

What is the problem?

Client - Backend (Communication)



Backend - Client (Communication)



To solve the problem... we need to learn...

Asynchronous/synchronous

Asynchronous requests should be preferred to **synchronous** requests for performance reasons. **Synchronous** requests block the execution of code, which causes *freezing* on the screen and an unresponsive user experience.

Mapping

Mapping JSON response to class instances in java

Flow

How can we organize to let the information flow straightforwardly?

Retrofit

What is Retrofit?

Overview

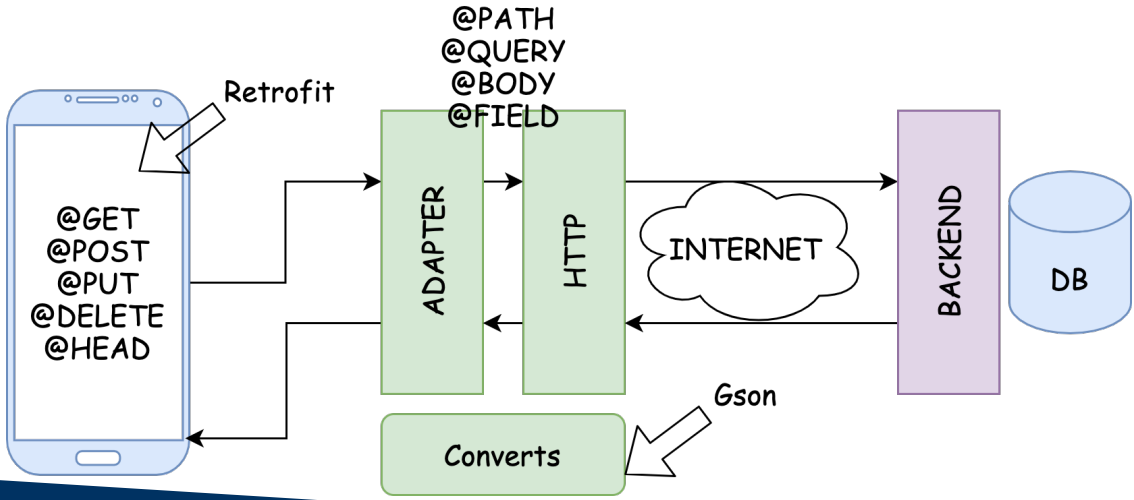
Retrofit is a *REST Client for Java and Android* by Square inc under Apache 2.0 license. It's an *easy and fast library* to retrieve and upload the data (*JSON or other structured data*) via a REST-based web service. It's a simple network library that is used for network transactions. We can seamlessly capture JSON responses from web service/web API using this library.

⇒ This library makes downloading **JSON or XML** data from a web API reasonably straightforward. Once the data is downloaded, it is parsed into a *Plain Old Java Object (POJO)*, which must be defined for each "resource" in the response.

Book

- Book json ⇒ is transformed into ⇒ Book class ⇒ `Book book = new Book()`
- $\forall key, value \in json$ ⇒ `book.setKey(value)`

How does Retrofit fit your goal?



How does it work?

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user")
    String user);
}
```

In this example, we are interfacing a **GET** operation asking for all the repositories for a specific user. Note the **{user}** is a variable, and it can be set in *runtime*. Moreover, this interface indicates that the JSON object received from the request will be a map to a JAVA data struct (a list) of the class Repo. We can map a **JSON** to an **Object** automatically and transparently.

Essential Annotations

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

Headers

```
@Headers({"Cache-Control: max-age=640000", "User-Agent:{}"})
@GET("xxxxxxxxx")
```

POST and Forms

Adding Forms

```
@FormUrlEncoded
@POST("xxxxxxx")
Observable<SomeResponse> getSth(@Field("property") String property);
```

MultiForms

```
@Multipart
@POST("xxxxxxx")
Call<Response> uploadImage(@Part("description")
        String description, @Part("image") RequestBody image)
```

Book sample

```java
public class Book {
    @SerializedName("id")
    private Integer id;
    @SerializedName("tittle")
    private String tittle;
    @SerializedName("description")
    private String description;
    @SerializedName("author")
    private String author;
```

```json
{
    "tittle": "One Piece 1044",
    "author": "Eichiro Odo",
    "description": ":) Lol"
}
```

```java
public interface BookService {
  @GET("/book/show")
  Call<Book> getBookById(@Query("id")
  int id);
}
```

The retrofit will send the petition to the backend via the internet. We ask the backend to obtain information of a specific book (*/book/show*), and we use a query parameter **@Query("id")**. Our backend will search in the DB and generate the **JSON**, once retrofit receives the JSON, then uses gson to map each property to an instance of the class Book! ⇒ Note: The **@SerializedName("id")** needs to be the same as the JSON key.

Book sample (Book Service)

```
public interface BookService  {
    @GET("/book/show")
    Call<Book> getBookById(@Query("id") int id);
    @POST("/book/create")
    Call<ResponseBody> createBook(@Body Book book);
    @PUT("/book/update")
    Call<ResponseBody> updateBookById(@Query("id") int id,
            @Body Book book);
    @DELETE("/book/delete")
    Call<ResponseBody> deleteBookById(@Query("id") int id);
    @GET("/books")
    Call<List<Book>> getBooks(@Header("Authorization") String token);
}
```

What do we need to use it?

Add dependency file in gradle:

```
compile 'com.squareup.retrofit:retrofit:@latest'
```

Add Internet Permission in the AndroidManifest.xml:

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

Build it and connect to the API

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();
```

Assumptions with our development environment (1)

127.0.0.1

127.0.0.1 is our localhost. We deploy our backend on our real host. But, the client (android app) is running on the SIMULATOR, another *operating system*..., so **127.0.0.1** is not our computer is the localhost of SIMULATOR...

:(and now... OMG, what can I do?

Don't worry, guys. Google knows that and provides you with a solution **10.0.2.2**. This IP is mapping to your host (localhost). Try it, open chrome on the phone (SIMULATOR) and navigate to 10.0.2.2:8080.

Amazing is our DB :)

We need to take care of these network configurations. But wait... What would you do with this knowledge if you want to test using a real android phone instead of connecting using the simulator? Now, what is the solution?

Assumptions with our development environment (2)

10.0.2.2

10.0.2.2 or 127.0.0.1 is not an actual domain, so Android will blame you. We need to tell them to trust it.

Create this file under *res/xml/network-security-config.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">10.0.2.2</domain>
    </domain-config>
</network-security-config>
```

Using Retrofit in our project

```
public class RetrofitClient {
    private static Retrofit retrofit;
    private static final String BASE_URL = "http://10.0.2.2:8000";
    private static OkHttpClient client = new OkHttpClient.Builder().build();
    public static Retrofit getRetrofitInstance() {
        if (retrofit == null) {
            retrofit = new retrofit2.Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create())
                    .callFactory(client) .build();}
        return retrofit; } }
```

- **Builder Pattern**: to separate the construction of a complex object from its representation.
- **Factory Pattern**: Factory is an abstract factory (*GsonConverterFactory*).
- **Singleton Pattern**: to ensure that only one instance in memory belongs to the creation mode.

Converters

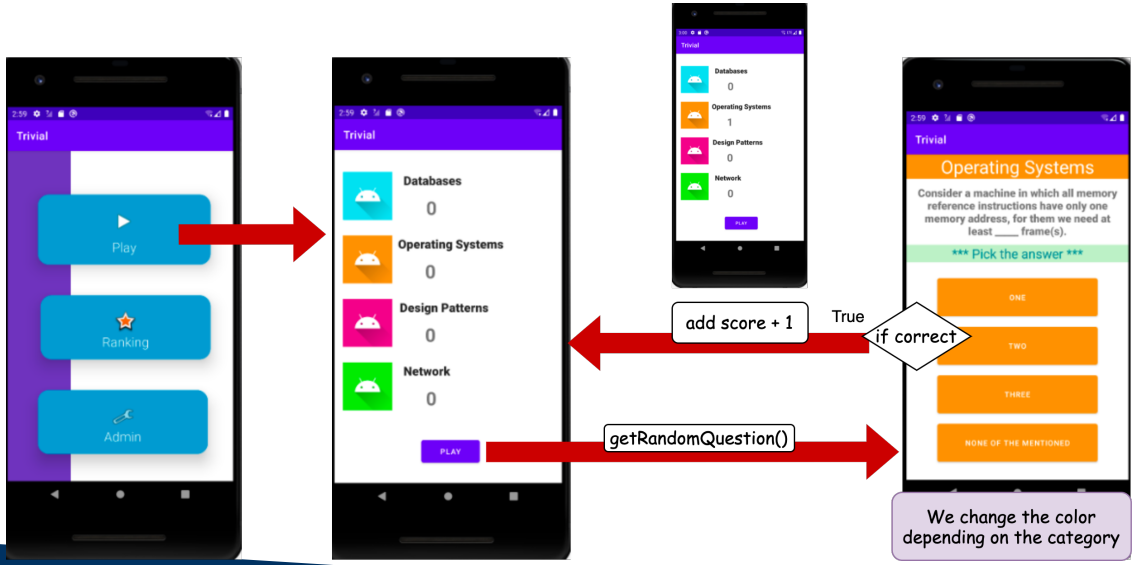
Converter	Library
Gson	com.squareup.retrofit2:converter-gson:2.9.0
Jackson	com.squareup.retrofit2:converter-jackson:2.9.0
Moshi	com.squareup.retrofit2:converter-moshi:2.9.0
Protobuf	com.squareup.retrofit2:converter-protobuf:2.9.0
Wire	com.squareup.retrofit2:converter-wire:2.9.0
Simple XML	com.squareup.retrofit2:converter-simplexml:2.9.0

HandsOn: Trivial

HandsOn: Trivial

StartKit - https://github.com/102386-DAM/TrivialApp-HandsON

HandsOn: Trivial - Goal



HandsOn: Trivial - Starter Kit - Question Model

```
class Question(SQLAlchemyBase, JSONModel):
    __tablename__ = "questions"
    id = Column(Integer, primary_key=True)
    question = Column(UnicodeText)
    category = Column(Enum(CategoryEnum))
    answers = relationship("AnswerQuestionAssiation")

    @hybrid_property
    def json_model(self):
        return {
            "id": self.id,
            "question": self.question,
            "category": self.category.value,
        }
```

HandsOn: Trivial - Starter Kit - Answer Model

```python
class Answer(SQLAlchemyBase, JSONModel):
    __tablename__ = "answers"

    id = Column(Integer, primary_key=True)
    answer = Column(UnicodeText)

    @hybrid_property
    def json_model(self):
        return {
            "id": self.id,
            "answer": self.answer,
        }
```

HandsOn: Trivial - Starter Kit - Relationship (Q&A)

```
class AnswerQuestionAssiation(SQLAlchemyBase, JSONModel):
    __tablename__ = "answer-question-association"

    id_question = Column(Integer, ForeignKey("questions.id",
        onupdate="CASCADE", ondelete="CASCADE"),
        nullable=False, primary_key=True)

    id_answer = Column(Integer, ForeignKey("answers.id",
        onupdate="CASCADE", ondelete="CASCADE"),
        nullable=False, primary_key=True)

    is_correct = Column(Boolean, nullable=False)
    answers = relationship("Answer")
```

HandsOn: Trivial - Starter Kit - Backend

```
# GET
application.add_route("/trivial/question", trivial_events.ResourceGetRandomQuestion())
```

DB init

The DB is initialized by a text file obtained using a web-scrapper.

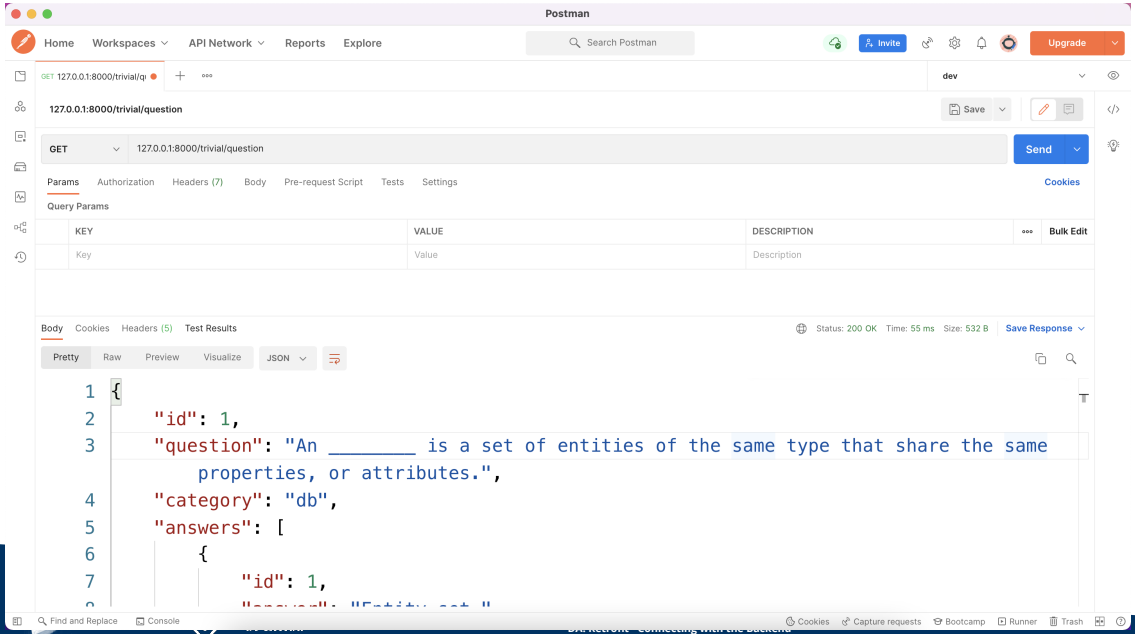
Be careful from now on

This starter kit is the same as your DamCore project. So, to use it, please do `docker system prune -a`, or enter to the DB (adminer) and drop all the tables manually. The container is created expecting a database model that is now wholly different (structure) from the one you have... and then problems will arise if you don't do it.

!!!! Again remember

Do it again `docker system prune -a` once you finish and want to work on your project.

HandsOn: Trivial - First Milestone



HandsOn: Trivial - JSON Body


```
{
    "id": 9,
    "question": "The minimum number of frames to be allocated to a process is decided by the ____________",
    "category": "os",
    "answers": [
        {
            "id": 33,
            "answer": "the amount of available physical memory",
            "is_correct": false
        },
        {
            "id": 34,
            "answer": "operating System",
            "is_correct": false
        },
        {
            "id": 35,
            "answer": "instruction set architecture",
            "is_correct": true
        },
        {
            "id": 36,
            "answer": "none of the mentioned",
            "is_correct": false
        }
    ]
}
```

HandsOn: Trivial - (Review Starter Kit - Android code)

Code revision

Check during 10 minutes the code provided (exam simulation) and understand what is doing.

Pay attention

Please go through the code and detect the files needed in your project. These are the following files, the ones we discussed in this session that it is required to communicate with the backend.

1. Retrofit Client Class
2. Android Manifest
3. XML (Network)

HandsOn: Trivial - Coding the Models (Category)

```
public enum Category {
    net,os,db,patterns;
    @NonNull
    @Override
    public String toString() {
        switch (this){
            case db: return "Databases";
            case os: return "Operating Systems";
            case net: return "Network";
            case patterns: return "Design Patterns";
            default: return "";
        }}
    public int getColor() {
        switch (this){
            case db: return R.color.trivial_blue;
            case os: return R.color.trivial_orange;
            case net: return  R.color.trivial_green;
            case patterns: return  R.color.trivial_pig;
            default: return R.color.black;
        }}
}
```

HandsOn: Trivial - Coding the Models (Category)

Backend model for the categories

```
class CategoryEnum(enum.Enum):
    db = "db"
    os = "os"
    net = "net"
    patterns = "patterns"
```

⇒ We will create the same `Enum` in our Android App to make the mapping easy.

HandsOn: Trivial - Coding the Models (Question)

```java
public class Question {
    @SerializedName("question")
    String question;
    @SerializedName("answers")
    List<Answer> answers;
    @SerializedName("category")
    Category category;

    public Question(String question, List<Answer> answers) {
        this.question = question;
        this.answers = answers;
    }

    public String getQuestion() {return question;}
    public void setQuestion(String question) {
        this.question = question;}

    public List<Answer> getAnswers() {return answers;}
    public void setAnswers(List<Answer> answers) {
        this.answers = answers;}

    public Category getCategory() {return category;}
    public void setCategory(Category category) {
        this.category = category;}
}
```

HandsOn: Trivial - Coding the Models (Question)

```java
public class Answer {

    @SerializedName("answer")   // This is the key in the json for the property. It must be the same.
    String answer;              // This could be any word. It is not required for the mapping.
    @SerializedName("is_correct")
    Boolean isCorrect;

    public Answer(String answer, Boolean isCorrect) {
        this.answer = answer;
        this.isCorrect = isCorrect;
    }

    public String getAnswer() {return answer;}
    public void setAnswer(String answer) {
        this.answer = answer;}

    public Boolean getCorrect() {return isCorrect;}
    public void setCorrect(Boolean correct) {
        this.isCorrect = correct;
        }
}
```

HandsOn: Trivial - QuestionService

Inteface

```
public interface QuestionService {
    @GET("trivial/question")
    Call<Question> getRandomQuestion();
}
```

Implementation

```
public class QuestionServiceImpl implements QuestionService{
    private  Retrofit retrofit = RetrofitClientInstance.getRetrofitInstance();
    @Override
    public Call<Question> getRandomQuestion() {
        return retrofit.create(QuestionService.class).getRandomQuestion();
    }
}
```

HandsOn: Trivial - QuestionsRepo

```java
public class QuestionRepo {

    private static String TAG = "QuestionsRepo";

    private QuestionService questionService;
    public MutableLiveData<Question> mQuestion;

    public QuestionRepo() {
        this.questionService = new QuestionServiceImpl();
        this.mQuestion = new MutableLiveData<>();
    }

}
```

⇒ The response is stored using MutableLiveData to take advantage of the observer pattern and trigger the modifications once we get the answer from the backend.

⇒ Note that our view model will provide our view to observe this variable (**mQuestion**). It can be public or obtained by coding a *getter*.

HandsOn: Trivial - QuestionRepo *getQuestion*

```
public void getQuestion() {

        questionDAO.getRandomQuestion().enqueue(
            new Callback<Question>() {
            @Override
            public void onResponse(Call<Question> call,
            Response<Question> response) {
                int code = response.code();
                if (code == 200) {
                    Question q = response.body();
                    assert q != null;
                    mQuestion.setValue(q); // Setting the MUTABLE!!!!!!
                }
            }

            @Override
            public void onFailure(Call<Question> call, Throwable t) {
                Log.d(TAG,  t.getMessage());
            }
        });
    }
```

HandsOn: Trivial - QuestionViewModel

```java
public class QuestionViewModel extends ViewModel {

    private final QuestionRepo questionRepo;
    public MutableLiveData<Answer> userAnswer;

    public QuestionViewModel() {
        this.questionRepo = new QuestionRepo();
        this.userAnswer = new MutableLiveData<>();
    }

    // We trigger the repo to ask the backend the information
    public void getQuestion(){
        this.questionRepo.getQuestion();
    }

    // We provide LiveData from the repo
    public LiveData<Question> isQuestionLoaded(){
        return this.questionRepo.mQuestion;
    }

    public void onClickedAt(Answer answer){
        userAnswer.setValue(answer);
    }
}
```

HandsOn: Trivial - Question Activity

```java
public class QuestionActivity extends AppCompatActivity {
    QuestionViewModel viewModel;
    ActivityQuestionBinding activityQuestionBinding;
    Question question;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        initDataBinding();
        initGame();
    }

    public void getPlayerAnswer(Answer answer){
        Log.d(this.getClass().getSimpleName(), "IsCorrect: "+answer.getCorrect());
    }
```

HandsOn: Trivial - Question Activity

```
public class QuestionActivity extends AppCompatActivity {
    public void initGame(){
        viewModel.getQuestion();
        viewModel.isQuestionLoaded().observe(this, new Observer<Question>() {
            @Override
            public void onChanged(Question question) {
                Log.d(this.getClass().getSimpleName(), "questionLoaded.");
            }
        });

        viewModel.userAnswer.observe(this, new Observer<Answer>() {
            @Override
            public void onChanged(Answer answer) {
                getPlayerAnswer(answer);
            }
        });
    }
```

HandsOn: Trivial - Question Activity Layout

```
<TextView android:text=
  "@{questionViewModel.isQuestionLoaded().category.toString()}" />
<TextView android:text=
  "@{questionViewModel.isQuestionLoaded().question}"/>
<Button android:text=
"@{questionViewModel.isQuestionLoaded().answers.get(0).answer}"
android:onClick=
"@{() -> questionViewModel.onClickedAt(questionViewModel
.isQuestionLoaded().answers.get(0))}" />
...
<Button android:text=
"@{questionViewModel.isQuestionLoaded().answers.get(3).answer}"
android:onClick=
"@{() -> questionViewModel.onClickedAt(questionViewModel
.isQuestionLoaded().answers.get(3))}" />
```

That's all

QUESTIONS?

About me

www — jordimateofornes.com
github — github.com/JordiMateo
twitter — @MatForJordi
gdc — Distributed computation group