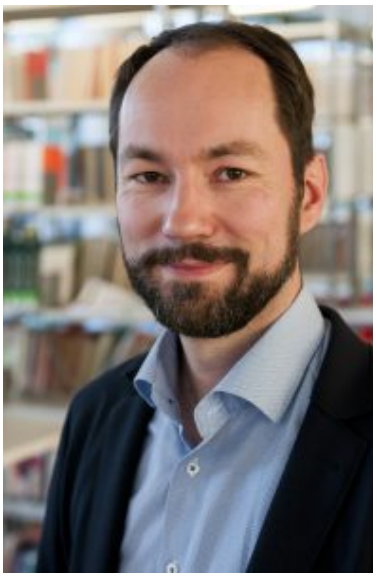
# Hector-School

Exercise



**Prof. Dr. Thomas Schuster**

**Jan Christoph**

**Simon Schmid**

# Inhalt

# Quizmaster-App

## Introduction

Your task will be to develop an app that utilizes Bluetooth low energy (BLE) connections. As a result, our app should recognize Bluetooth beacons and respond to the recognition with a quiz (either present questions or possible answers to the user). Thus, we will create a quiz based on and about IoT technologies. To provide your app with questions, the app will access a web service. The web service also delivers the functionality to check your answer of choice for correctness.

The functionality of our app will be as follows: we will have many clients (your devices running the app) and a web service, which is utilized to respond to the IoT events, issued by the iBeacons. The web service provides different operations to collect and process data:

- registration (you as a quiz player; with a unique username),
- provide questions to your app,
- provide answers for each question (three answers may be collected for each question), and
- to check your suggested answer.

We set up a network of four iBeacons within this room. Each iBeacon has its own configuration and sends out a BLE signal. It will be your task to automate the logic of the quiz app by sending requests to the web service when you discovered a new beacon.

To develop the frontend and make the app cross platform ready, we are going to use the Ionic framework. Therefore, we provide a base project on GitHub. It contains the basic structure that can be used to develop and complete the quizmaster app.

In the following chapter Exercise – Frontend Development you will find the exercises and some useful hints to develop the quiz app.

# Exercise – Frontend Development

## Task 0: Read the content found under app

In order to be able to start with the tasks, the starter template of the app must first be loaded. Further information on accessing the web service and other tips can also be found in the app chapter.

## Task 1: Get iBeacon Data!

In the page *home* you need to scan for the iBeacons. We suggest using the Ionic iBeacons plugin. Please follow the guide on https://ionicframework.com/docs/native/ibeacon/ to install the plugin and to find an example (how to use it). You can copy the example on the website and change it to your own needs for this task.

It's most suitable to expand your application logic in the home page for this task. You should create the code for the function *handleBeaconDiscovered*, which should be executed when you have detected a beacon. Within this function you can decide to either call the function *getQuestion* for getting a new question or *getAnswer* for getting a new answer. Each of the two functions should send its own requests to the web service and process the result of the web service. The HTTP provider already contains fully functional methods to send web service requests for the given task. You can just call these functions from home.ts.

With the detection of the first Beacon you should request a question from the web service, the next Beacon should trigger to get an answer for the previously received question.

Basically, you can stick to the order of execution for web service calls just like in the user page. Handling the requests of the web service also contains handling the response and assigning the received values to the variables *data* (contains meta information for each answer. Information handled: flag if answer was found already, flag if answer was checked by the user as correct, answer ID to check for approval with the web service) and *a1*, *a2* and *a3* (each variable contains the answer text of one answer, which will be displayed in home.html) in home.ts.

With the code given and your knowledge about iBeacon monitoring and sending HTTP requests, you will be able to further develop your app and retrieve question-answer tuples.

### Task 2: Answer Check

Finally: Check if your suggested answer is correct. Implement a function to send a HTTP request to the web service. With the given parameters the web service will check whether the answer is right or not and increase or decrease your score. The result of the check will be sent to your app. The button to trigger the check is also located in home.ts.

When the user found the right answer, don't forget to call the functions *resetQuestion*, *resetAnswers* and *setScore* in home.ts to reset the game.

### Bonus task: High Score Page

If this was too easy for you, you may also create a high score page which displays all teams and their score. Remember to call the web service to get the high scores.

## App

Edit the code of the app and run it on your mobile device. For this purpose, you must get the project on your computer. The code of the app is available on GitHub under https://github.com/futureLABHsPforzheim/quizmaster_start.

To download the code to your current directory on your computer, open your command line tools (under Windows press *Win + R*, type in "cmd" and press "OK", under Mac press *CMD + space*, search for and execute "Terminal" to open the command line tools) and execute:

```
git clone https://github.com/futureLABHsPforzheim/quizmaster_start
```

This command creates a folder called "quizmaster_start" and puts all downloaded files from the GitHub project in this folder. When finished, your command line tools should print:

```
Cloning into 'quizmaster_start'...
remote: Enumerating objects: 125, done.
remote: Counting objects: 100% (125/125), done.
remote: Compressing objects: 100% (110/110), done.
remote: Total 125 (delta 12), reused 108 (delta 4), pack-reused 0
Receiving objects: 100% (125/125), 2.13 MiB | 456.00 KiB/s, done.
Resolving deltas: 100% (12/12), done.
```

The structure of the quizmaster_start folder will look like this:

```
quizmaster_start/
├── resource/
│   └── ...
├── src/
│   ├── app/
│   │   ├── app.component.ts
│   │   ├── app.html
│   │   ├── app.module.ts
│   │   ├── app.scss
│   │   └── main.ts
│   ├── assets/
│   │   └── ...
│   ├── model/
│   │   └── team.ts
│   ├── pages/
│   │   ├── home/
│   │   │   ├── home.html
│   │   │   ├── home.scss
│   │   │   └── home.ts
│   │   └── user/
│   │       ├── user.html
│   │       ├── user.scss
│   │       └── user.ts
│   ├── providers/
│   │   └── http/
│   │       └── http.ts
│   ├── theme/
│   │   └── ...
│   ├── index.html
│   ├── manifest.json
│   └── service-worker.js
├── .editorconfig
├── .gitignore
├── config.xml
├── ionic.config.json
├── package.json
├── package-lock.json
├── README.md
├── start.bat
├── tsconfig.json
└── tslint.json
```

You can find the relevant code for your further tasks in the folder *src*. The app consists of two pages *user* and *home*. Each page consists of three files:

The **.html** file is the declarative file of the page which describes the elements to display in the app. They determine which elements will be shown in which order and what data and interaction elements they contain.

The **.scss** file can be used to determine the styling of the elements within the .html file. This file can be ignored for this Seminar, as it only changes the look of the HTML elements. You can check https://ionicframework.com/docs/theming/ for further information about how to style your Ionic app.

The **.ts** file contains the logic of your page. It determines for example what should be done when the user clicks on a graphical element in your app.

### User page

Location: src/pages/user

This page is the first page you will see when starting the app. It consists of an input field to enter your name and a button to request the creation of your entered user name (.html file). The button's property *(click)* determines which function to execute when clicking the button.

You can find the function *getName* in the file user.ts of this page. It uses the HTTP provider (see below) to send a HTTP request to the web service for creating the user. Furthermore, the subscribe addition determines what your app will do when it received the response from the web service. In case of success ( 'response => {…}') your app will guide you to the home page, in case of an error ('error => {…}') an alert will be shown and you will not be guided to the home page.

### Home page

Location: src/pages/user

The home page is your main point for interaction. It consists of the following elements:

- Textfield to either display a question or a hint to search for beacons. To differentiate which text to display this element is conditionally bound to a variable in the home.ts file (**ngIf="qReady"*)
- Textfield and Checkbox for each answer. The textfield shows the answer text, the checkbox can be checked by the user when he thinks the answer is the right one. Again, the visibility of these elements are bound to a variable in the home.ts file and makes each textfield and checkbox only visible, when the corresponding answer has been found by the user (**ngIf="data.answer0.found"*)
- Button to check whether the selected answer was the correct answer. Again, the function to execute when tabbing the button is in the home.ts file. The function *proofResult* again must call the web service to execute the answer check. It will be your challenge to write the code for the web service calls in Task 2.

This page also should be extended by the logic for the iBeacon handling (Task 1).

### HTTP provider

Location: src/providers/http/http.ts

The provider will be the interface to interact with the web service. In comparison to a page, a provider only contains a .ts file. Function wise it is only a shared service to execute HTTP requests and will be used by multiple components of your app. Thus, providers will have no graphical elements to display data, but support your pages with the data they request.

This provider must be extended to send a request to the web service for checking an answer for correctness (Task 2). The function *proof* should perform the requests. To send a request, you must call the web service URL (variable *web service* in http.ts) and append the port and the parameters in the correct order (see chapter RESTful services) to check your answer. You can also look at functions like *getUserId* or *getQuestion* in http.ts to see how a request is basically done.

In order to run the app several code libraries must be downloaded first. The libraries are listed in the file package.json in your project folder. They contain for example the Ionic framework itself, which is needed to build and execute the app later. To download all necessary libraries execute:

```
cd quizmaster_start
npm install
```

The command "cd quizmaster_start" first changes your current location to the project folder. Afterwards the command "npm install" installs the libraries. This task will create new folder "node_modules" within the project folder which contains the libraries.

When the download has finished, the system will respond with

```
added 750 packages from 682 contributors and audited 3607 packages in 45.513s
found 6 vulnerabilities (1 low, 5 moderate)
  run `npm audit fix` to fix them, or `npm audit` for details
```

The vulnerabilities occure due to updated libraries. You can ignore the warnings, as the application was developed and will be running with the old library versions.

At several points in the code you will find some useful comments where to add code.

The questions for the app are stored in the web service. To fetch a question from the web service, your app must listen to Bluetooth signals sent by iBeacons. When your app discovers a new beacon, this triggers the request of a new question first and for each additional spotted beacon the request of an additional question. This should be repeated until you found all beacons.

The beacons will be placed within this room and send a signal via BLE every second.

To receive a question an answer, you will have to find one of our iBeacons. Each iBeacon can only trigger one web service request per question-answer tuple. The latter means that you will have to collect all answers for a single question by finding all iBeacons.

## iBeacons

The provided iBeacons configurated as followed:

| Name | Major | Minor | UUID | Range |
|---|---|---|---|---|
| **ibeacon_hector_1** | 1 | 1 | E2C56DB5-DFFB-48D2-B060-D0F5A71096E0 | ~ 2m |
| **ibeacon_hector_2** | 2 | | | |
| **ibeacon_hector_3** | 3 | | | |
| **ibeacon_hector_4** | 4 | | | |

You will need this information to configure your app. The parameters above tell you how to listen and distinguish between the iBeacons.

## RESTful service

The RESTful service provides data and functions used by your app. The service is available under the Base-URL ec2-52-212-125-177.us-east-2.compute.amazonaws.com, Port 8080. The URIs for requests are the following:

| ***HTTP Method*** | **URI** | **Parameter Type** | **Description** | **Example** | **Example Response** |
|---|---|---|---|---|---|
| *GET* | /userid/:name | name: any | Registers your user to the backend | /userid/foo | { "uid" : 9 } |
| *GET* | /question/:uid | uid: Integer | Getting a question for your registered user | /question/9 | { "uid" : 9, "qid" : 9, "question" : "Wann muss ich aufs Klo?" } |
| *GET* | /answer/:uid/:qid | uid: Integer, qid: Integer | Getting 1-3 answers for your question | /answer/9/9 | { "qid" : 9, "uid" : 9, "aid" : 0, "answer" : "JETZT!!" } |
| *GET* | /proof/:uid/:qid/:aid | uid: Integer, qid: Integer, aid: Integer | Checks if your suggested answer was correct | /proof/9/9/1 | { "proof" : "OK", "qid" : 9, "uid" : 9, "aid" : 1, "result" : "Answer was correct. Your score went +1" } |
| *GET* | /score | | The scores of all teams | /score | [ { "username" : "Team1", "score" : 1 }, { "username" : "Team2", "score" : 1 }, { "username" : "Team3", "score" : 0 } ] |