POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master degree course in Ingegneria Informatica (Computer Engineering)

Master Degree Thesis

Native mobile application development with web technologies

Politecnico di Torino

Relatore
Prof. Antonio Servetti

Candidate
Daniele Salaris
matricola: 265553

Academic year 2020 – 2021

Summary

Mobile applications are the best way to exploit mobile devices capabilities in terms of hardware and software environment. However, to create this kind of application, a developer should design a different native application for each operating system. The most used mobile operating systems are Android and iOS [1], so each native mobile application should be developed at least two times. On the contrary, web technologies such as HTML, CSS and Javascript are highly portable but have the limitation that a web application must run inside the browser and cannot fully exploit the advantages of the operating system and the hardware device. Hybrid applications come to solve this problem giving the possibility to develop a mobile application with web technologies, combining the portability with the possibility to exploit the device features. This project aims to enhance further the flexibility of the development of mobile and web applications by sharing the same business layer with a custom presentation layer. The project implements an application to provide surveys on environmental comfort. It has a client-server architecture. The server is written in Javascript and exposes a REST API through the express library. The client is written in Typescript with the angular framework, and it uses the Nativescript framework to build the native app.

Contents

List of Figures

Chapter 1

Introduction

1.1 Mobile applications

A *mobile app* is an application that runs on mobile devices. Every Operating system has its own environment and APIs to develop applications. Applications developed with such APIs are called native applications. The most popular operating systems for mobile devices are Android and iOS [1]. Android apps mainly use Java or Kotlin as programming language [3] while iOS apps mainly use Swift [4]. For this reason, to create a native mobile application, a developer should write at least two separate applications increasing development cost.

1.2 Web applications

A *web application* is an application used through a browser that delivers services over the network. Web apps are generally organized in a client-server architecture with the client making use of standard technologies such as HTML, CSS and Javascript. The use of these standards causes one of the greatest advantages of a web application: portability. In fact, it is possible to write a web app once to work in all modern browsers.

1.3 Cross-platform mobile applications with web technologies

A *cross-platform application* is an application developed with frameworks that allow building the code for different platforms [5]. *Cross-platform mobile applications* are often developed with web technologies [6] such as HTML-like tags,

stylesheets and javascript, creating mobile applications giving portability between different platforms reducing development cost.

There are different architectural approaches to implement a cross-platform framework. For example Cordova framework encapsulate the application inside a web view [7] while Nativescript framework translate components to native widget [8]. With frameworks of the first type it is possible to use proper HTML while Nativescript uses a set of custom Components to match native widgets.

The purpose of this project is to explore possibilities given by such frameworks to create a cross-platform mobile + web application that shares logical service layer between web and mobile but has two custom presentation layer.

1.4 Progressive Web App

There is another option which is *Progressive Web App*, or PWA. A PWA is an application that runs in the browser, but it acts similarly to a native application and exploit some hardware features when running on mobile.

1.4.1 Why Cross-platform?

The same approach used in this project to develop a cross-platform application could be used to develop a Progressive one. So, why using a cross-platform approach? [9] There are two main reasons to prefer this kind of approach:

- Native components: Cross-platform applications use native components on the lower UI level while PWA use browser ones. The use of native components can lead to a better user experience thanks to the user's familiarity with platform-specific components and a faster UI response.
- Hardware: Even if it is true that PWA can use some hardware feature, this use is limited if compared with a native (or cross-platform) application that gives full access to the device capabilities.

The use case of this project does not need much hardware usage, so the reason for using a cross-platform approach is the use of native components.

1.5 Case study

In previous years there has been a growing interest in considering the environmental comfort perceived by people to reduce energy consumption inside buildings. This case study is placed within a project that has, among other purposes, the intent to collect data on the perceived environmental comfort levels.

In order to collect this data, a previous study used google forms to carry out surveys on environmental quality perceived by some researchers at an ARPA building in Val d'Aosta. Those surveys asked questions about the perception of temperature (thermal comfort), natural and artificial light (visual comfort), sounds and noises (acoustic comfort), odours (air quality) and all those perceptions as a whole (global comfort). [10]

The case study of this project aims to create an application to provide those surveys without using external instruments such as google forms.

Chapter 2

Technologies

2.1 REST

REST or *REpresentational State Transfer* [11] is an architectural style for distributed systems. It can be used to build a server interface over HTTP. There are some principles to follow to define a REST interface:

- Client-Server: The interface implements a set of server endpoints that a client can use to read and write some data.
- Resources: The interface handles unique resources. A URI identifies each resource.
- Stateless: The interface does not have a session. This principle allows horizontal scalability.
- Layered architecture: The server has a layered architecture to promote maintainability and separation of concerns. For example, a 3-layer server could have a data access layer to communicate with the database, a service layer to implements business logic and a control layer to handles HTTP requests and responses and orchestrate service calls.

2.2 JWT

As previously said, a REST API should be stateless, but in some cases could be helpful to remember some information about the client. For example, if some protected endpoints need user authentication and authorization, it could be advantageous to remember the user identity to avoid asking username and password

for each request. A solution could be using HTTP sessions and cookies, but this solution would lose horizontal scalability.

A better solution is to use a *JSON Web Token* (JWT) [12]. JWT is an open standard that defines a way to exchange information as a JSON object securely. It is composed of three parts, header, payload and signature, and it is encoded in base 64 to have the format "xxx.yyy.zzz", where "xxx" is the header, "yyy" the payload and "zzz" the signature. Each part has a different meaning:

- header: is a JSON object with two attributes.
  - typ: is always the string "JWT".
  - alg: is a string the indicates the algorithm used for the signature, usually HMAC SHA256 or RSA.
- payload: is the proper JSON object carried by the JWT. It can have any attributes. For example, it could have the user id and the user role.
- signature: The server signs the JWT using the algorithm specified in the header plus a secret. This sign authenticates the JWT, ensuring that the server has created it.

The fact that information is self-contained in the JWT lets the server avoid storing session information explicitly. Consequently, the service can still be considered stateless, and it is still possible to scale it horizontally.

2.3 Node.js

Node.js [13] is a Javascript runtime based on the Google V8 engine and designed to run Javascript code directly over the operating system, instead of running inside the browser. Node is very useful to develop the server with javascript technologies. The execution model is the same as standard Javascript: a single thread and an event handler loop that handles asynchronous code. For this reason, it is not possible to implement multithreading patterns.

Node comes with a set of standard modules, and it is highly extensible thanks to the possibility to create custom ones. Examples of standard libraries are "fs", used to interact with the file system and "http", used to implement an HTTP server. External modules can be installed through a package manager such as NPM.

2.4 Express

Express [14] is a javascript library and framework for node js, used to develop web servers. With Express, it is possible to implement HTTP endpoint, creating dynamic web pages or REST APIs. One of the primary concepts of this framework is middleware. A middleware is a function that accepts three parameters: req, res and next.

- req: an object with some attributes and methods that gives information about the request. For example, the attribute "body" is an object that contains data of the request body.

- res: an object with some attributes and methods to handle the response: For example, the methods "status" set the response's status code while the method "send" send the actual response.

- next: a function that calls the next middleware.

An essential aspect of middlewares is the fact that they can be chained together. In fact, it is possible to associate an endpoint with more than one middleware. Each middleware calls the next one (parameter "next"), and they can share data attaching attributes to the "req" parameter. Using middlewares to perform simple actions can lead to high code reusability.

2.5 MongoDB

MongoDB [15] is a document-oriented database. A document-oriented database is a NoSQL DB that stores data in a semi-structured way; it can be seen as a hierarchical key-value store [16]. One of the main advantages of document-oriented databases is the flexibility of the documents structure [16], which is helpful during the development phase.

In mongo, there are two main elements: documents and collections. Documents are the database records stored in BSON format (a binary version of JSON). Collections are sets of documents; they should be used to group documents modelling the same entities and having the same structure, but technically it is not mandatory. An example of a collection could be the set of all users.

2.6 Document and relational databases comparison

However, why choosing a document-based database instead of a relational one? Both types of DB have advantages and disadvantages. The main advantages of a relational database are the ACID properties [17]:

- Atomicity: a relational DB can group operations into atomic transactions. If a transaction fails, the DBMS rolls back all previous operations.
- Consistency: Also referred to as correctness, is a property that guarantees the impossibility for transactions to violate data constraints and the inexistence of corrupted data.
- Isolation: Transactions cannot affect other transactions so, mid-air collisions are not possible.
- Durable: System failures do not affect the result of committed transactions.

Relational models are usually designed to avoid data repetitions and redundancy. This process, known as data normalization [18], guarantees efficient storage memory usage, avoiding waste of space. Thank to ACID properties relational databases are really good when there is the need of a strong realiable and consisten database but they tend to scale vertically [19] [20]. For this reason they can became slower as the data size grows.

Document databases advantages are mainly due to a completely different way of designing models. As previously said, relational databases privilege a strictly relational and normalized model to avoid data consistency and integrity problems. Otherwise, document databases use data redundancy (denormalization) to increase the speed of reading operations [21]; in fact, repeating data inside documents in a hierarchical way allows to avoid JOIN operations.

Data denormalization cause a loss of consistency. For this reason, document DBMS also provides ACID transactions that can be used to perform critical operations where ACID properties are needed [22], for example, payment transactions. It is essential to note that ACID transactions temporally lock resources, causing a reduction in the reading speed. However, if most DB operations are reading ones, and only a limited part of writing operations are critical, then document database reading operations are still faster than their relational counterparts.

Another advantage of document databases is that the semi-structured arrangement of entries is more flexible [16] than the relational model. For this reason, the model structure can change and evolve easier than the relational one.

2.7 Mongoose

Mongoose [23] is an ODM (Object Document Mapping) library for Node and Mongo. It models mongo entries using schemas; schemas are used to read and write data in the DB. Each schema saves data in a different mongo collection.

2.8 Angular

Angular [24] is a Typescript framework used to build single-page web applications. A single-page application (SPA) is a web application that can dynamically generate the UI on the client-side. With traditional web applications, the client make a request for each different page. With SPA, the client requests the entire application once and then requests raw data and autonomously takes care of the presentation.

The central aspect of Angular is the creation of components. Component allows organizing the UI in modular pieces. Each component is characterized by a template, one or more stylesheets and a Typescript class. The template represents the UI structure of the component. It is written in an extension of HTML that uses both HTML tags and other Angular components. The stylesheet defines the component style. They can be simple CSS or preprocessed style like SASS. The Typescript class is in charge of handling all the logic of the components. For example, it handles component lifecycle, elaborates component data and interacts with other Angular elements such as services or router. Angular components follow the MVC pattern with the View corresponding to the template, and both Model and Controller corresponding to the Typescript class. Another vital aspect of Angular is the set of infrastructural modules that can be used to implement and define several parts of the application. For example, with the *AppRoutingModule* it is possible to configure and handle the internal navigation, while with the *HttpClientModule* it is possible to communicate with a server using the HTTP protocol.

2.9 Nativescript

Nativescript [25] is an open-source framework to develop cross-platform Android and iOS applications using web technologies. It is not bound to a single framework. For example, it works with plain javascript, Angular, React and Vue. Being a cross-compiled framework, Nativescript does not use HTML to define the UI structure, but a tag set compiled to native UI components. For example, *Label* is a native text label, while *TextField* is native text input. An essential set of tags is the set used to arrange native components in the interface, the layouts. There are several layouts; for example, *StackLayout* arranges the components side by side vertically

or horizontally while *FlexboxLayout* arranges the components in a way analogue to *CSS Flexboxes*.

2.10 Nativesctipt + Angular

The purpose of this project is to create a web application and a mobile application in a single codebase, sharing business logic with custom presentation layer. Nativescript and angular support this development paradigm using *nativescript-schematics* [26]. These schematics give the possibility to share some parts of the code splitting only the platform-specific part.

The project should mostly share:

- Components logic: All the operations to manage the component lifecycle, Event handling and interaction with other Angular utilities or services.
- Services: As told in the previous sections, services are the core logic of an angular application. They usually perform operations that are not related to the specific platform so, they are highly shareable. Note that there are cases where it might be useful to use platform-specific services; for example, in this project, the service that stores and read the JWT token is split between web and mobile because they access the local memory in different ways.
- Routing: Routes declaration and programmatic navigation.

The project should mostly not share:

- UI: The web application uses HTML tags, while the mobile application uses native components tags. So, they cannot inherently be shared.
- Several modules implement platform-specific logic. For example, in order to communicate with the server, the web app uses *HttpClientModule* while the mobile app uses *NativeScriptHttpClientModule*. Both modules implement the *HttpClient* interface, so services that need to make HTTP calls can inject the *HttpClient* instance, and the framework will inject the correct platform-specific instance.

2.10.1 Example of code sharing

To specify which files are shared and which ones are platform-specific, it is necessary to follow these file naming rules [27]:

- By default, all files are shared.
- A mobile-only file must have ".tns" before the file extension.

- A file in the same directory of a mobile-only file with the same name, but without the ".tns", is web-only.

Let us take as an example a screen to login into an application. We will need an Angular component to implement the screen and an Angular service to communicate with the server.

The component directory will have the following files:

- login.component.html: web-only file that defines the screen structure with HTML tags.
- login.component.css: web-only file that defines the style for the web component.
- login.component.tns.html: mobile-only file that defines the screen structure with Nativescript tags.
- login.component.tns.css: mobile-only file that defines the style for the mobile component.
- logic.component.ts: shared typescript file that defines the Angular component class. It handles component lifecycle, implements form logic and communicates with the authentication service.

The authentication service will be a simple file *auth.service.ts* with the Angular service class that implements the logic to communicate with the server using the *HttpClient*.

Chapter 3

Architecture

The application has a client-server architecture with client and server communicating via a REST interface. The server is written in javascript using Node.js and Express framework with the document-oriented database MongoDB providing data persistence. The client is built using the Angular framework. It consists of a dual Web and Mobile application with the shared logical layer. Nativescript framework provides native compilation.

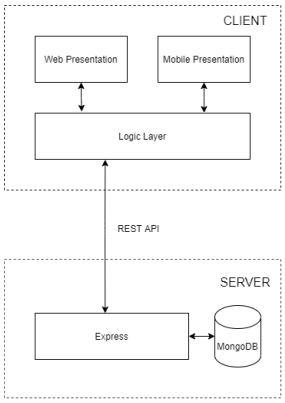

Figure 3.1: Project architecture.

The Server source code is stored in the repository "https://github.com/DanieleSalaris/thesis-server.git" [28] while the client source code is stored in the repository "https://github.com/DanieleSalaris/thesis-client.git" [29]

In order to create a TEST environment the application has been deployed using some cloud services. The database is accessible through Mongo atlas [30] the official mongo cloud DB as a Service (DBaaS). The server has been deployed using Heroku [31] a cloud Platform as a service (PaaS). The server contains both the REST apis and the web server distribution. Heroku server url: "https://s265553-thesis-server.herokuapp.com" [32]

3.1 Configuration

The application can start In order to start the dev environment it is possible to follow these instructions:

3.1.1 Server

Instruction to start the server:

- Install Node.js.
- Install Docker [33] and docker compose [34].
- Clone server project from github repository [28].
- Command: npm install (installing dependencies).
- Command: npm run start-db (starting a container with mongodb server).
- Command: npm run dev (starting server).

Note that command "npm start-db" start a docker container that exposes a MongoDB server. It is possible to avoid docker and docker compose installation starting an instance of MongoDB server that listens on port 27017 with root-username: "root" and password "password". Moreover it is possible to avoid having a local db-server connecting the server to Mongo atlas DBaaS.

3.1.2 Client

The client can connect to both local and remote server. The web server uses two different starting commands while the mobile needs some configuration.

Instruction to start web client:

- Clone client project from github repository [29].
- Command: npm install (installing dependencies).
- Command: npm run start-local (starting web server connected to local api server).
- Command: npm run start-remote (starting web server connected to remote api server)

Mobile application reads the server url from the file with the interceptor with path "/src/app/global/mobile.interceptor.tns.ts". Instruction to start mobile app:

- Connect to local-server:

  - Install ngrok [35] to create and https tunnel with the server.
  - Command: ngrok http 4242 (server listen on port 4242).
  - Insert tunnel-link in the interceptor file adding '/api' in the end. (e.g. https://805e2271cd53.ngrok.io/api/).
- Connect to remote-serve
  - Insert server url in the interceptor file: "https://s265553-thesis-server.herokuapp.com/api/"
- Configure nativescript environment following instruction on the official guide.
- Start an android emulator or connect an android mobile device with developer permissions enabled.
- Command: npm run android.

3.2 Models

Models are the structured data that the application uses to handle and store information. All models are designed in a document-db way so they are denormalized (Figure: 3.2). Some models are static and defined in configuration files while other are dynamically generated and stored on the database.

3.2.1 Survey

A *survey* is a model that wrap a set of questions. It is possible to define surveys using a configuration file in JSON format. A survey is an object with two attributes:

- _id: used to identify the survey.
- questions: an array of objects; each object represent a different question.

Question

The skeleton of a question. There are three types of questions: Choice, Array and Input. The inspiration for these types was taken from Lime Survey [36]. a question is an object with the following attributes:

- _id: used to identify the question.
- type: the type of the question. It Can be "choice", "array", or "input".
- data: Each type of question has a different set of specific data. These set are contained in this attribute.

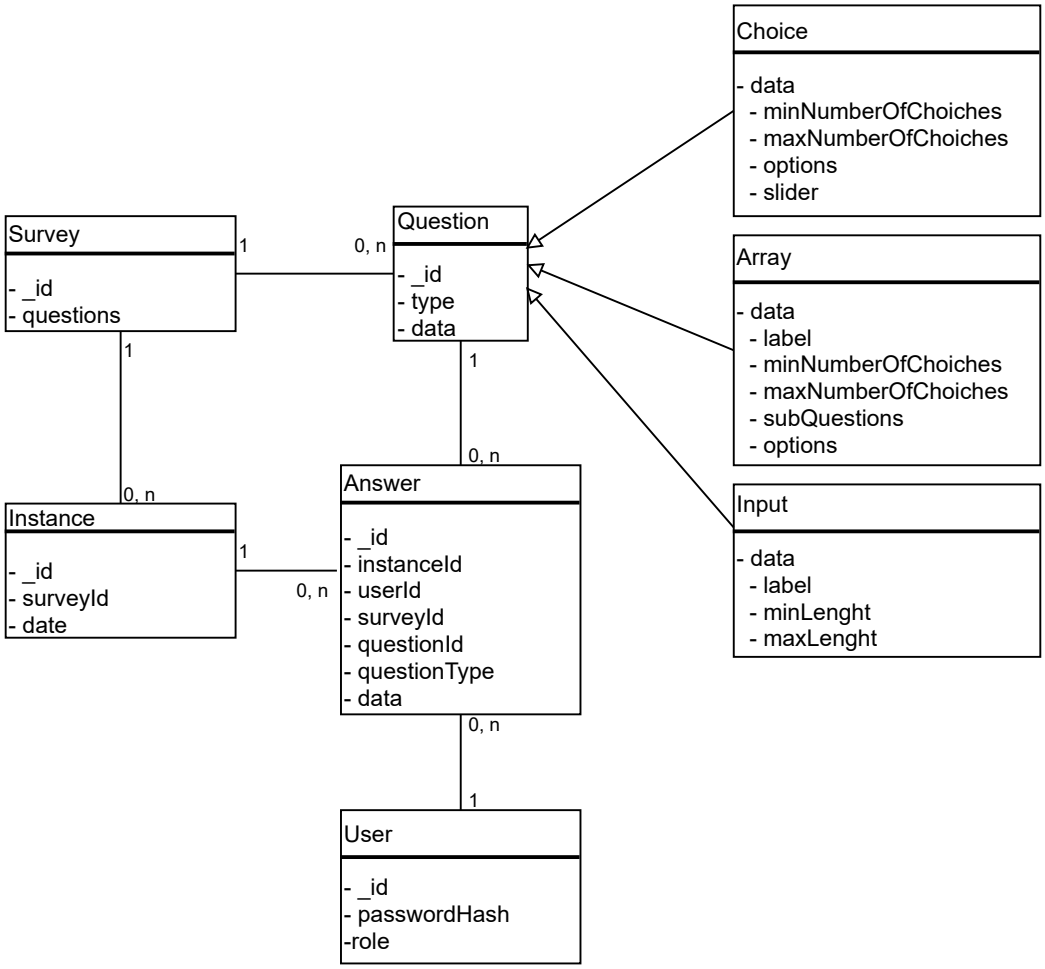

Figure 3.2: Model

Choice

Question of type *Choice* models a close-handed question. The inspiration for this type was taken from lime survey single choice [37] and multiple choice [38] questions. Data field contains:

- label: a string that represents the text of the question.
- minNumberOfChoices: a number that represents the minimum number of options to select.
- maxNumberOfChoices: a number that represents the maximum number of options to select.
- options: an array of objects that represents the list of possible answers. Each

object has an attribute "label", which is the text of the close-handed answer.

- slider: an optional boolean that tells if the options should be presented as a list or as a slider. It can be "true" only if maxNumberOfChoices is equal to "1"
- aspect: And ad hoc attribute added in a late phase of development used to know the aspect of comfort for this specific question.

Array

Array question represents a group of close-handed questions, all with the same answers. It can be seen as a two-dimensional table where the rows are a *choice* question while the columns are the options (answers). The inspiration for this type was taken from lime survey array questions [39]. Data fields contains:

- label: a string that represents the main answer to the question.
- minNumberOfChoices: a number that represents the minimum number of options to select.
- maxNumberOfChoices: a number that represents the maximum number of options to select.
- subQuestions: an array that represents the choice answers. Each object has an attribute "label", which is the text of the answer.
- options: an array of objects that represents the list of possible answers. Each object has an attribute "label", which is the text of the answer.

Input

It represents an open-ended question. The inspiration for this type was taken from lime survey text question [40]. Data field contains:

- label: a string that represents the main answer to the question.
- minLenght: a number that represents the minimum length of the answer.
- maxLenght: a number that represents the maximum length of the answers.

Example of survey

Figures 3.6, 3.9 and 3.10 are questions of type choice, array and input respectively. The following snippet configures these questions.

```
{
  _id: "1",
  questions: [
    {
      "_id": "1",
      "label": "Indichi qual'e il suo ambiente di lavoro",
      "minNumberOfChoices": 1,
      "maxNumberOfChoices": 3,
      "options": [
        {"label": "UFFICIO RUMORE AMBIENTALE - LATO
NORD PRIMO PIANO"},
        {"label": "UFFICIO NIR - LATO SUD PRIMO PIANO"},
        {"label": "LAB AS (ACQUE E SPETTROMETRIA) - LATO
SUD PRIMO PIANO"},
      ]
    },
    {
      "_id": "2"
      "label": "Quanto e soddisfatto delle condizioni
termiche (temperatura ed umidita) nel suo ambiente di lavoro?",
      "minNumberOfChoices": 1,
      "maxNumberOfChoices": 1,
      "subQuestions": [
        "Inverno",
        "Estate",
        "Stagioni intermedie (autunno e primavera)"
      ],
      "options": [
        {"label": "Molto insoddisfatto"},
        {"label": "Insoddisfatto"},
        {"label": "Leggermente insoddisfatto"},
        {"Label": "Neutro"},
        {"label": "Leggermente soddisfatto"},
        {"label": "Molto soddisfatto"}
      ]
    },
    {
      "_id": "3",
      "label": "What is your name?"
      "minLenght": 0,
      "maxLenght": 32
```

```
      }
    ]
}
```

3.2.2 Instance model

The same survey can be associated with different days. This association can be modeled using the Instance model. It is possible to declare instances in a configuration file, or it is possible to dynamically create and store in the DB the instance of the current day for a given survey. Instance model is an object with the following attributes:

- _id: it is used to identify the instance.
- surveyId: id of the survey associated with the instance.
- date: a string that represents the date of the instance. It is in the format "DD-MM-YYYY" where "DD" is the day of the month, "MM" is the month number, and "YYYY" is the year.

Example of instances

```
[
  {
    "_id": "1",
    "surveyId": "1",
    "date": "10-06-2021"
  },
  {
    "_id": "2",
    "surveyId": "1",
    "date": "15-06-2021"
  },
  {
    "_id": "3",
    "surveyId": "2",
    "date": "18-06-2021"
  },
]
```

3.2.3 Answer model

Each user can answer to a given instance (associated to a survey). All answers are stored in the DB. An answer is an object with the following attributes:

- _id: used to identify the answer, it is in the format instanceId + questionId + userId (where the sum is a string concatenation).
- instanceId: id of the instance.
- userId: id of the user who sent the answer.
- surveyId: id of the survey associated with the instance.
- questionId: id of the question, inside the survey.
- questionType: type of the question (choice, array, input).
- data: the actual answer. The format depends on the question type.
  - choice: an array of numbers, where the number are the indexes of the options.
  - array: an array of arrays of number where the arrays are associated to the sub questions while the numbers are the indexes of the options.
  - input: a string that represents the open-ended answer.

Example of answers

```
[
  {
    "_id": "1142",
    "userId": "42"
    "instanceId": "1",
    "surveyId": "1",
    "questionId": "1",
    "questionType": "choice",
    "data": [3]
  },
  {
    "_id": "1242",
    "userId": "42"
    "instanceId": "1",
    "surveyId": "1",
    "questionId": "2",
    "questionType": "array",
```

```
    "data": [
      [2],
      [3],
      [0]
    ]
  },
  {
    "_id": "1342",
    "userId": "42"
    "instanceId": "1",
    "surveyId": "1",
    "questionId": "3",
    "questionType": "input",
    "data": "Daniele"
  },
]
```

3.2.4 User model

User data are stored in the database as an object with the format:

- _id used to identify the user
- passwordHash: hashed and salted password.
- role: role of the user. It can be "user" or "admin"
  - admin: Can create other users and surveys.
  - user: Can answer surveys questions.

3.3 Server

3.3.1 Architecture

The server is written in Javascript with Node.js as runtime and implements a REST web service using the framework Express. The internal architecture is composed of four layers: data access, service, middlewares and routing.

Data access

The *data access* layer interacts with the database providing persistence of data. It uses the node library mongoose to interact with MongoDB. There are four schemas

used to model users, survey instances and answers. The meaning of the fields is explained in the models' section.

Schemas' snippet:

```
const userSchema = new mongoose.Schema({
  _id: {
    type: mongoose.SchemaTypes.String,
    required: true
  },

  passwordHash: {
    type: 'string',
    required: true
  },

  role: {
    type: 'string',
    required: true,
  }
})

const AnswerSchema = new Schema({
  _id: {
    type: SchemaTypes.String,
    required: true,
  },

  surveyId: {
    type: SchemaTypes.String,
    required: true,
  },

  instanceId: {
    type: SchemaTypes.String,
    required: true,
  },

  questionId: {
    type: SchemaTypes.String,
    required: true,
```

```
  },

  questionType: {
    type: SchemaTypes.String,
    required: true,
  },

  userId: {
    type: SchemaTypes.String,
    required: true,
  },

  data: {
    type:  Object,
    required: true,
  }
})

const InstanceSchema = new Schema({
  date: {
    type: SchemaTypes.String,
    required: true
  },

  surveyId: {
    type: SchemaTypes.String,
    required: true
  }
})
```

Service

The *service* layer is the core of server architecture. It implements business logic performing operation such as interacting with mongoose library, validating data, and throwing errors. There are five services: *jwtService*, *userService*, *surveyService*, *instanceService* and *answerService*.

jwtService *jwtService* handles the creation and validation of the JWT. It has the following methods:

- getJwtAlgorithm: Extract the algorithm to sign the JWT from the environment variables. The default value is "HS256".
- getJwtSecret: Extract the secret used to sign the JWT from the environment variables.
- createJwt: Accept a payload as input and create and returns the JWT with the payload.
- extractToken: accept an authorization header as input and extract and validate the JWT. If it fails, it throws a *JwtNotFound"* or a *JwtNotValid* error.

userService *useService* handles user creation and login. It has the following methods:

- generatePasswordHash: it accepts a plain password as input, and returns the hashed and salted value.
- validateUserFormat: it accepts a *userId*, a *password* and the boolean value *isAdmin* as input and validates those values.
- createUser: it accepts the same values of the previous methods as inputs and creates the user.
- upsertFirstAdmin: it accepts a *userId* and a *password*. The server calls this method at boot time to create the first admin defined in the environment variables.
- updatePassword: it accepts a *userId* and a *password*. If the user exists, it updates the hashed password associated with the user.
- login: it accepts a *userId* and a *password*, check that the user exists and that the password is correct and in case of success, it returns an object with the *userId* and the user's role.

answerService *answerService* handles operations to read and write answers. It has the following methods:

- validateAnswerFormat: It accepts an answer value and a question and checks that the value is valid for that question. Answer format is explained in the Model section.
- answer: It accepts an *instanceId*, a *surveyId*, a *questionId*, a *userId* and the answer value, checks that the value is correct (calling *validateAnswerFormat*) and it upsert the answer entry in the DB. If the value is not correct, it throws a *WrongAnswerError*

- formatAnswerId: it accepts an *instanceId* a *questionId* and a *surveyId*, and returns the answer id concatenating the three input values.
- getAnswer: it accepts an *instanceId* a *questionId* and *surveyId* and returns the associated answer if it exists.
- getAnswers: it accepts a mongo filter and returns the list of answers that match the filter.
- getAnswersCsv: it accepts an *instanceId* and finds all the answers associated with that instance, writing them on a CSV file. When this operation is done, the method returns a promise that returns the file path.

instanceService *InstanceService* is in charge of the logic to handle the surveys, for example, retrieving surveys questions. It has the following methods:

- loadIntances: it loads instances from the configuration file.
- getInstance: it accepts an *instanceId* and returns the corresponding instance. It can search the instance firstly on the DB and then on the configuration file.
- getSurvey: it accepts an *instanceId* and returns the survey related to the instance.
- getQuestion: it accepts an *instanceId* and returns the question list of the survey related to the instance.
- getAnswers: it accepts an *instanceId* and a *userId* and returns the response of the equivalent method of the *answerService*.
- getAnswers: it accepts an *instanceId* a *questionId* and a *userId* and returns the response of the equivalent method of the *answerService*.
- setAnswer: it accept an instance a *questionId* a *userId* and the value of the answer, and it set the answer calling the method answer of the *answerServic*.
- singletonGetTodayInstance: it implements the logic of retrieving the instance of the day.

surveyService *surveyService* handles logic to retrieve surveys and related questions. It has the following methods:

- loadSurveys: it loads surveys from configuration file.
- getSurvey: it accepts a *surveyId* and returns the corresponding survey if it exists.

- getQuestions: it accepts a *surveyId* and returns the questions related to the corresponding survey if it exists.
- getQuestion: it accepts a *surveyId* and a *questionId* and returns the corresponding question related to the corresponding survey if they exist.
- getAverage: it compute answers average for surveys with questions of type choice. As the database grows an operation of this kind became to heavy to be handled directly by the server. This kind of operation must be demanded to the DBMS. With document databases queries are not as much powerful as their relational counterparts but mongodb gives some instruments to perform complex operation on data. In particular this service make use of aggregation pipeline, a mongodb feature that give the possibility to perform complex operation one after the other. It accept as input a surveyId and returs an array of objects. Each objects has the following attributes:
  - questionId: The id of the question
  - rate: The average of answers. Each answer has a value between zero and the number of possibile close-handed answers minus one. The avarage is the sum of all answers over the number of close-handed answers minus one.
  - aspect: aspect of comfort of the question (thermal, acoustic, visual, indoor air quality). This value is used for visualization purposes.

  Operation of the pipeline:
  - filters answers with the input surveyId and the type choice.
  - takes as attribute the questionId and the value of answer.
  - group items with questionId computing the average of values.
  - sort results by questionIds.

Middlewares

As explained in the Technologies section, *middlewares* are a crucial aspect of an Express server. In this project there is a middleware to handle each REST API plus some middlewares for repeated actions in this project. REST API's middlewares always have the same structure. The initial part extract data from params, body, or JWT. The middle part calls services' methods to perform business logic. The final part sends a successful response or an HTTP error response in case of failure. Let us take as an example the middleware for the API "GET /api/instance/id". This middleware extracts the *instanceId* from the request params. It calls the

service to retrieve the instance passing the instance-id and. In case of success, it responds with the instance as a JSON object; in case of failure, it responds with an *HTTP 404 NOT FOUND* error.

```
async (req, res, next) => {
  const {instanceId} = req.params
  try {
    const instance = await instanceService.getInstance(instanceId)
    return res.json(instance)
  }
  catch (e) {
    if (e instanceof InstanceNotFound) {
      return res
        .status(httpStatusCode.clientError.NOT_FOUND)
        .json(formatError(e))
    }
    return next(e)
  }
})
```

The middleware that validates JWT extracts it from the authorization header and attach the extracted value to req values. The middlewares to check the role read the extracted token from the req values, and check that the role is correct. All three middlewares call the next middleware in case of success or respond with an HTTP 401 UNAUTHORIZED error in case of failure. It follows a snippet of the three middleware plus an example of use in a route.

```
validateJwt: (req, res, next) => {
  try {
    req.token = tokenService.extractToken(req.headers.authorization)
  } catch (e) {
    if (e instanceof JwtNotValid || e instanceof JwtNotFound) {
      res
        .status(httpStatusCode.clientError.UNAUTHORIZED)
        .send()
    }
    else {
     throw e
    }
    return
  }
```

```
  next()
},

roleAdmin: (req, res, next) => {
  const {token} = req
  if (token.role !== 'admin') {
    return res
      .status(httpStatusCode.clientError.UNAUTHORIZED)
      .send()
  }
  next()
},

roleUser: (req, res, next) => {
  const {token} = req
  if (token.role !== 'user') {
    return res
      .status(httpStatusCode.clientError.UNAUTHORIZED)
      .send()
  }
  next()
},

app.get(
    '/path',
    validateJwt,
    roleUser,
    (res, res, next) => {/* ...  */}
)
```

Routing

Routing in Express is the process of defining application endpoints. An endpoint is defined by an HTTP verb, a path and one or more middlewares. Express gives the possibility to defines routers to group routes that perform similar actions defining a prefix for all those routes. In this project, there are two routers:

- authRouter: defines all endpoints for user creation and authentication. The prefix for this router is "api/auth".
- instanceRouter: defines all endpoints to handle questionnaires. The prefix for this router is "api/instance".

- surveyRouter: defines enpoints to handle survey operations.

All the endpoints are described in the REST section. It follows a snippet with the routers' declaration plus an example of a route

```
app.use('/api/auth', authRouter,)
app.use('/api/instance', instanceRouter)

instanceRouter.get(
   '/:instanceId/question/:questionId/answer/',
   authMiddleware.validateJwt,
   authMiddleware.roleUser,
   async (req, res, next) => {/* ... */}
)
```

3.3.2 REST API

In order to deliver services to the client, the server exposes a REST API. It is possible to divide such api in three groups: *Auth* and *Instance* and *Survey*.

Auth

Auth exposes services to handle user creation and login and password modification.

```
POST api/auth/user
POST api/auth/login
POST api/auth/user/update-password
```

In order to create a user, it is possible to send a "POST" request to the endpoint "api/auth/user". If the user exists already the request fail. Request body accepts the following attributes:

- userId: id of the user, it is a string with minumum 3 characters and maximum 64.
- password: plain password of the user, it is a string with minumum 8 characters and maximum 128.
- isAdmin: a boolean that tells if the new user is an admin or a simple user, the default value is false.

Only admins can use this endpoint. In case of success, the request returns a "201 Created" response.

To login, it is possible to send a "POST" request to the endpoint "api/auth/login" passing as a body an object with the attributes:

- userId: id of the user.
- password: plain password of the user.

In case of success, the request returns an object with a single attribute "token" containing a JSON web token used to authorize other requests.

To change password it is possible to send a POST request to the endpoint "api/auth/user/update-password". If the user does not exists the request fail. The body of the request accept the following attributes:

- userId: id of the user.
- password: plain password of the user.

Data validation constraint is the same as user creation endpoint. This is not a proper REST endpoint because it has a verb in his path.

Instance

The instance group is composed of two more groups of endpoints. The first of endpoints work as a proper REST API with level 2 of maturity [41]. The second group is not a proper REST API but exposes some utilities needed to perform specific actions.

```
GET api/instance
GET api/instance/:id
GET api/instance/:id/question/
GET api/instance/:id/question/:id
GET api/instance/:id/question/:id/answer
POST api/instance/:id/question/:id/answer

GET api/instance/:id/answer-csv
POST api/instance/today-instance
```

The endpoint of the first group exposes the "instance-question-answer" relation.

- "GET api/instance[/:id]": returns all instances or an instance with a specific id with the Instance model format. Both users and admins can use this endpoint.

- "GET api/instance/:id/question[/:id]" given a specific instance, it returns all questions or a question with a specific id with the Question model format. Both users and admins can use this endpoint.
- "GET api/instance/:id/question/:id/answer." given a specific question, it returns the answer of the question (if it exists) with the Answer model format. Only Users can use this endpoint.
- "POST api/instance/:id/question/:id/answer." given a specific question, it let the user writing the answer. Only users can use this endpoint.

As previously said the second group it is not a proper REST API but it exposes some utilities endpoints:

- "GET api/instance/:id/answer-csv": given a specific instance, it returns a csv file containing all the answers of all the users who answered. Only admins can use this endpoint.
- "POST api/instance/today-instance": it return the dynamically created instance of today following the singleton pattern, with the Instance model format. Only users can use this endpoint.

Survey

Survey exposes a single endpoint used to receive some statistic on surveys answers.

```
GET api/survey/:id/average
```

This endpoint requirement appeared in a late phase of development as an ad hoc endpoint to receive the average of answers for survey's questions. It works only for question of type choice returning an object with a single attribute:

- average: it is the array returned from the method *surveyService.getAverage*

3.3.3 Environment variables

Some environment variables can be used to configure some server behaviours. In the development environment, it is possible to configure those variables in the file ".env".

Environment variables:

- TCP port to reach the server.
- JWT_SECRET: secret used to sign the JWT.
- JWT_SIGN_ALGORITHM: hash algorithm to sign the JWT

- DB_URL: URL used to reach the database.
- DB_NAME: name of the database.
- DB_USER: username used to login into the database.
- DB_PASSWORD: password used to login into the database.
- SURVEYS_PATH: the path of the file with the surveys declaration.
- INSTANCES_PATH: the path of the file with the instances declaration.
- TMP_FILES: the path of the directory used to write temporary files.
- ADMIN_NAME: When the server starts, it creates an admin with this name if it does not exists yet.
- ADMIN_PASSWORD: admin password: the password of the admin described above.

Example

Source code of the ".env" file:

```
PORT=4242
JWT_SECRET=SECRET
JWT_SIGN_ALGORITHM=HS256
DB_URL='mongodb://localhost'
DB_NAME=tesi_db
DB_USER=root
DB_PASSWORD=password
SURVEYS_PATH="/Users/danielesalaris/Development/web/thesis/
    thesis-server/resources/survey.json"
INSTANCES_PATH="/Users/danielesalaris/Development/web/
    thesis/thesis-server/resources/instances.json"
TMP_FILES="/Users/danielesalaris/Development/web/thesis/
    thesis-server/tmp_files"
ADMIN_NAME="root"
ADMIN_PASSWORD="12345678"
```

3.4 Client

The client is realized using the angular framework and its development patterns. It is composed of three main layers: Service layer, component logic, and component

presentation. This project aims to share the much logic as possible between mobile and web applications; for this reason, most of the services and component logic is in common, while the presentation part is different for mobile and web.

3.4.1 Services

"Service is a broad category encompassing any value, function, or feature that an application needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well." [42].

In the project, there are several services, each one handling a different part of the base logic.

AuthService

AuthService handles authentication and authorization logic using a sub-service called "TokenStorageService" to handle JWT storage. It has three methods:

- login: it accepts a username and a password, send login requests to the server and returns an observable with the server response saving the JWT in the storage as a side effect.
- readToken: it returns the token saved in the storage.
- validToken: it returns a boolean value that tells if the token exists and is not expired.

TokenStorageService

TokeneStorageService handles JWT storage. It has a different implementation for web and mobile apps because they have different APIs to store data, "localStorage" for the web and "ApplicationSettings" for mobile. The service has the following methods:

- saveToken: it writes the JWT on the storage.
- readToken: it returns the JWT written in the storage.
- decodeToken: it decodes the token and returns its body.

SurveyService

SurveyService is a service that handles surveys logic and interaction with the server for survey APIs. It has the following methods:

- getSurvey: it accepts an instance id and returns an Observable of the survey associated with the instance. It also stores the survey and returns the stored survey if instanceId doesn't change.
- getQuestion: it accepts an instance id and a question id, and it returns an Observable with the associated question.
- getNextQuestionId: it accepts an instance id and the id of the current question; it returns the id of the next question.
- getPrevQuestionId: it works in the same way as the last method, but it returns an Observable with the id of the previous question. These two methods combined allows the implementation of scrolling between questions.
- getInstances: it returns an Observable with the list of instances.
- getTodayInstance: it returns an Observable with the instance of the day.
- answerQuestion: it accepts an instance id, a question id and the answer value and sends a post request to the server to the API to the endpoint to write the answer. It returns an Observable with the server response.
- getAnswer: it accepts an instance and a question id and returns the answer value if it exists.
- getAverage: it communicate with the server endpoint to read answers statistics. It returns an Observable with the server response adding the *percentage* attribute on the aggregated answer object. The percentage is computed multiplying the *rate* attribute for one hundred.

3.4.2 Components

As previously said, the other two main layers of client architecture are components logic and components presentation. These two layers are implemented using Angular components. "Components are the most basic UI building block of an Angular app. An Angular app contains a tree of Angular components". [43]. It follows a list of the main components of the application.

LoginComponent

LoginComponent handles the logic and the interaction for authenticating the users. File *login.component.ts* is shared between web and mobile and it implements the component logic. It defines an angular form with two required fields: username and password. It handles form logic implementing a method to clear the form and a method that interacts with *AuthService* to send a login request. If the request

fails it enables an error flag. At UI level it appears as a card with a form to insert a username and password with a icon button to toggle password visibility and a pair of buttons to confirm and reset the fields. If the user does not insert one of the fields it appears an error message right below the field. In case of login request failure it also appears an error message below the form. Web (Figure: 3.3) structure and style are defined in files *login.component.html* and *login.component.css* while the mobile counterparts (Figure 3.13) are defined in files *login.component.tns.html* and *login.component.tns.css*. The structure is similar but is implemented with different tags. Web uses *HTML divs* and *CSS flexboxes* to organize the layout while mobile uses nativescripts layout components such as *FlexboxLayout* and *StackLayout*. For the form field and buttons the Web UI uses angular material components, while the mobile one uses nativescript *TextField* and *Button*. In both cases the form fields are connected to the angular form using angular reactive forms directives. Angular materials fields automatically handles the appearance of the field error message but in the mobile layout they are implemented using a nativescript *Label* component. Finally the request failure error message is implemented with a *span* tag in the web and with a *Label* in mobile.

InstancesComponent

InstancesComponent is a list the contains all the available instances (Figures 3.4, 3.14). The architecture is organized following the smart-dumb paradigm. *InstancesComponent* is fully shared between web and mobile and is in charge of handling the logic. The logic layer (*instances.component.ts*) interact with *SurveyService* to retrieve instances from the server. The UI layer (*instances.component.html*) calls the presentational component passing the list of instances retrieved from the server.

InstancesDummyComponent is the presentational component and it is charge of implementing web and mobile layouts. The file *instances-dummy.component.ts* is shared and it simply declare the input variable to read the list of instances. At UI level in both web and mobile it renders a list of card. Each card represent an instance and it is possible to click the card to navigate to the associated survey. On the web (*instances-dummy.component.html*, *instances-dummy.component.css*) the list of instances is represented as rows of cards. On the mobile (files *instances-dummy.component.tns.html*, *instances-dummy.component.tns.css*) the list of instances is structured in a vertical scroller. The nativescript component that handles the scrolling is *ScrollView*. Both web and mobile calls *InstanceComponent* which is charge of rendering the instance card.

InstanceComponent

InstanceComponent is the component in charge of rendering the instance card on *InstancesComponent*. Its logic layer (*instance.component.ts*) is shared. It declares the input variable *instance* which is an instance type (section: 3.2.2, parse the instance date and generate the link used to navigate to the survey. At UI level it appears in both mobile and web app as a clickable card showing the instance date (Figures: 3.4, 3.14). The web UI (*instance.component.html*, *instance.component.css* uses an *<a>* tag with an angular *routerLink* directive to wrap the date and implement the navigation. The mobile app (*instance.component.tns.html*, *instance.component.tns.css*) uses a *FlexboxLayout* with the angular-nativescript *nsRouterLink* directive for the same purpose.

TodayInstance

TodayInstance is a component that replaces InstancesComponent when working on single instance mode. At logic level (*today-instance.component.ts*) it handles interaction with *SurveyService* to retrieve today instance id and redirection to the survey. At UI level (Figures 3.5, 3.21) appears both in web and mobile as a card with a label and a button to start answering the surveys. On the web app (*today-instance.component.html*, *today-instance.component.css*) the component is implemented with HTML tags (* *, *<button>*...). On the mobile app (*today-instance.component.tns.html*, *today-instance.component.tns.css*) the component is implemented with nativescript component (*FlexBoxLayout Button*....).

QuestionRedirectionComponent

QuestionRedirectionComponent is a component that handles interaction with *SurveyService* to retrieve the id of the first question and redirects to it. Logic of the component is shared between web and mobile and it is implemented in file *question-redirection.component.ts*. This component does not have a presentation layer.

QuestionComponent

QuestionComponent is a wrapper for surveys questions (Figures: 3.6 3.16). It handles questions common logic such as retrieving questions from the service, navigating between questions and retrieving answers if they exist yet. At the presentation level, it appears as a card that contains a question, an input to answer the questions and a pair of navigation buttons to scroll to the next and previous question. Rendering of question, input and buttons is handled by a specific component that depends on the type of question. This component is

divided in three levels following a pattern similar to smart-dumb one.

In this case the first level is implemented by *QuestionComponent*. This component is not a fully smart component but have a presentational part. The logic layer of this component (*question.component.ts*) is shared between web and mobile and it is in charge of extracting the data to recognize the current question from the url path (*instanceId* and *questionId*) and to interact with *SurveyService* to retrieve the question and the ids of the previous and next question if they exists which are used for navigating to those questions. The UI is different for web (*question.component.html*, *question.component.css*) and mobile (*question.component.tns.html*, *question.component.tns.css*) but in both cases it implements the layout for the wrapping card and calls *QuestionComponentDummy*.

The second level is *QuestionDummyComponent*. This component is fully shared between web and mobile. Its logic layer (*question-dummy.component.ts*) receive some input data from *QuestionComponent* and understands the question type in order to call the right component. It also emits when the user click on question navigation buttons. The UI layer (*question-dummy.component.html*) is also shared and does not add any platform-specific content. It consist on a list of angular **ngIf* directives that renders the correct type-specific question component.

The third layer is implemented by the type-specific question components. Each component handles the form logic and renders the form with platform-specific tags. For this reason each one has a shared logic layer and a platform-specific presentation layer. Navigation buttons are rendered by these components but to avoid too much code repetitions the rendering is delegated to the presentational component *ConfirmButtonsComponent*.

QuestionChoiceComponent

QuestionChoiceComponent implements an input to answer a question of type choice (close-handed question). At UI level (Figures: 3.6, 3.16) , it appears as a list of checkboxes. Each checkbox has a label that represents the answer.

QuestionChoiceSliderComponent

QuestionChoiceSliderComponent is a variant of *QuestionChoiceComponent*, but instead of a list of checkboxes, it appears as a slider (Figures: 3.7, 3.8, 3.17, 3.18). The current selected answer is represented as a label beneath the slider and it changes in real time accordingly to slider value.

QuestionArrayComponent

QuestionArrayComponent implements an input to answer a question of type array. On the web app (Figure 3.9, it appears as a table where the rows represent the subquestions while the columns represent the answers. On the mobile app, it appears as a group of QuestionChoiceComponent where each element let answer to a sub-question (Figures 3.19, 3.20).

QuestionInputComponent

QuestionInputComponent implements an input to answer the question of type input. It appears as a simple text field to enter the open-ended answer (Figures 3.10, 3.21).

DataVisualizationComponent

DataVisualizationComponent let the admins read statistics on the answers for a specific survey. The UI design for this component was taken from Fasano and Fissore master degree thesis [2] (Figure 3.12. The component's logic layer communicates with SurveyService to retrieve answers statistics. The presentation layer (Figure 3.11 shows the aspect of comfort (Thermal, Acoustic, Visual, Indoor Air quality) for each aggregated answer and a horizontal bar that shows the percentage value. The horizontal bar is implemented by a sub-component called *DataVisualizationRow*. The main difference with the proposed design is that the expected percentage values from the project and monitored values are missing. Future development of the application will integrate those values.

3.4.3 Routing

Routing is about organizing navigation between components. Routing structure is divided in three main areas. *Login*, *Admin* and *User*.

- Login: It contains only the *loginComponent* used to insert users and admin credential. Only not logged user can use this component. After a corret login there is a redirection to other areas.
- Admin: Only admins can use this area. It has "admin" prefix. For now there is only one sub-route that shows *DataVisualizationComponent*
- User: Only users can use this area. It groups all the other components used for answering the surveys and described previously.

[

```
{
  path: 'login',
  component: LoginComponent,
  canActivate: [LoginGuard]
},
{
  path: 'role-redirect',
  component: RoleRedirect,
},
{
  path: 'admin',
  canActivate: [AuthGuard, AdminGuard],
  children: [
    {
      path: 'data-visualization',
      component: DataVisualizationComponent
    },
    {
      path: '',
      redirectTo: '/admin/data-visualization',
      pathMatch: 'full'
    },
  ]
},
{
  path: '',
  canActivate: [AuthGuard, UserGuard],
  children: [
    {
        path: '',
        redirectTo: '/instance',
        pathMatch: 'full',
    },
    {
      path: 'home',
      component: HomeComponent,
    },
    {
      path: 'instance',
      component: TodayInstanceComponent,
      // component: InstancesComponent,
```

```
        canActivate: [AuthGuard]
      },
      {
        path: 'instance/:instanceId',
        component: QuestionContainerComponent,
        canActivate: [AuthGuard]
      },
      {
        path: 'instance/:instanceId/question/:questionId',
        component: QuestionComponent,
        canActivate: [AuthGuard]
      }
    ]
  },
];
```

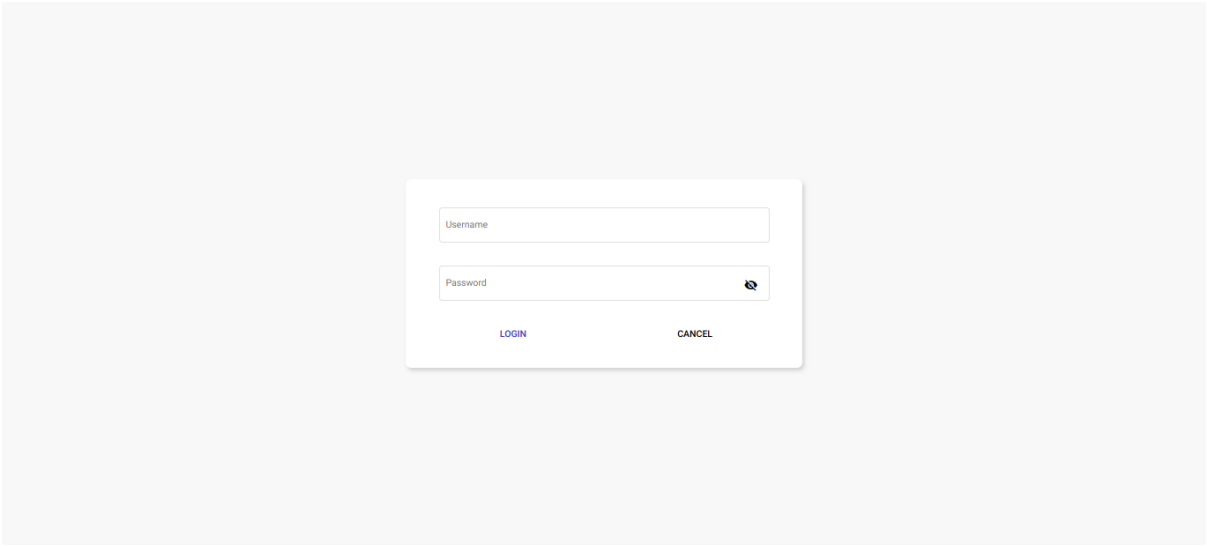

Figure 3.3: Login web

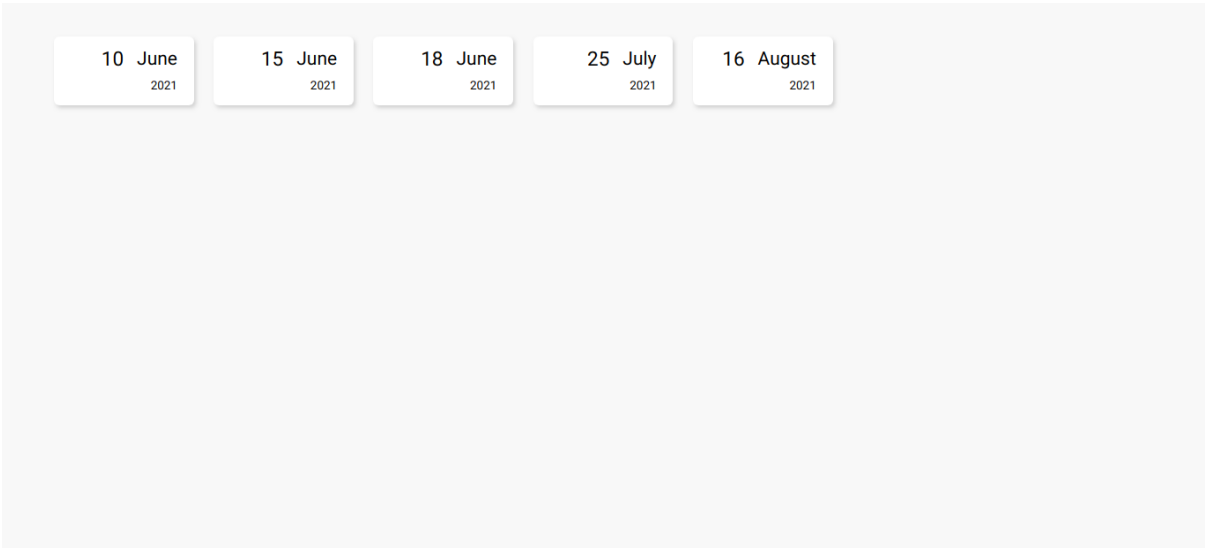

Figure 3.4: Instances web

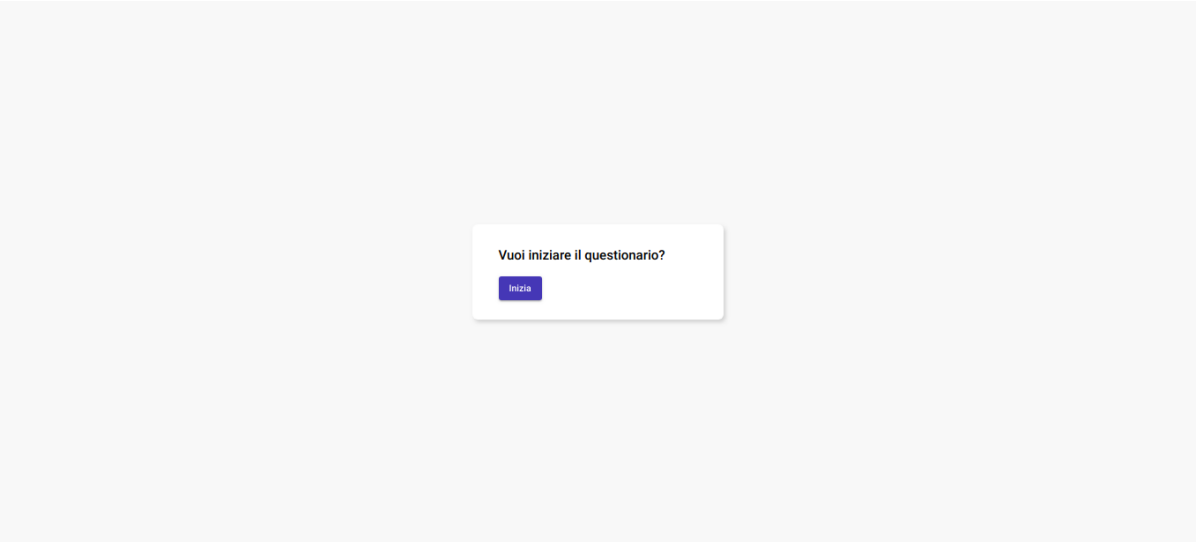

Figure 3.5: Today instance web

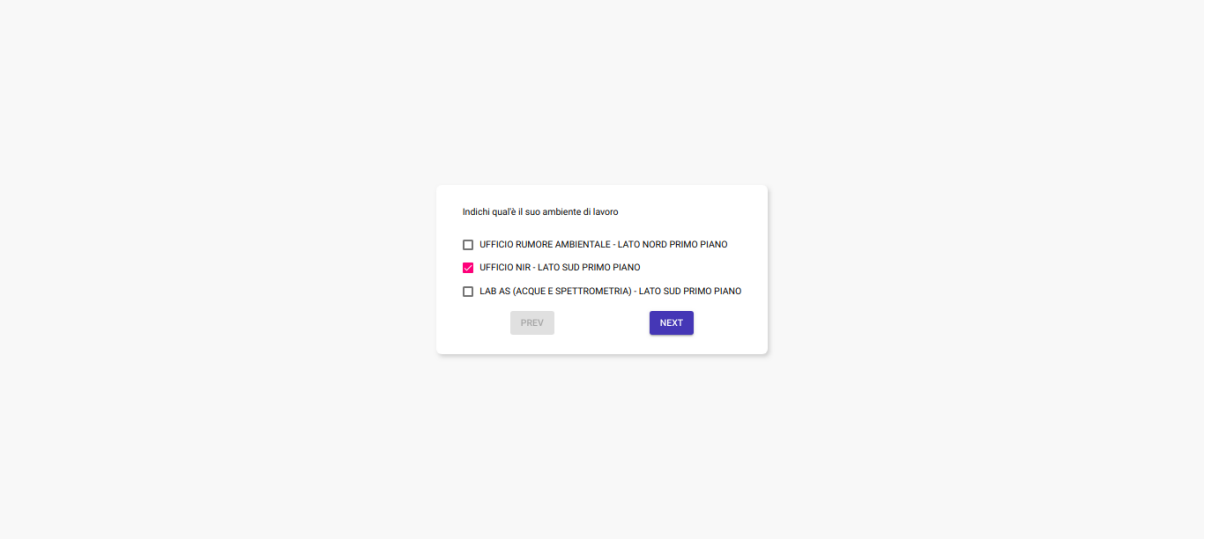

Figure 3.6: Question choice web

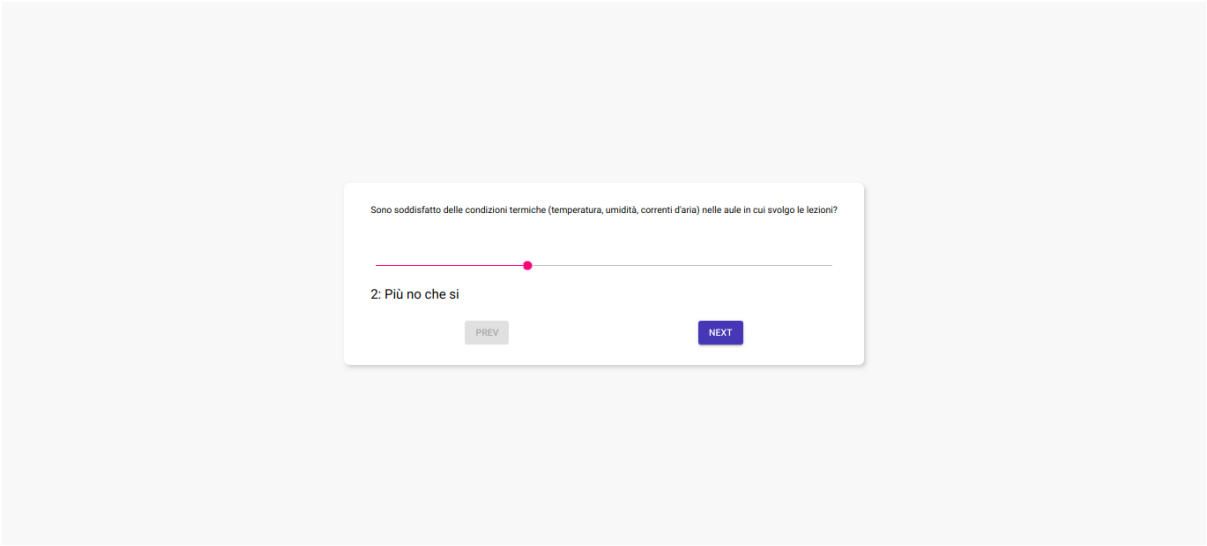

Figure 3.7: Question choice slider 1 web

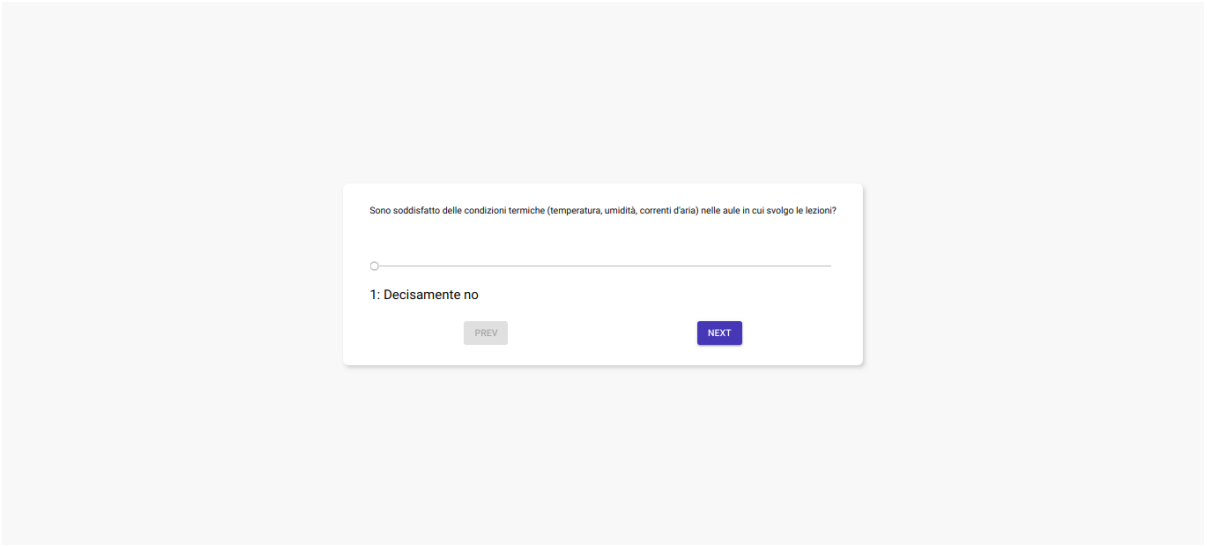

Figure 3.8: Question choice slider 2 web

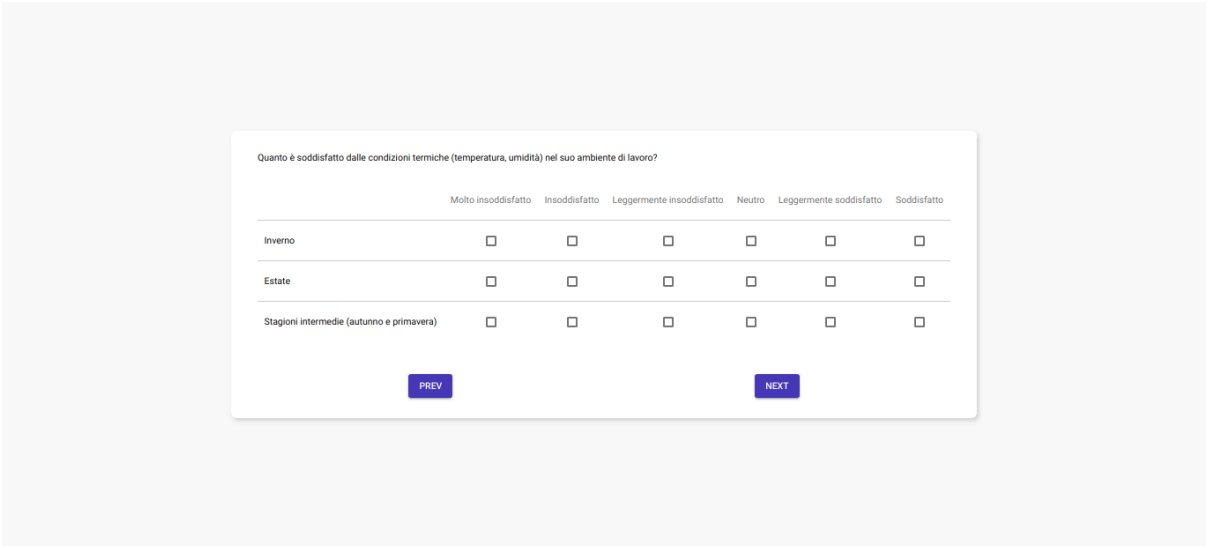

Figure 3.9: Question array web

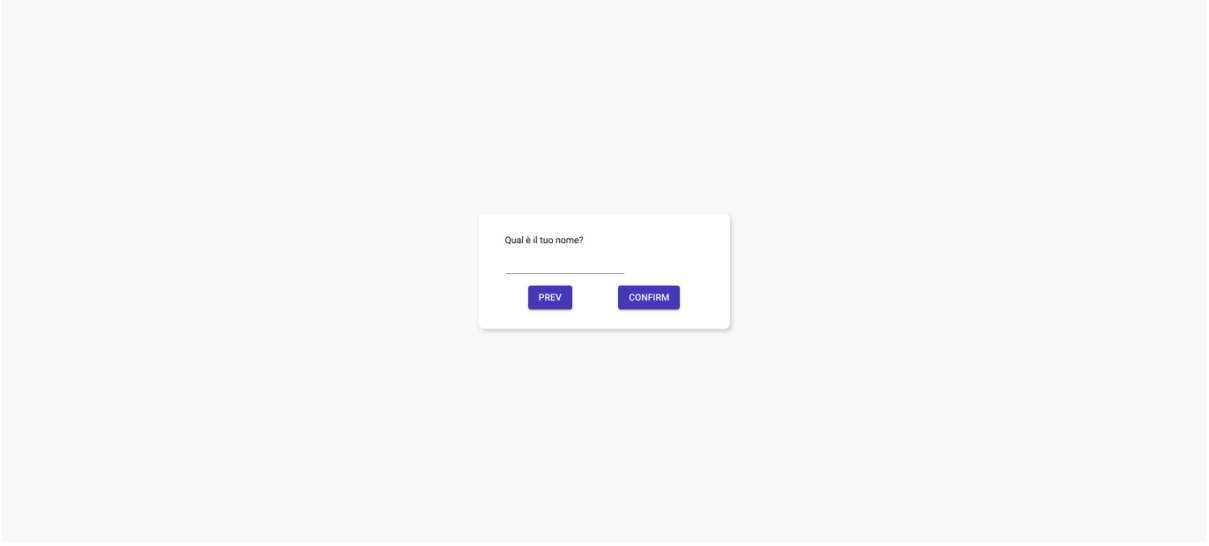

Figure 3.10: Question input web

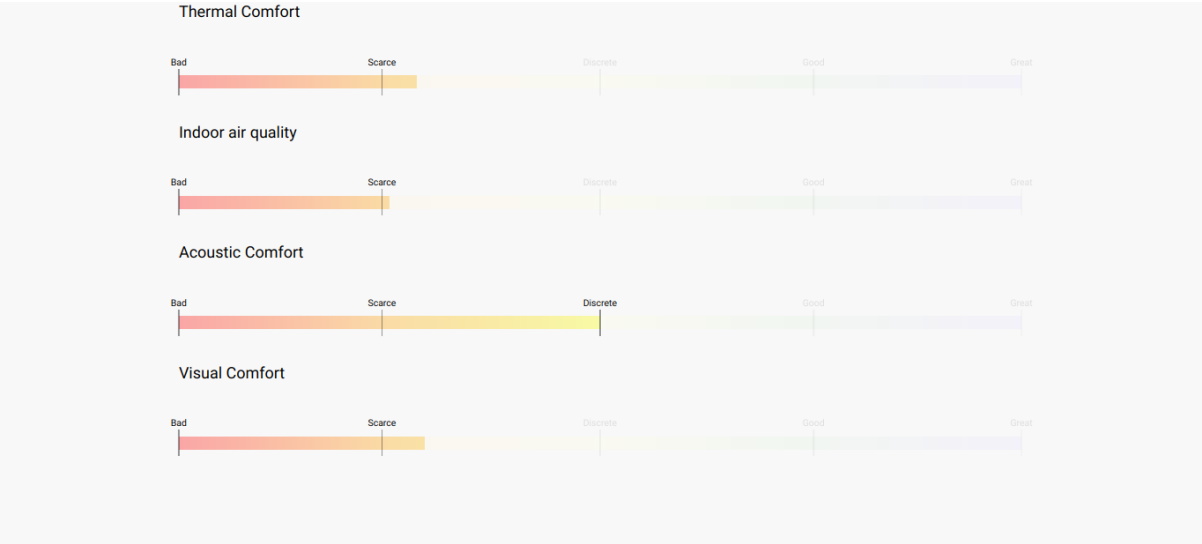

Figure 3.11: Data visualization

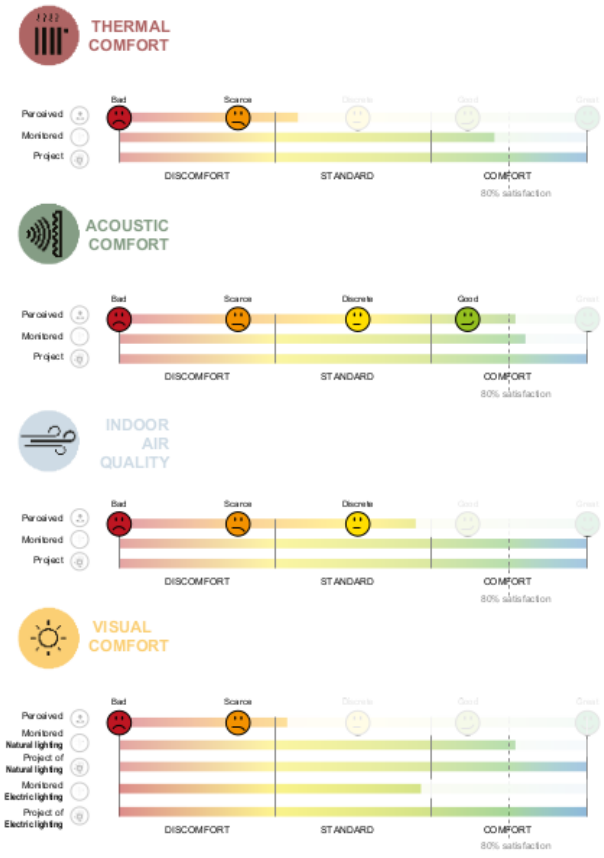

Figure 3.12: Data visualization original design. Adapted from [2]

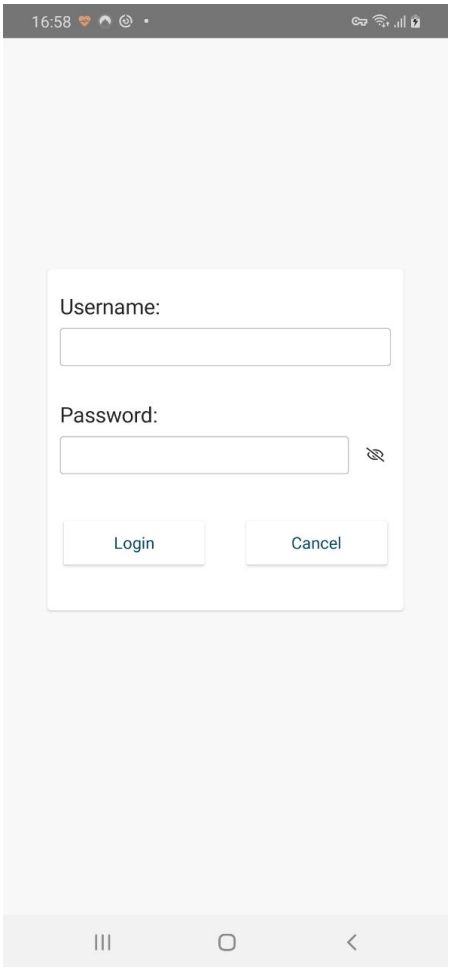

Figure 3.13: Login mobile

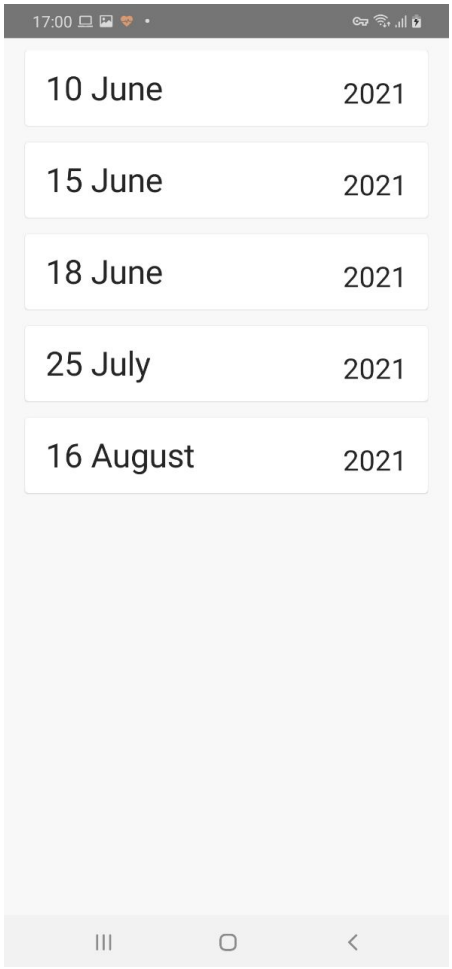

Figure 3.14: Instances mobile

Figure 3.15: Today instances mobile

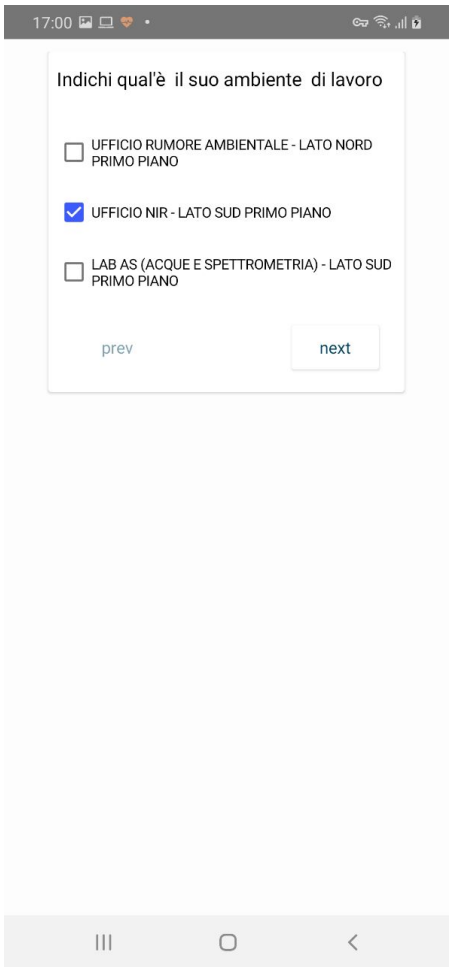

Figure 3.16: Question choice mobile

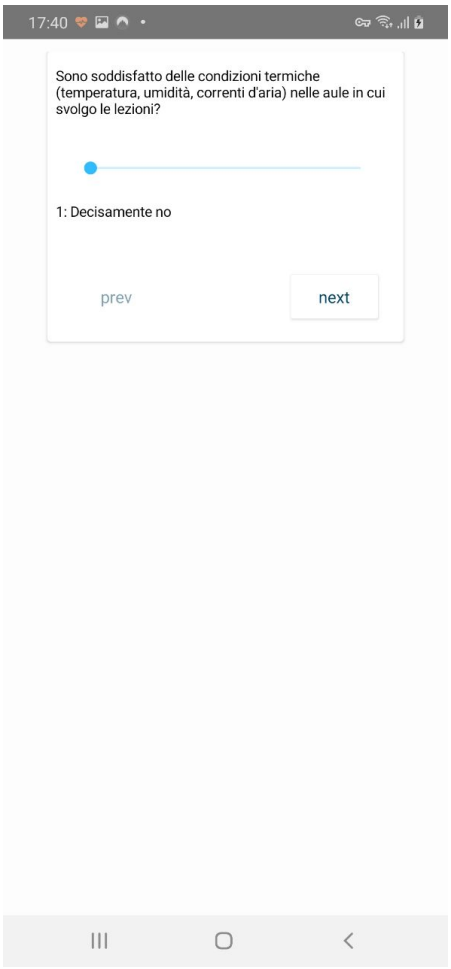

Figure 3.17: Question choice slider 1 mobile

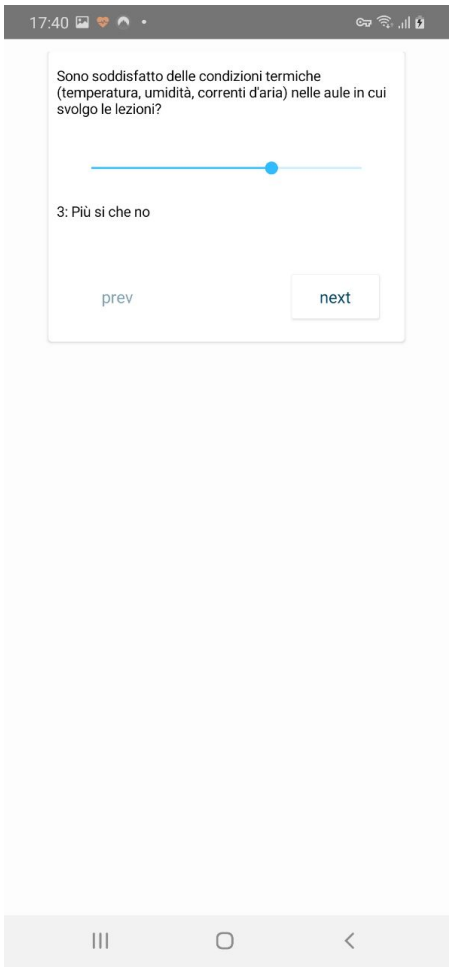

Figure 3.18: Question choice slider 2 mobile

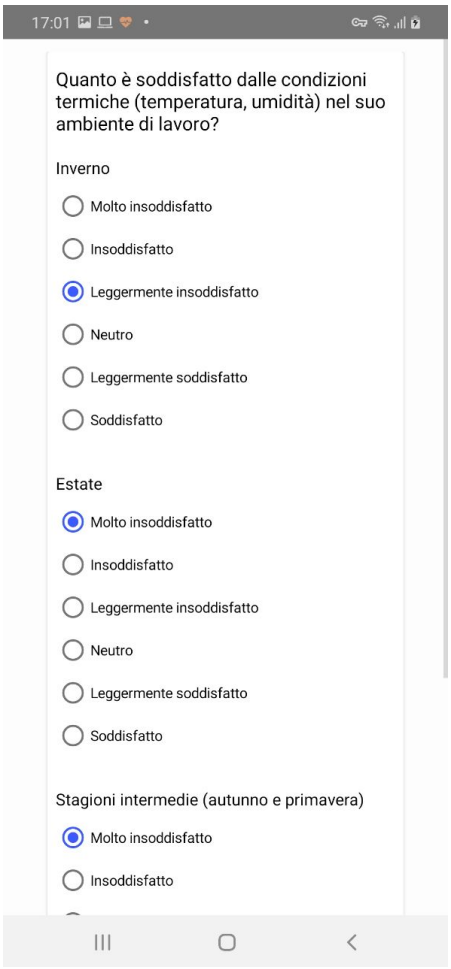

Figure 3.19: Question array 1 mobile

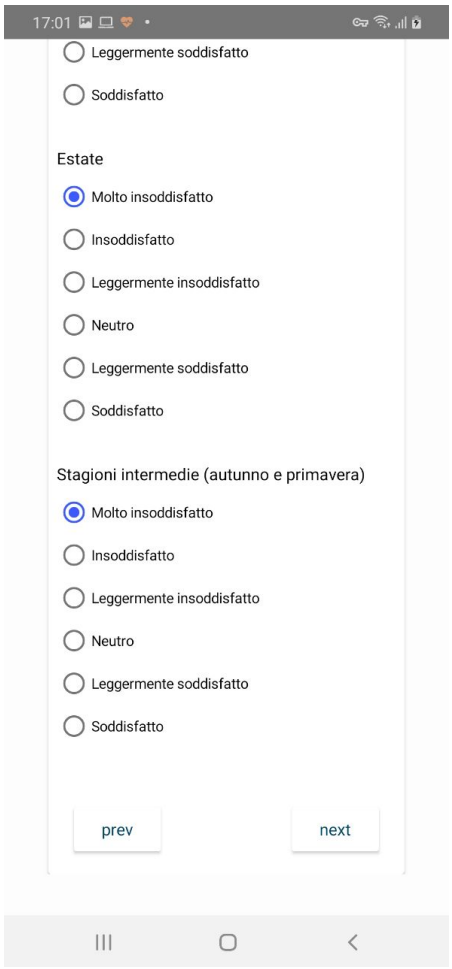

Figure 3.20: Question array 2 mobile

Figure 3.21: Question input mobile

Chapter 4

Conclusions

This thesis aimed to explore the possibilities offered by combining web development and mobile native cross-platform development with web technologies to create an application that works both as a web app and a mobile application, sharing the business logic with a custom presentation layer. I think that this goal has been met. The project shows how it is possible to use this kind of approach to develop an application using frameworks that support this paradigm, angular and nativescript in this case. It shows that the application thus created follows angular development standards such as componentization, decoupling between business logic and components using services and routing organization. This development approach is compatible with componentization paradigms such as smart-dumb components. Moreover, the project not only shows that it is possible to have a common business logic with a platform-specific presentation layer, but it also shows that this approach is more flexible than that. In fact, the framework gives the possibility to decide which files are platform-specific and which ones are shared. For example, it is possible to have a fully shared component (e.g. a smart component) or a platform-specific service (e.g. service to handle local storage).

Bibliography

[1] Statista. Mobile operating systems' market share worldwide from January 2012 to June 2021, 2021. URL https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[2] Silvia Fasano and Virginia Fissore. Indoor environmental quality and comfort perception in offices, 2021. URL https://webthesis.biblio.polito.it/19774/.

[3] GeeksforGeeks. Top Programming Languages for Android App Development. URL https://www.geeksforgeeks.org/top-programming-languages-for-android-app-development/.

[4] Apple. Swift. URL https://developer.apple.com/swift/.

[5] Asper Brothers. Cross-Platform App Development – Explore Frameworks, Technology and Business Benefits. URL https://asperbrothers.com/blog/cross-platform-app-development/.

[6] Anirudh Nagesh and Carlos Caicedo. Cross-Platform Mobile Application Development, 2012. URL https://www.researchgate.net/publication/263416908_Cross-Platform_Mobile_Application_Development.

[7] Apache Cordova. Apache Cordova Overview. URL https://cordova.apache.org/docs/en/10.x/guide/overview/.

[8] Nativescript. User Interface Widget Components, . URL https://v7.docs.nativescript.org/ui/overview.

[9] Mariusz Marcak. Cross-Platform Apps vs Native Apps vs Progressive Web Apps, 2021. URL https://www.businessofapps.com/insights/cross-platform-apps-vs-native-apps-vs-progressive-web-apps/.

[10] Niccolò Oggiani. Indoor environment quality and global comfort: field measurements and analysis at the ARPA headquarters in the Aosta Valley., 2020. URL https://webthesis.biblio.polito.it/16368/.

[11] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Fielding PHD dissertation.

[12] jwt.io. Jwt. URL https://jwt.io/introduction.

[13] Node js. Node js. URL https://nodejs.org/en/.

[14] Express. Express. URL https://expressjs.com/it/.

[15] MongoDB. Mongodb, . URL https://www.mongodb.com/.

[16] MongoDB. Document database, . URL https://www.mongodb.com/document-databases.

[17] Essential SQL. Sql acid database properties explained. URL https://www.essentialsql.com/sql-acid-database-properties-explained/.

[18] Wikipedia. Database normalization. URL https://en.wikipedia.org/wiki/Database_normalization.

[19] Sneha Binani, Ajinkya Gutti, and Shivam Upadhyay. Sql vs. nosql vs. newsql-a comparative study. URL https://caeaccess.com/archives/volume6/number1/binani-2016-cae-652418.pdf.

[20] Douglas Kunda and Hazael Phiri. A comparative study of nosql and relational database. URL https://ictjournal.icict.org.zm/index.php/zictjournal/article/view/8.

[21] GeeksForGeeks. Denormalization in databases. URL https://www.geeksforgeeks.org/denormalization-in-databases/.

[22] MongoDB. What are acid transactions?, . URL https://www.mongodb.com/basics/acid-transactions.

[23] Mongoose. Mongoose. URL https://mongoosejs.com/.

[24] Angular. Angular, . URL https://angular.io/.

[25] Nativescript. Nativescript, . URL https://nativescript.org/.

[26] Nativescript. Nativescript schematics, . URL https://github.com/NativeScript/nativescript-schematics.

[27] Nativescript. Code sharing introduction, . URL https://v6.docs.nativescript.org/angular/code-sharing/intro.

[28] Daniele Salaris. Thesis server, 2021. URL https://github.com/DanieleSalaris/thesis-server.

[29] Daniele Salaris. Thesis client, 2021. URL https://github.com/DanieleSalaris/thesis-client.

[30] MongoDB. Mongo atlas, . URL https://www.mongodb.com/atlas/database.

[31] Heroku. Heroku. URL https://www.heroku.com/.

[32] Daniele Salaris. s265553 thesis server, 2021. URL https://s265553-thesis-server.herokuapp.com.

[33] Docker. Docker, . URL https://www.docker.com/.

[34] Docker. Overview of docker compose, . URL https://docs.docker.com/compose/.

[35] ngrok. ngrok. URL https://ngrok.com/.

[36] Lime Survey. Lime survey, . URL https://www.limesurvey.org/.

[37] Lime Survey. Single choiche question, . URL https://manual.limesurvey.org/Question_types#Single_choice_questions.

[38] Lime Survey. Multiple choiche question, . URL https://manual.limesurvey.org/Question_types#Multiple_choice_questions.

[39] Lime Survey. Array, . URL https://manual.limesurvey.org/Question_types#Array.

[40] Lime Survey. Text questions, . URL https://manual.limesurvey.org/Question_types#Text_questions.

[41] Leonard Richardson. Justice will take us millions of intricate move, 2008. URL https://www.crummy.com/writing/speaking/2008-QCon/.

[42] Angular. Introduction to services and dependency injection, . URL https://angular.io/guide/architecture-services.

[43] Angular. Component, . URL https://angular.io/api/core/Component.