Software documentation and UML Model for Metrica Application

Authors

Corresponding author: Anna Przewięźlikowska [1] https://orcid.org/0000-0002-5894-0201,
przewie@agh.edu.pl
Wioletta Ślusarczyk [2], Klaudia Wójcik [3], Marek Ślusarski[4],

[1] AGH University of Science and Technology, Faculty of Mining Surveying and Environmental Engineering,
Department of Integrated Geodesy and Cartography, Al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Centre of Security Technologies at AGH University of Science and Technology
[3] Administracja Nieruchomości MEEST Sp. z o.o. Kraków, Poland
[4] University of Agriculture in Krakow, Faculty of Environmental Engineering and Land Surveying,
https://orcid.org/0000-0002-8573-936X

1. Requirements engineering for the mobile application

Metrica mobile application development has followed the standard requirements engineering processes - use cases, user groups, and functional and non-functional requirements have been identified (Nuseibeh and Easterbrook 2000; Hull et al. 2011).

There are three user groups defined in the platform: Guest, Client, and Administrator. The three user groups have been allocated the following access rights:

Table 1. Availability of functions for individual users of the Metrica application

		USER		
		Guest	*Client*	*Administrator*
Functions	Viewing the list of points	+	+	+
	Displaying of information about points	+	+	+
	Image display	+	+	+
	Displaying of topographical descriptions	+	+	+
	Displaying points on the map	+	+	+
	Navigation to the point	+	+	+
	Editing point information	-	+	+
	Adding a new point	-	+	+
	Adding photos	-	+	+
	Adding topographical descriptions	-	+	+
	Deletion of photos	-	+	+
	Deletion of topographical descriptions	-	-	+

The non-functional requirements include:

- simple and intuitive user interface,
- running on Android (version 9.0 and higher),
- security guaranteed by limiting access to functions at a given level of use (*Guest / Client Administrator*),

- using the REST API communication protocol, the server is open to for communication with other types of client applications in the future (e.g. web browser, iPhone app).

The use cases of the mobile application have been designed for all user groups. As an example, the use cases for the Administrator have been presented below:

1.1. Functional and non-functional requirements

In order for the server and the application to function correctly, several requirements were imposed on them. These are divided into functional and non-functional requirements. The former include the functions to be performed by the application. Non-functional requirements, on the other hand, relate to technical requirements that affect the correct operation of the application.

REQUIREMENTS:

- user registration,
- user login,
- view a list of all points in the database,
- searching for a point from the list,
- display of point information,
- photo display,
- display of topographical descriptions,
- navigation to a point,
- displaying points on the map,
- the possibility of adding a point,
- possibility to edit point information,
- possibility to add a photo of a point,
- the possibility of adding a topographical description,
- deletion of photographs of a point.

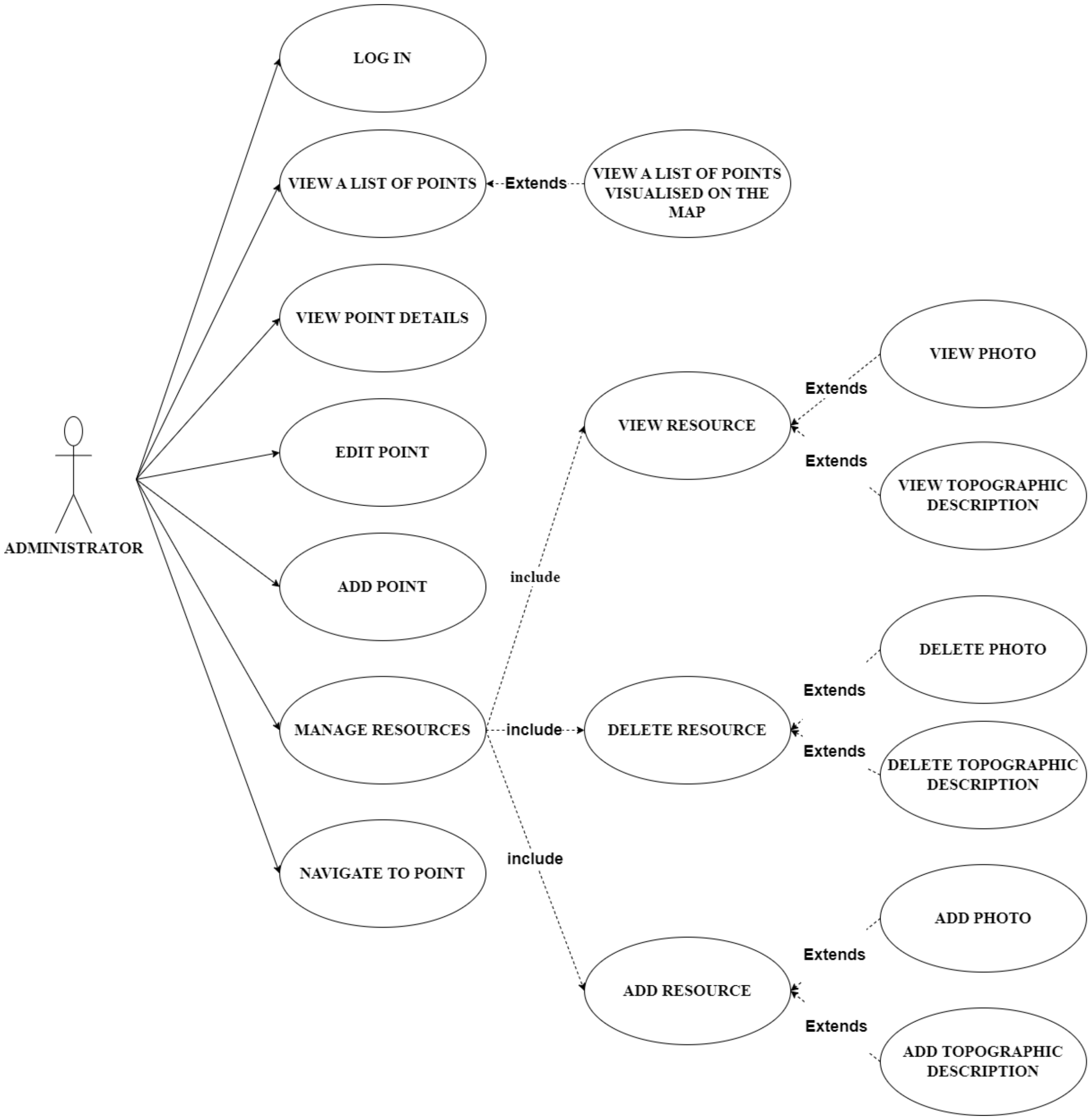
NON-FUNCTIONAL REQUIREMENTS:

- running on Android (version 9.0 and higher),
- security guaranteed by limiting access to functions at a given level of use (*Administrator / Customer / Visitor*),
- using the REST API, the server will be able to interact with other applications in the future,
- simple and intuitive user interface,

1.2. Use cases for the mobile application

The following use case diagrams for different types of users provide a graphical representation of how the user sees the properties of the system (**Błąd! Nie można odnaleźć źródła odwołania.**, **Błąd! Nie można odnaleźć źródła odwołania.**, **Błąd! Nie można odnaleźć źródła odwołania.**).

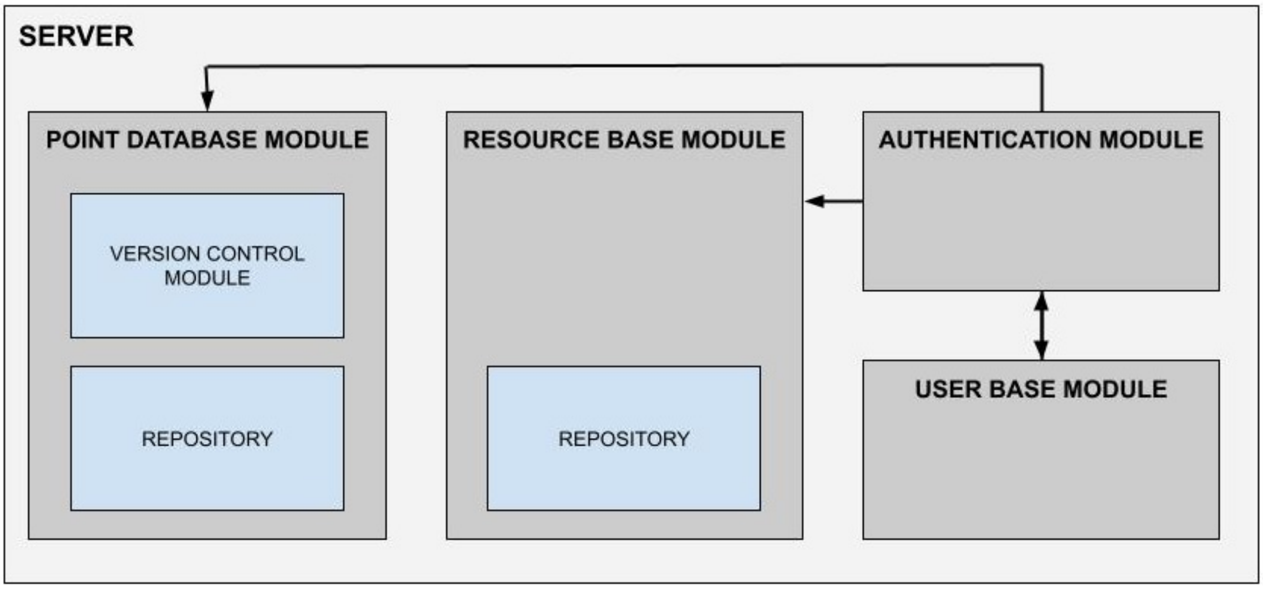
Figure 1 Use case diagram - Administrator



1.1. Server

- **Points database module** - responsible for adding, storing, processing and validating point information. It collects them in a standardized format. It works with the authentication module to control access to the data. It consists of two components:
  - version control module - allows changes made to be checked,
  - repository - allows objects to be written straight into the database regardless of which database system is used and where it is located,
- **Asset base module** - allows the addition and deletion of images as well as topographic descriptions. It has a repository that operates in the same way as the repository of the point base module.
- **Authentication module** - the authentication module also controls access and is responsible for logging in and registering users. It communicates with the user base module in order to retrieve user information. It also communicates with the points database module to restrict access to selected data,
- **User database module** - stores user information about first name, surname and e-mail address.

Figure 2 Metrica server modules



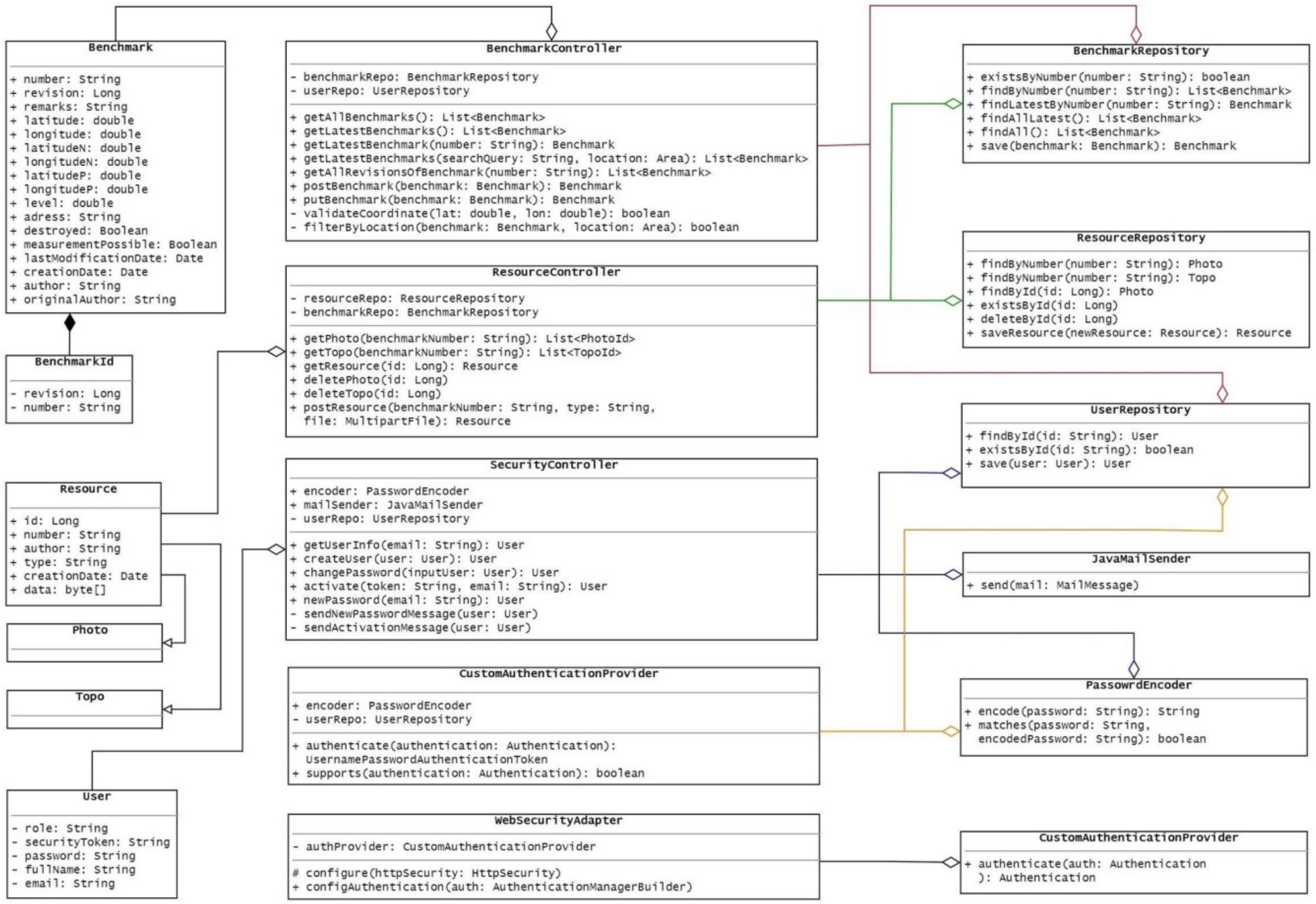
The server is divided into packages, each containing classes for different tasks:

- data Model, Controller with business logic
- Repository responsible for the database operations
- Security managing data access and encryption with SSL.

Spring MVC (Model-View-Controller) (Spring by VMware Tanzu) has been used to create classes and methods that control the handling of HTTP requests. One of the capabilities of the Spring Framework is Request Mapping, which is done using annotations(Spring by VMware Tanzu).

- The main data models are : *Benchmark, Resource* and *User*. They describe the points, the images / topographic descriptions, and the users, respectively.
- The *BenchmarkController* class performs all actions on the points. This class contains several methods to display all points, display a point with a given number, and edit or add a new point. It is based on the *Benchmark* model.
- Data is downloaded and sent via the *BenchmarkRepository* and the *UserRepository.*
- With *ResourcesController*, one can manage the resources for photos and topographic descriptions. It contains methods for adding or deleting a photo/topographic description.
- The *SecurityController* class is responsible for user-related activities. Among other things, it includes methods that allow to register the user, change the password and activate the account.

Figure 3 Class diagram - server


Benchmark
+ number: String
+ revision: Long
+ remarks: String
+ latitude: double
+ longitude: double
+ latitudeN: double
+ longitudeN: double
+ latitudeP: double
+ longitudeP: double
+ level: double
+ adress: String
+ destroyed: Boolean
+ measurementPossible: Boolean
+ lastModificationDate: Date
+ creationDate: Date
+ author: String
+ originalAuthor: String
BenchmarkId
- revision: Long
- number: String
Resource
+ id: Long
+ number: String
+ author: String
+ type: String
+ creationDate: Date
+ data: byte[]
Photo
Topo
User
- role: String
- securityToken: String
- password: String
- fullName: String
- email: String
BenchmarkController
- benchmarkRepo: BenchmarkRepository
- userRepo: UserRepository
+ getAllBenchmarks(): List<Benchmark>
+ getLatestBenchmarks(): List<Benchmark>
+ getLatestBenchmark(number: String): Benchmark
+ getLatestBenchmarks(searchQuery: String, location: Area): List<Benchmark>
+ getAllRevisionsOfBenchmark(number: String): List<Benchmark>
+ postBenchmark(benchmark: Benchmark): Benchmark
+ putBenchmark(benchmark: Benchmark): Benchmark
- validateCoordinate(lat: double, lon: double): boolean
- filterByLocation(benchmark: Benchmark, location: Area): boolean
ResourceController
- resourceRepo: ResourceRepository
- benchmarkRepo: BenchmarkRepository
+ getPhoto(benchmarkNumber: String): List<PhotoId>
+ getTopo(benchmarkNumber: String): List<TopoId>
+ getResource(id: Long): Resource
+ deletePhoto(id: Long)
+ deleteTopo(id: Long)
+ postResource(benchmarkNumber: String, type: String,
file: MultipartFile): Resource
SecurityController
+ encoder: PasswordEncoder
+ mailSender: JavaMailSender
- userRepo: UserRepository
+ getUserInfo(email: String): User
+ createUser(user: User): User
+ changePassword(inputUser: User): User
+ activate(token: String, email: String): User
+ newPassword(email: String): User
- sendNewPasswordMessage(user: User)
- sendActivationMessage(user: User)
CustomAuthenticationProvider
+ encoder: PasswordEncoder
- userRepo: UserRepository
+ authenticate(authentication: Authentication):
UsernamePasswordAuthenticationToken
+ supports(authentication: Authentication): boolean
WebSecurityAdapter
- authProvider: CustomAuthenticationProvider
# configure(httpSecurity: HttpSecurity)
+ configAuthentication(auth: AuthenticationManagerBuilder)
BenchmarkRepository
+ existsByNumber(number: String): boolean
+ findByNumber(number: String): List<Benchmark>
+ findLatestByNumber(number: String): Benchmark
+ findAllLatest(): List<Benchmark>
+ findAll(): List<Benchmark>
+ save(benchmark: Benchmark): Benchmark
ResourceRepository
+ findByNumber(number: String): Photo
+ findByNumber(number: String): Topo
+ findById(id: Long): Photo
+ existsById(id: Long)
+ deleteById(id: Long)
+ saveResource(newResource: Resource): Resource
UserRepository
+ findById(id: String): User
+ existsById(id: String): boolean
+ save(user: User): User
JavaMailSender
+ send(mail: MailMessage)
PassowrdEncoder
+ encode(password: String): String
+ matches(password: String,
encodedPassword: String): boolean
CustomAuthenticationProvider
+ authenticate(auth: Authentication
): Authentication

1.1.1. Database

The H2 relational database management system (RDBMS) (Mueller and H2 Contributors) has been used to store the data. H2 RDBMS allows databases to be managed from within applications written in Java. It can be embedded in a Java application or operate in a client-server mode. When using the embedded mode, data is stored and maintained in a file, preserved when the application is closed and restarted. The software is available under an open-source Mozilla Public License 2.0 or the original Eclipse Public License.

During the pilot deployment, the H2 database management system has been embedded on the server, storing the data in a local file. Direct access to the database is available only to application administrators. In case of a deployment of the application for a larger number of users, the database will need to be hosted on a separate server for security and scaling purposes.

The information between the server and the database in the H2 platform is transmitted using the HTTP protocol. The communication architecture has been developed using the REST (**RE**presentational **S**tate **T**ransfer) paradigm (Tilkov), which is the standard protocol of communication between distributed systems. The REST software architecture defines the requirements for APIs (**A**pplication **P**rogramming **I**nterface) and the features they should have. The data between the Metrica client and server is sent via XML or JSON (JavaScript Object Notation) documents The identification of database resources is done using a Uniform Resource Identifier (URI).

1.1.2. Server API

Błąd! Nie można odnaleźć źródła odwołania. denotes several examples of methods available through the server API, along with their endpoints:

Table 2 Methods available through the server API

Function	Method	Address
Add a point	POST	/api/benchmarks
Display all points	GET	/api/benchmarks
Display a point with a given ID	GET	/api/benchmarks/{pointID}
Edit a point	PUT	/api/benchmarks/{pointID}
Add a description of the point	POST	/api/resources, • type: photo/topo, • pointID, • photo or topographic description
Deleting a point	DELETE	/api/resources/photos/{photoID}

1.1.3. Development of the server application

The server application has been developed with the Java programming language using the IntelliJ IDEA Integrated Development Environment (IDE), available under Apache 2 Licensed community edition (IntelliJ Company). It is available in two versions. The basic Community version allows the creation of applications in Java SE, while the Ultimate version supports Java EE and Spring. Students have free access to the Ultimate version, which supports the development of web applications using HTML and JavaScript without the need to install additional plug-ins.

1.1.4. Server hosting

A hosting service has been used to make the server efficient and reliable. The server is equipped with (OVHcloud Company): 1vCore processor (1 virtual core), 4GB RAM, 40GB of storage space for point data, and a 100 Mbps Internet connection. The server runs under the Debian operating system (from the Linux family). The parameters of the VPS server will allow it to support a few dozen clients simultaneously, which will be sufficient to run the pilot version of the application.

2. Metrica mobile application development

Metrica has been designed for the VCoP of surveyors to facilitate their work in the field. To fullfill its purpose, it should be intuitive and easy to use. It is essential not to take the surveyors' time to fill in much information, which could discourage them from simultaneously using and updating the data. At the same time, the application needed to be very intuitive so that surveyors could use it without any training, which could slow down the adaptation. The technology solutions presented in this section have been chosen based on these principles.

Android Studio environment, which is built on IntelliJ IDEA, has been used to develop the mobile application, and ready-made external libraries have been utilized to make the programming process easier and quicker:

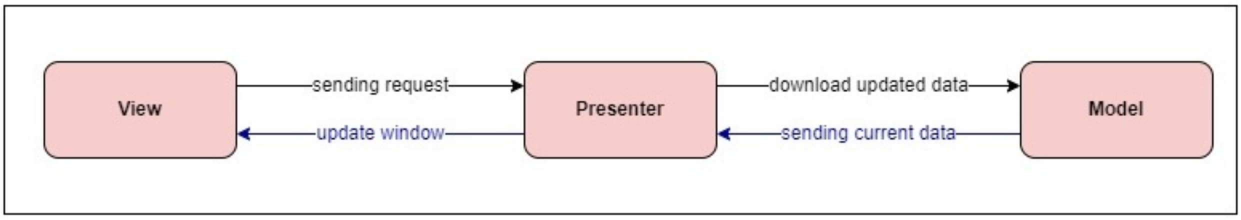
- **Retrofit** - a library that converts REST API methods into corresponding Java objects and vice versa
- **RxAndroid** - an extension of the **RxJava** (Reactive Extensions in Java) library. The library facilitates the development of programs based on reactive programming, allowing asynchronous, non-blocking data processing based on observed events as repeated sequences (AndroidCode)

2.1. Mobile application architecture and design patterns

Figure 6:

- Model - This is the layer that manages the data and makes it available by responding to requests from the presenter,
- View - is the layer responsible for presenting the data, displaying errors and animations that constitute the graphical user interface. It intercepts user interactions and sends them back in the form of requests to the presenter, awaiting further instructions,
- Presenter - is responsible for receiving requests from the view and communicating with the model layer, through which it sends a given response to the user interaction that is displayed in the view.

Figure 4 Model - View - Presenter pattern



Another pattern used is the Observer pattern
(

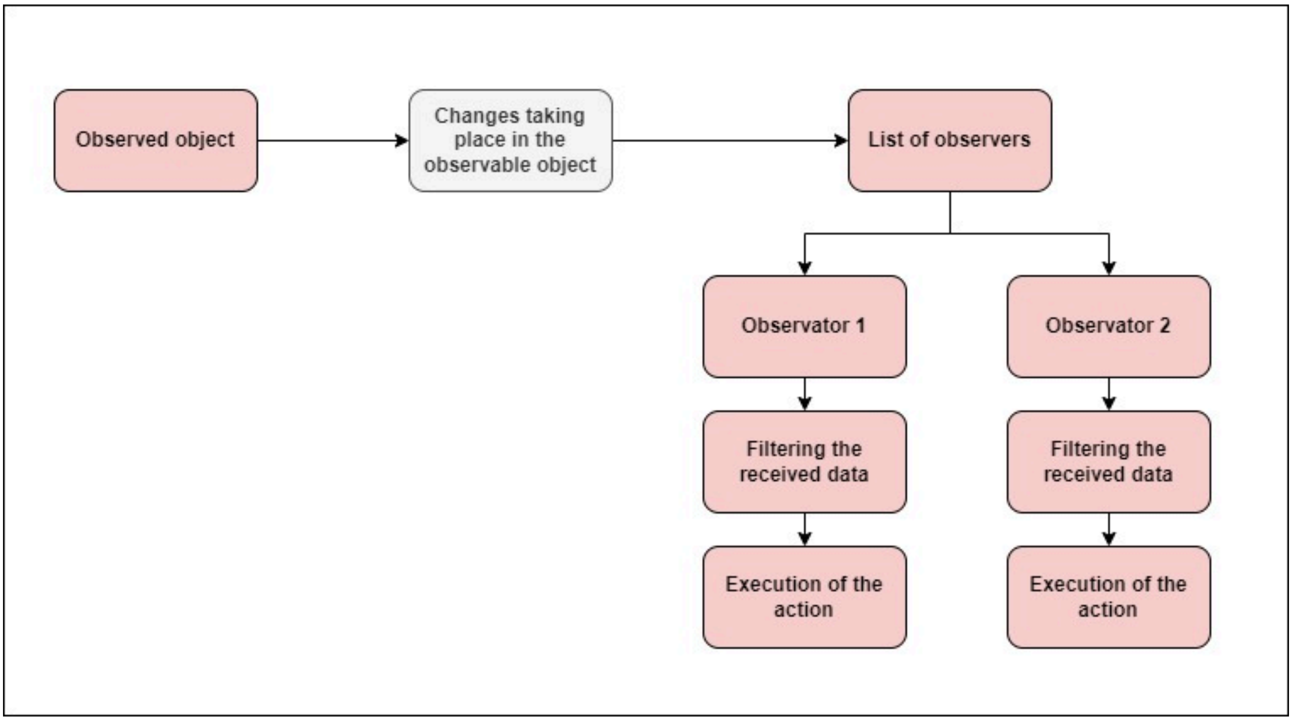


Figure 5). It is used where it is necessary to inform other objects of changes. This means that one object may have several observers who do not know about each other and receive information about the state of the observed object (Observer - Design Patterns, 2018).

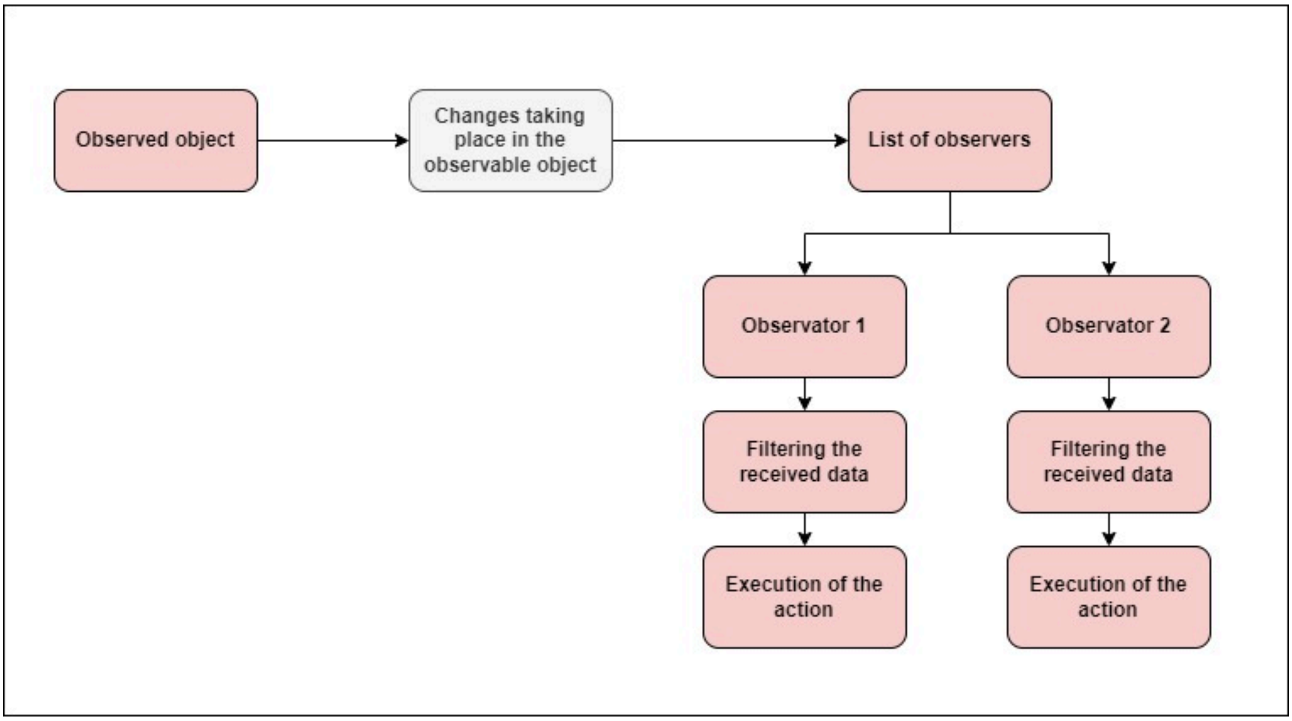
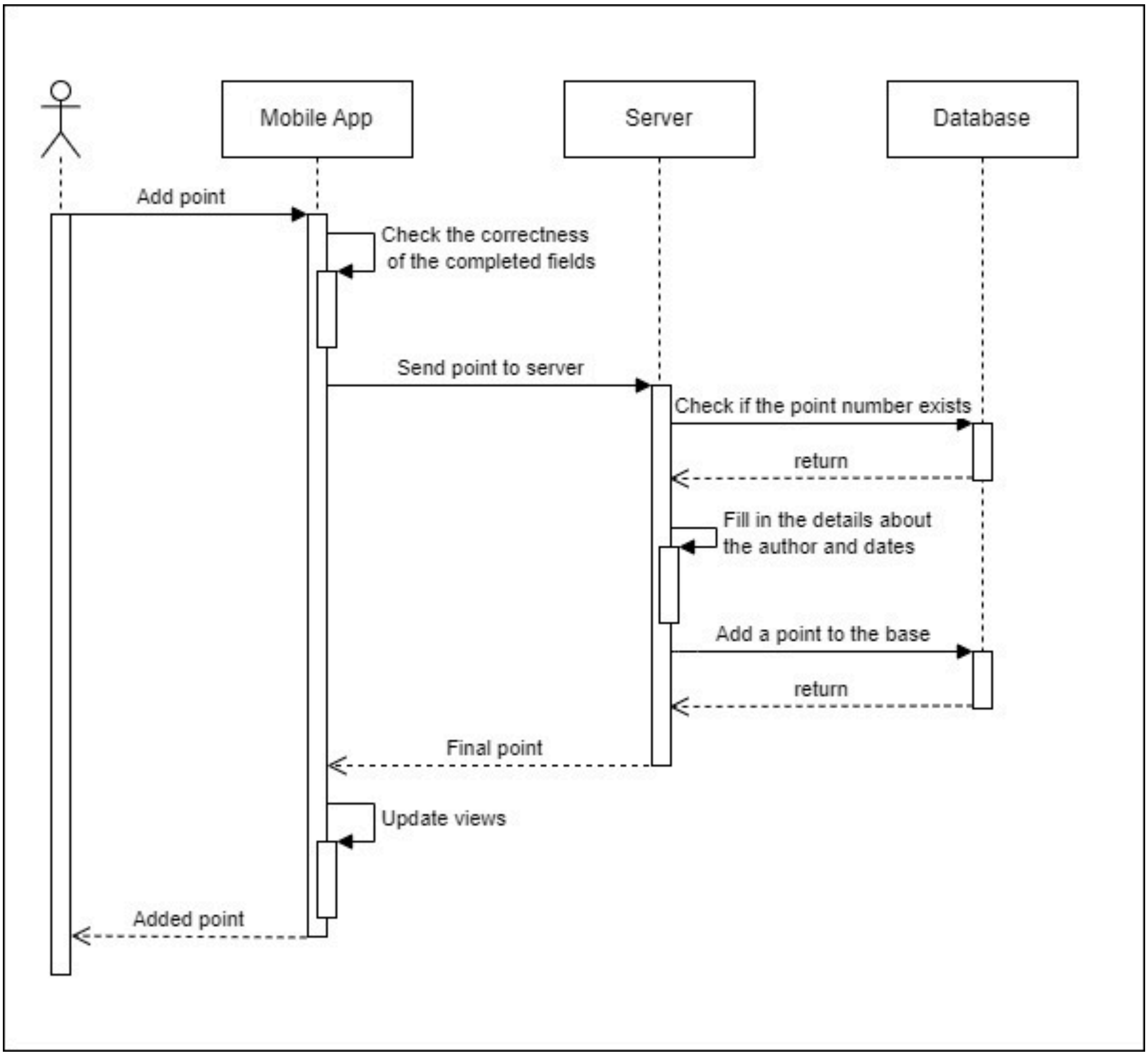


Figure 5 Observer pattern

Figure 10: adding a new point (Figure 6).

Figure 6 Data flow diagram - adding a point


Mobile App
Server
Database
Add point
Check the correctness of the completed fields
Send point to server
Check if the point number exists
return
Fill in the details about the author and dates
Add a point to the base
return
Final point
Update views
Added point

When the user presses the "Save point" button in the interface, the application checks the correctness of the filled-in fields. When all the conditions are reviewed and a positive result is obtained, the point is sent to the server, from which, in turn, a query is sent to the database as to whether a point with the given number already exists. If a point with such a number does not exist in the database, the confirmation is returned to the server, where data on the author and dates are completed. The whole data is subsequently sent to the database, which saves the point. A message is displayed to the user that the point has been added correctly.

2.1.1. Structure of Metrica mobile application

The structure of the Metrica application is based on packages (Fig. 7.11). In the application, we have three significant packages, which are subdivided into smaller packages:

- java,
- manifesto,
- res.

The *java* package consists of smaller packages that are responsible for interacting with the server and the application view:

- api - responsible for the communication of the application with the server,
- models - contains the models used in the application,

- uiDetails - includes code responsible for displaying point details, photo galleries and topographical descriptions, as well as editing point information,
- login - the package responsible for the login function in the application,
- listPoints - contains a class that creates the home screen of the application. The list of points appears in it, which is managed by listPointsFragment,
- mainMap - the package that is responsible for displaying points on the map,
- aboutAplication - contains a class responsible for displaying information about the application.
- security - responsible for protecting stored data.

The next main package is the *manifesto* package, which contains the AndroidManifest file responsible for configuring the application (e.g., setting the application window hierarchy).

The final package is the *res package,* which has also been divided into smaller packages:

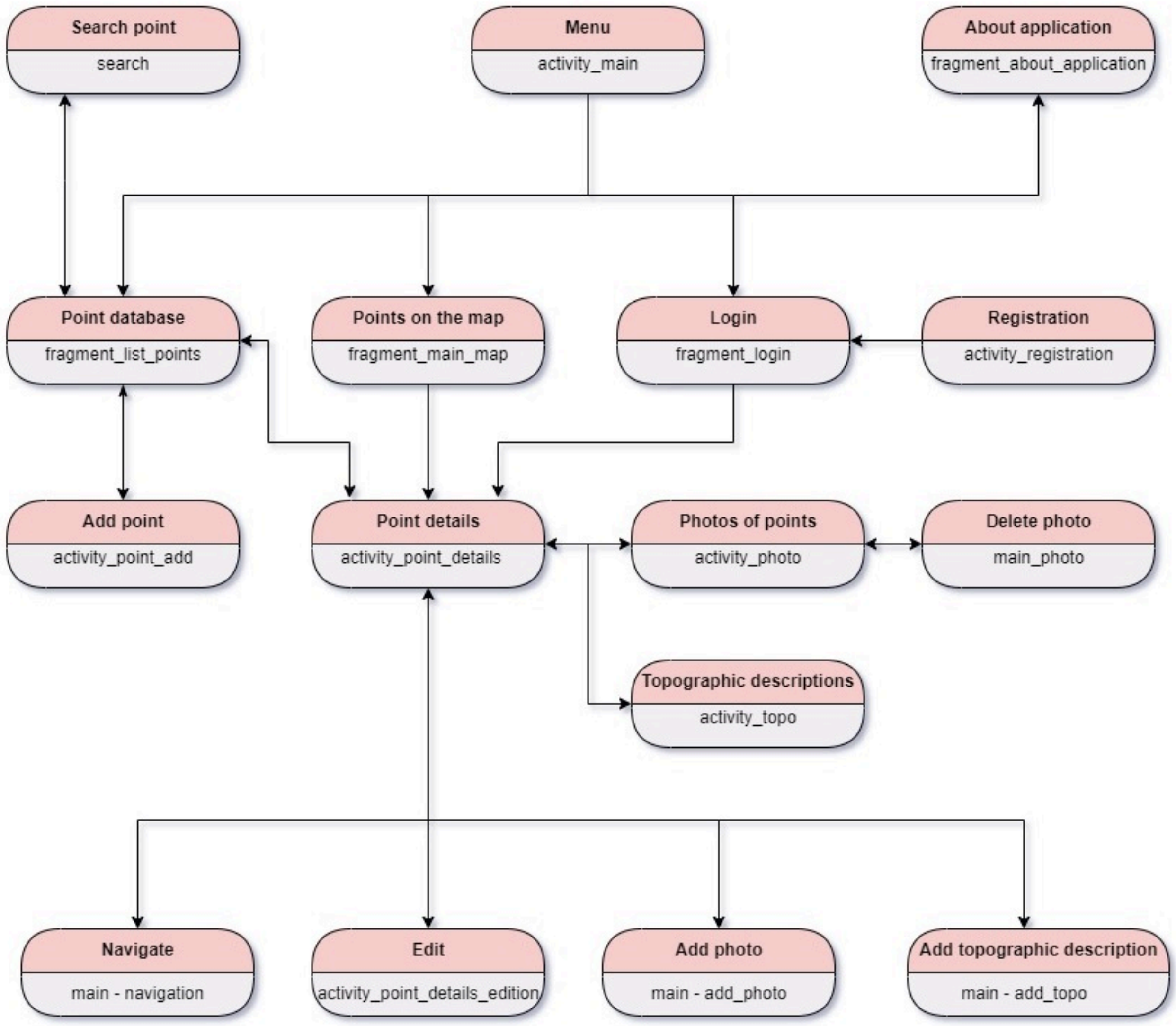
- drawable - stores the image files used in the application,
- layout - contains .xml files that are responsible for the layout of the components and the entire visualization of the application windows,
- menu - responsible for setting all the buttons on the top menu bar,
- values - manages styles, allowing them to be edited in individual application windows from a single location.

2.1.2. Presentation of the user interface

The application has been designed to be clear, intuitive and, above all, helpful to users. A diagram of these actions is included to show how the application works and the possible transitions between the various windows (

Figure 7).

Figure 7 Diagram of the transitions between the individual views in the application



All functions presented in the requirements analysis have been implemented in the mobile application. The look and feel of the final product can be demonstrated on a sample use case – viewing information about a point.

A window with information about the point is displayed by selecting a point from the list. There, the user can find information on the point's coordinates, topographic descriptions, photographs of the point's foundation, collected data on its technical condition or any comments on it (**Błąd! Nie można odnaleźć źródła odwołania.**).

In the upper right-hand corner of the window with details about the points, there is a drop-down menu in which the non-logged-in user can only select the *navigate* option.
This function redirects to Google Maps, which determines the shortest route to a selected point based on the user's location. It uses the coordinates of the point, which are chosen in an earlier window (**Błąd! Nie można odnaleźć źródła odwołania.**).

It is possible to select the type of coordinates displayed at any given time:

- approximate - coordinates from topographic descriptions received from CODGiK,
- navigational - coordinates obtained by measuring with a mobile device with GPS function,
- geodetic - coordinates obtained from a measurement by geodetic technology.

The logged-in user can use the add point function, which is located in the list of points in the database window. For this function to work correctly, the necessary fields must be filled in point number, approximate coordinates and height of the point. These point attributes are mandatory, as they remain uneditable at a later stage. This prevents the database from becoming "cluttered" with incorrect information. The required fields cover fundamental information that surveyors can access. Hence, completing these fields is not a problem for them and may discourage uninterested people who only want to harm the application.

Every user has access to the photo gallery and topographical descriptions. However, only those who are logged in can delete a photo. The delete function for topographic descriptions is unavailable, as these documents are only updated. Only application administrators have access to this activity to control the stored data in the database.

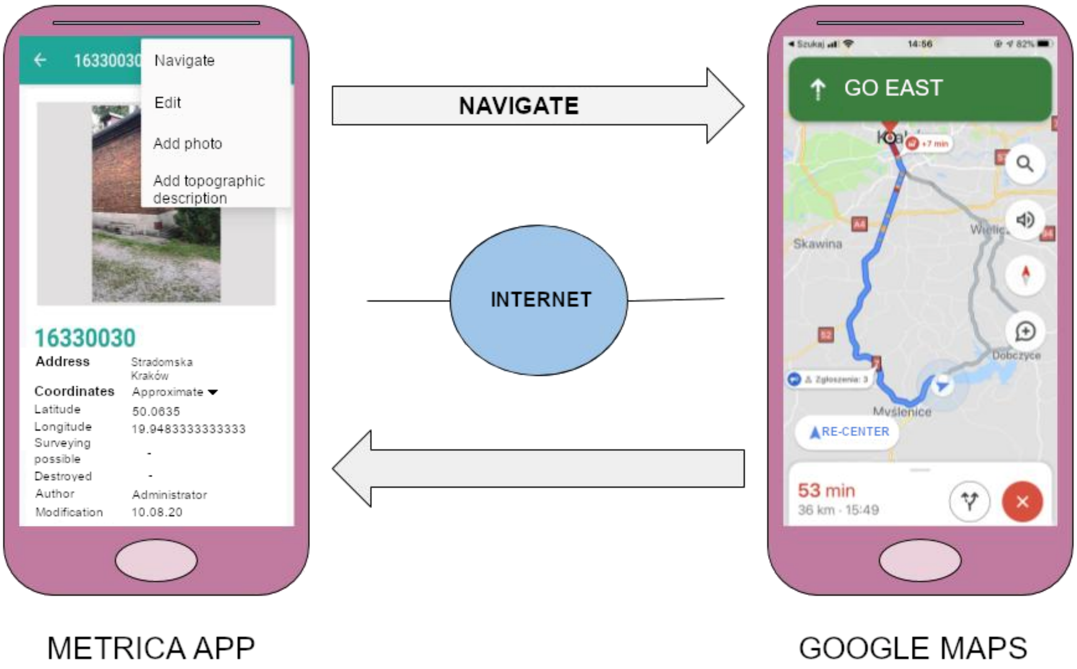
The approach in which the application designed for a VCoP has to be easy to use by the members but not too prone to manipulation by non-skilled parties – e.g., children of surveyors accessing their mobile phones while the parent is performing measurements or Internet vandals, is appropriate for the Web 2.0 concept.

2.1.3. Navigation and map service provider

The Android system takes advantage of Google services specifically dedicated to it, including Google Maps (Google Inc.)Using these services, Metrica can run Google Maps while passing additional parameters such as the destination coordinates (**Błąd! Nie można odnaleźć źródła odwołania.**). This uses objects of the Intent class, called Intents for short. Based on the information entered into the Intent, the Android system can launch another application located on the same device. The coordinates of the place from which navigation is started are not sent; instead, Google Maps will automatically use the current coordinates of the device. Intent is determined by the following parameters (Google Inc.):

- action - the action to be performed by the view that the Intent will take us to. In this case, it is ACTION_VIEW; this means that we want Google Maps to show us the route to a point,
- URI - the data on which the running view or application will operate. In this case, the URI is specified as google.navigation:q=widthGeographical,lengthGeographical. This means that our goal is to launch navigation to a point with the given coordinates,
- package - guarantees that the launched application will come from a specific package. Each application has its package. By specifying it in the Intent, we ensure that the Google Maps application will be launched, not another application with an action with the same name.

Figure 8 Integration of the navigation function in Metrica mobile application with Google Maps



16330030

Address	Stradomska Kraków
Coordinates	Approximate ▾
Latitude	50.0635
Longitude	19.9483333333333
Surveying possible	-
Destroyed	-
Author	Administrator
Modification	10.08.20

1.3. Implementation

The server is divided into packages, each containing classes responsible for different tasks. (Fig. 1.1. The *Model* package contains classes that describe objects with variables (they define the data model). The next package is *Controller*, which contains classes that implement the server's business logic, i.e. all the operations it can perform on the data and all other actions performed by the server. Thanks to the classes included in the Repository package, data can be written to the database regardless of the database management system used. In order to ensure that the data stored in the database is secure, a *Security* package has been created, which forces the user to register or log in in order to obtain access to selected functions. It also controls access to resources on the basis of permissions granted to the user. The Security package also includes the configuration used to ensure secure SSL communication (data sent between the client and the server is encrypted so that no one can intercept it).

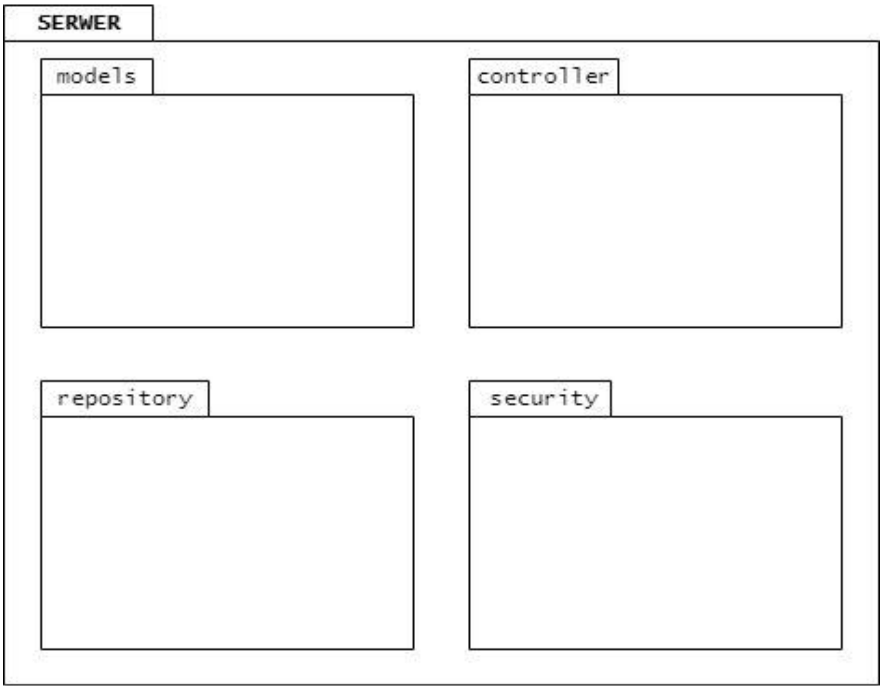


Fig. 1.1. *Server structure - package diagram (own elaboration)*

Using the Spring framework, we can take advantage of ready-to-use solutions on how to organise our work and code to avoid common mistakes and maintain code clarity. These are called design patterns. Spring MVC (Model-View-Controller) was used to create classes and methods that control the handling of HTTP requests. One of the capabilities of the Spring Framework is Request Mapping, which is done using annotations (Spring, 2020):

- **@GetMapping** - handles requests made using the GET method,
- **@PostMapping** - supports requests made using the POST method,
- **@PutMapping** - supports requests made using the PUT method,
- **@DeleteMapping** - supports requests made using the DELETE method.

The above annotations pass the HTTP request to the appropriate method, which is located in the respective controller. To access the data, a method argument must be provided that specifies the variable used when applying @RequestMapping (Spring Framework - Spring MVC, 2020):

- **@RequestParam** - the name of the parameter with the value is set in the access path e.g. *HTTP GET: http://.../benchmarks?number=23,*
- **@PathVariable** - the parameter value is set directly in the access path, e.g. *HTTP GET: http://....../benchmarks/12,*
- **@RequestBody** - data is sent in the request body in JSON format. The request body is the area used to transfer the data.

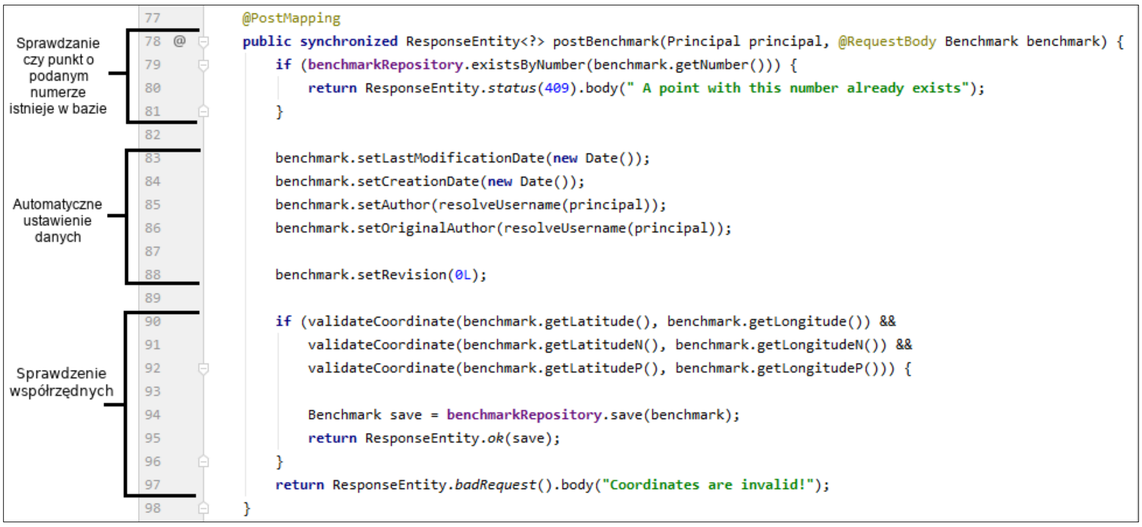
The main models in the server are: *Benchmark, Resource* and *User*. They describe the points, the images / topographic descriptions and the users respectively (Fig. 1.8). The class *BenchmarkId* serves as an identifier for the points. All versions of changes to the point information will be stored in the server. Each model has attributes describing it, in the example of the *Resources* class these are: *id*, *number*, *creationDate*, *author*, *type*, *photo*. These can be of different data types, in the server the following are used (JAVA START, 2020) :

- straight type:
  - double - floating point type, memory space taken up: 8 bytes, accuracy of stored number: max. 15 digits after the decimal point,
- object type:
  - String - belongs to the object type, in which strings of characters (text) are stored,
  - Boolean - a logical type, used to store two values: false or true,

- Date - a type used to store dates,
- Long - an integer that occupies 8 bytes in memory,
- Byte - an integer, occupying 1 byte of memory, its range: 128 to 127.

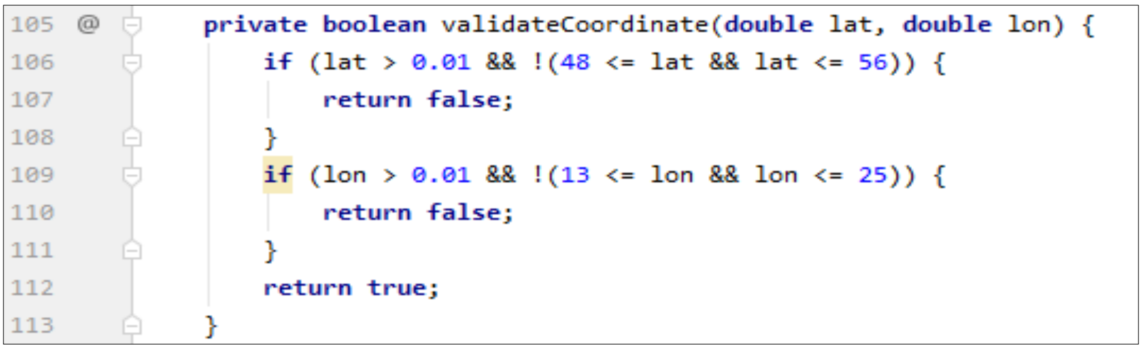
The *BenchmarkController* class performs all actions on the points. This class contains several methods to display all points, display a point with a given number, edit or add a new point. It is based on the *Benchmark* model. Data is downloaded and sent via the *BenchmarkRepository* and the *UserRepository* (Fig. 1.8).

The use of the @RequestMapping annotation in the thesis is shown in the example (Fig. 1.2). The method allows a new point to be added to the database. At the beginning of adding a point, it is checked whether a point with the given number already exists in the database (line of code: 79). If such a number already exists, an error with status 409 is returned with the message: "*A point with such a number already exists*" (line of code: 80). Otherwise, the point is saved provided the fields are completed correctly. It is worth noting that the fields concerning: date of last modification, date of creation, author of changes and original author are automatically retrieved from the server level (line of code: 83-86). When saving, the *validity* of the coordinates of the added point is also checked using the *validateCoordinate* method invoked in lines of code: 90-92 and shown in figure (Fig. 1.3). The coordinate values are checked corresponding to the range covering Poland.


Sprawdzanie czy punkt o podanym numerze istnieje w bazie
Automatyczne ustawienie danych
Sprawdzenie współrzędnych

```java
77  @PostMapping
78  public synchronized ResponseEntity<?> postBenchmark(Principal principal, @RequestBody Benchmark benchmark) {
79      if (benchmarkRepository.existsByNumber(benchmark.getNumber())) {
80          return ResponseEntity.status(409).body(" A point with this number already exists");
81      }
82
83      benchmark.setLastModificationDate(new Date());
84      benchmark.setCreationDate(new Date());
85      benchmark.setAuthor(resolveUsername(principal));
86      benchmark.setOriginalAuthor(resolveUsername(principal));
87
88      benchmark.setRevision(0L);
89
90      if (validateCoordinate(benchmark.getLatitude(), benchmark.getLongitude()) &&
91          validateCoordinate(benchmark.getLatitudeN(), benchmark.getLongitudeN()) &&
92          validateCoordinate(benchmark.getLatitudeP(), benchmark.getLongitudeP())) {
93
94          Benchmark save = benchmarkRepository.save(benchmark);
95          return ResponseEntity.ok(save);
96      }
97      return ResponseEntity.badRequest().body("Coordinates are invalid!");
98  }
```

***Fig. 1.2.** Adding a new point to the database - server (Own elaboration)*



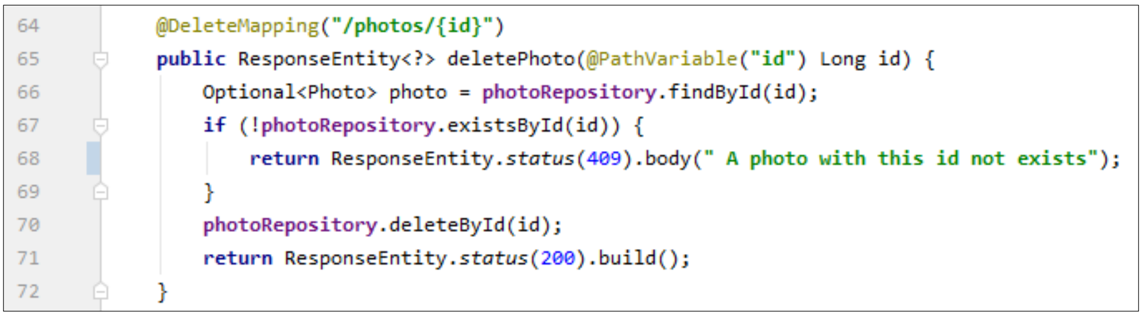
```java
private boolean validateCoordinate(double lat, double lon) {
    if (lat > 0.01 && !(48 <= lat && lat <= 56)) {
        return false;
    }
    if (lon > 0.01 && !(13 <= lon && lon <= 25)) {
        return false;
    }
    return true;
}
```

***Fig. 1.3.** Coordinate checking method - server (own elaboration)*

The next class in the server is *ResourcesController*, with which you can manage the resources for photos and topographic descriptions. It contains methods for adding or deleting a photo/topographic description. Also in this class, the Spring Framework has been used (e.g. @DeleteMapping) (Fig. 1.8).

The method for adding a photo or topographic description in this class checks whether a point with that number is in the *BenchmarkRepository*, where point data is stored. If the point exists a photo is added to the *ResourcesRepository,* which has a specific model (Resource class). One of the variables of this model is a unique id number.

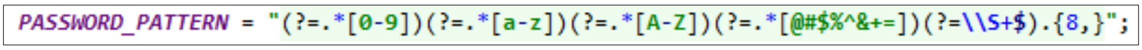
In the figure (Fig. 1.4) shows another method, which is the deletion of photos. It searches the photo resource based on id (code line: 67), if a photo with the given id number exists then it is deleted (code line: 70). Otherwise, a message is displayed: "The *photo with this id does not exist*" (line of code: 68).



```
64      @DeleteMapping("/photos/{id}")
65      public ResponseEntity<?> deletePhoto(@PathVariable("id") Long id) {
66          Optional<Photo> photo = photoRepository.findById(id);
67          if (!photoRepository.existsById(id)) {
68              return ResponseEntity.status(409).body(" A photo with this id not exists");
69          }
70          photoRepository.deleteById(id);
71          return ResponseEntity.status(200).build();
72      }
```

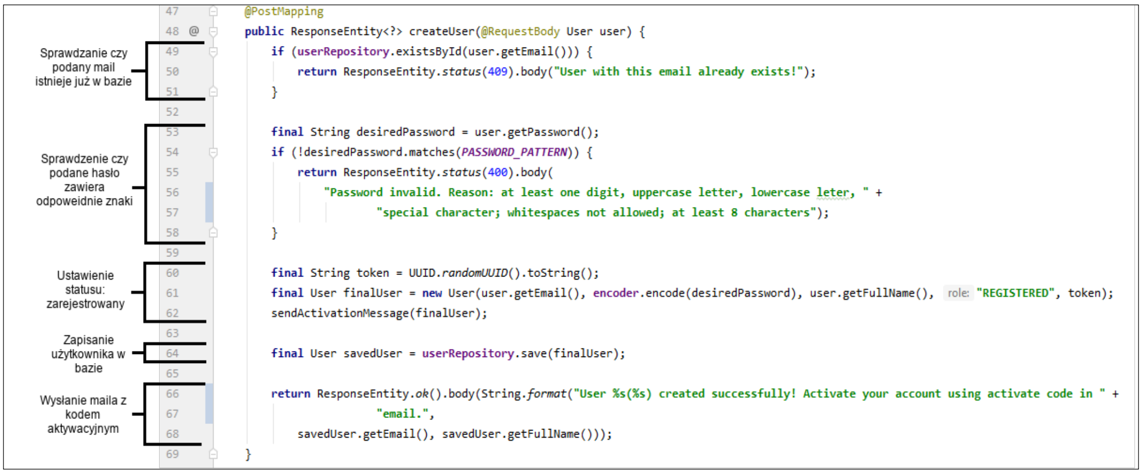
Fig. 1.4. *Method of deleting images - server (own elaboration)*

The SecurityController class is responsible for user-related activities. Among other things, it contains methods that allow you to register the user, change the password and activate the account. The user registration method is shown below (Fig. 1.6). It creates an account based on an email, a name (first and last name) and a password. The server checks whether the specified email already exists in the database, as one email cannot support multiple accounts (code line: 49-50). The password entered by the user should be structured appropriately, containing a lowercase and uppercase letter, a number and a special character (code line: 54-57). The total number of characters in the password should be a minimum of 8 (Fig. 1.4). Once the data have been completed correctly, the person is given the status of registered. To complete the registration and activate the account, use the activation code received to the e-mail address (Fig. 1.8).



```
PASSWORD_PATTERN = "(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=])(?=\\S+$).{8,}";
```

Fig. 1.5. *Regex imposing the format of the password to be created (Own elaboration)*

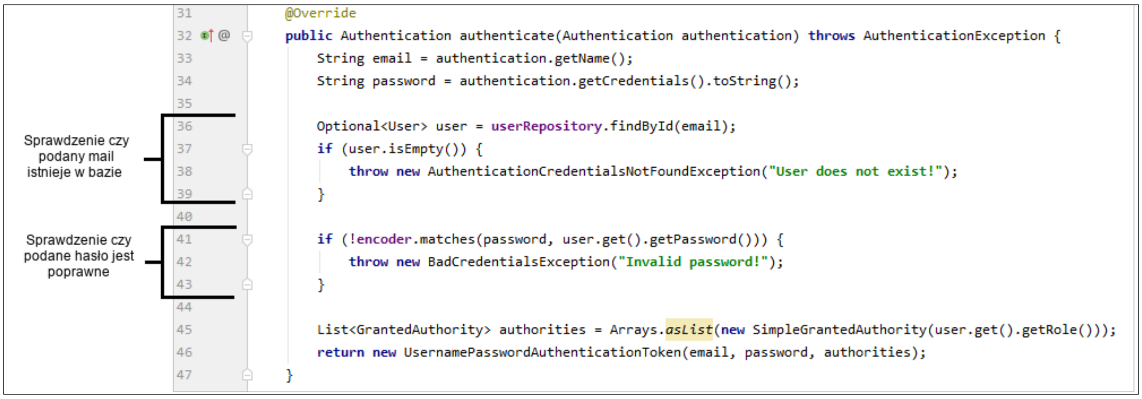


Sprawdzanie czy podany mail istnieje już w bazie
Sprawdzenie czy podane hasło zawiera odpoweidnie znaki
Ustawienie statusu: zarejestrowany
Zapisanie użytkownika w bazie
Wysłanie maila z kodem aktywacyjnym

```
47      @PostMapping
48      public ResponseEntity<?> createUser(@RequestBody User user) {
49          if (userRepository.existsById(user.getEmail())) {
50              return ResponseEntity.status(409).body("User with this email already exists!");
51          }
52
53          final String desiredPassword = user.getPassword();
54          if (!desiredPassword.matches(PASSWORD_PATTERN)) {
55              return ResponseEntity.status(400).body(
56                  "Password invalid. Reason: at least one digit, uppercase letter, lowercase leter, " +
57                          "special character; whitespaces not allowed; at least 8 characters");
58          }
59
60          final String token = UUID.randomUUID().toString();
61          final User finalUser = new User(user.getEmail(), encoder.encode(desiredPassword), user.getFullName(),  role: "REGISTERED", token);
62          sendActivationMessage(finalUser);
63
64          final User savedUser = userRepository.save(finalUser);
65
66          return ResponseEntity.ok().body(String.format("User %s(%s) created successfully! Activate your account using activate code in " +
67                          "email.",
68              savedUser.getEmail(), savedUser.getFullName()));
69      }
```

Fig. 1.6. *User registration method - server (own elaboration)*

The CustomAuthenticationProvider class (Fig. 6.12) is responsible for the login function, which verifies whether a user with a given email exists in the database (line of code: 37-38). If it does exist, it checks that the password entered by the user after encoding matches the password given during registration (line of code: 41-42). This is stored as strings of characters which are hashed to guarantee data protection. Administrators do not have access to user passwords; in the event of a data leak, they are protected (encrypted).



```
31          @Override
32          public Authentication authenticate(Authentication authentication) throws AuthenticationException {
33              String email = authentication.getName();
34              String password = authentication.getCredentials().toString();
35
36              Optional<User> user = userRepository.findById(email);
37              if (user.isEmpty()) {
38                  throw new AuthenticationCredentialsNotFoundException("User does not exist!");
39              }
40
41              if (!encoder.matches(password, user.get().getPassword())) {
42                  throw new BadCredentialsException("Invalid password!");
43              }
44
45              List<GrantedAuthority> authorities = Arrays.asList(new SimpleGrantedAuthority(user.get().getRole()));
46              return new UsernamePasswordAuthenticationToken(email, password, authorities);
47          }
```

Sprawdzenie czy podany mail istnieje w bazie
Sprawdzenie czy podane hasło jest poprawne

Fig. 1.7. *User login method - server (own elaboration)*

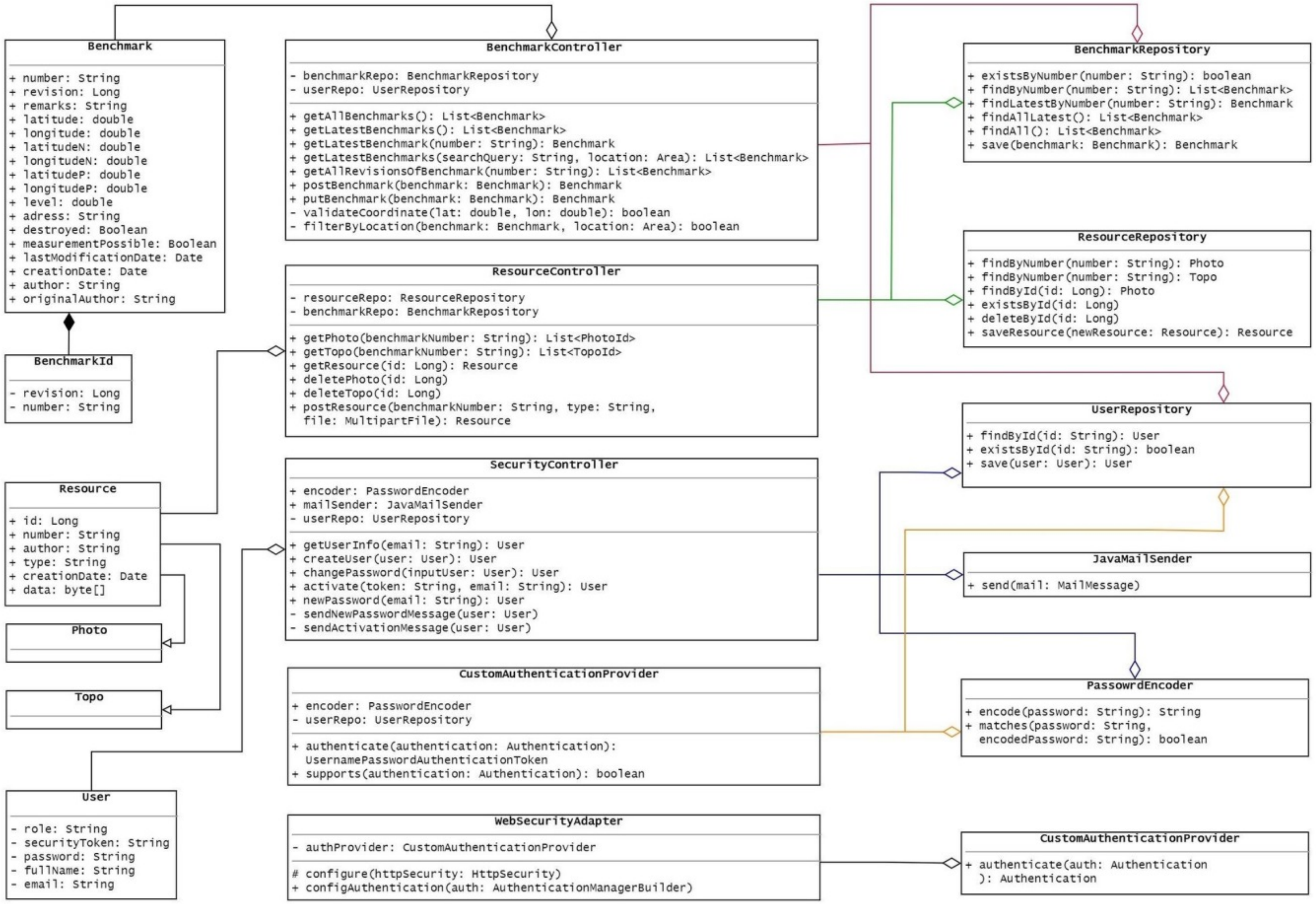

Benchmark
+ number: String
+ revision: Long
+ remarks: String
+ latitude: double
+ longitude: double
+ latitudeN: double
+ longitudeN: double
+ latitudeP: double
+ longitudeP: double
+ level: double
+ adress: String
+ destroyed: Boolean
+ measurementPossible: Boolean
+ lastModificationDate: Date
+ creationDate: Date
+ author: String
+ originalAuthor: String
BenchmarkId
- revision: Long
- number: String
Resource
+ id: Long
+ number: String
+ author: String
+ type: String
+ creationDate: Date
+ data: byte[]
Photo
Topo
User
- role: String
- securityToken: String
- password: String
- fullName: String
- email: String
BenchmarkController
- benchmarkRepo: BenchmarkRepository
- userRepo: UserRepository
+ getAllBenchmarks(): List<Benchmark>
+ getLatestBenchmarks(): List<Benchmark>
+ getLatestBenchmark(number: String): Benchmark
+ getLatestBenchmarks(searchQuery: String, location: Area): List<Benchmark>
+ getAllRevisionsOfBenchmark(number: String): List<Benchmark>
+ postBenchmark(benchmark: Benchmark): Benchmark
+ putBenchmark(benchmark: Benchmark): Benchmark
- validateCoordinate(lat: double, lon: double): boolean
- filterByLocation(benchmark: Benchmark, location: Area): boolean
ResourceController
- resourceRepo: ResourceRepository
- benchmarkRepo: BenchmarkRepository
+ getPhoto(benchmarkNumber: String): List<PhotoId>
+ getTopo(benchmarkNumber: String): List<TopoId>
+ getResource(id: Long): Resource
+ deletePhoto(id: Long)
+ deleteTopo(id: Long)
+ postResource(benchmarkNumber: String, type: String,
file: MultipartFile): Resource
SecurityController
+ encoder: PasswordEncoder
+ mailSender: JavaMailSender
- userRepo: UserRepository
+ getUserInfo(email: String): User
+ createUser(user: User): User
+ changePassword(inputUser: User): User
+ activate(token: String, email: String): User
+ newPassword(email: String): User
- sendNewPasswordMessage(user: User)
- sendActivationMessage(user: User)
CustomAuthenticationProvider
+ encoder: PasswordEncoder
- userRepo: UserRepository
+ authenticate(authentication: Authentication):
UsernamePasswordAuthenticationToken
+ supports(authentication: Authentication): boolean
WebSecurityAdapter
- authProvider: CustomAuthenticationProvider
# configure(httpSecurity: HttpSecurity)
+ configAuthentication(auth: AuthenticationManagerBuilder)
BenchmarkRepository
+ existsByNumber(number: String): boolean
+ findByNumber(number: String): List<Benchmark>
+ findLatestByNumber(number: String): Benchmark
+ findAllLatest(): List<Benchmark>
+ findAll(): List<Benchmark>
+ save(benchmark: Benchmark): Benchmark
ResourceRepository
+ findByNumber(number: String): Photo
+ findByNumber(number: String): Topo
+ findById(id: Long): Photo
+ existsById(id: Long)
+ deleteById(id: Long)
+ saveResource(newResource: Resource): Resource
UserRepository
+ findById(id: String): User
+ existsById(id: String): boolean
+ save(user: User): User
JavaMailSender
+ send(mail: MailMessage)
PassowrdEncoder
+ encode(password: String): String
+ matches(password: String,
encodedPassword: String): boolean
CustomAuthenticationProvider
+ authenticate(auth: Authentication
): Authentication

Fig. 1.8. *Class diagram - server (own elaboration)*

1.4. Server hosting

A hosting service was used to make the server efficient and reliable. The server was placed on the hard disk of a VPS (Virtual Private Server). Its principle can be presented as a physical server (machine) (dedicated), which is divided into several smaller virtual machines. Server administrators are given partial access to the resources of the dedicated server. This is done using cloud technology (Madej, 2020). In order for the data to be made available to all users, the server must be connected to a network with a public IP. Due to the technical requirements, the costs of maintaining such a link would be very high. The solution to the above problems turned out to be a virtual private server VPS. The server is equipped with (OVHcloud, 2020):

- 1vCore processor (1 virtual core),
- 4GB RAM,
- 40GB storage space for point data,
- connection 100 Mbps.

The server runs under the Debian operating system (from the Linux family). The CPU and network usage graphs for the VPS server are shown below (Fig. 6.14, **Błąd! Nie można odnaleźć źródła odwołania.**). The results are shown for data stored on the server in the test version of the Metrica application. Inbound transfer is defined as the amount of data received by the server from the user, while outbound transfer is defined as the amount of data sent from the server to the user.

The parameters of the VPS server will allow it to support a few dozen clients, which will be sufficient to run the initial version of the application, since, as you can see from the charts above, the CPU and network usage is not high.

2. Metrica mobile application

The title application is aimed at surveyors to facilitate their work in the field. To fulfil its purpose, it should be intuitive and easy to use. It is important not to take the surveyors' time to fill in a lot of information, which could discourage them from using and updating the data at the same time. This is why the solutions presented in this chapter were chosen.

2.1. Tools and technologies

2.1.1. Android Studio

When developing applications for the Android operating system, the Android Studio environment, which is built on Intellij IDEA, can be used. Each project to be set up has a specific structure (folders) (Developers, 2020):

- **manifest** - this contains the AndroidManifest.xml file responsible for configuring the application,

- **java** - contains the source code of the application,
- **res -** contains resources, including but not limited to folders:
    - layout - XML files that are responsible for the layout of the components displayed on the screen,
    - drawable - contains image files, bitmaps (.png, .jpg, .gif),
    - values - responsible, among other things, for the colours and styles used in the application; thanks to this catalogue, the values of the application are collected in one place, which makes it easier to change them (for example to provide a different language version).

The user interface of an application consists of components, the most important of which are Activity and Fragment. The Activity class can be created via a template or manually by creating all the components from scratch. The *AndroidManifst.xml* file allows you to manage them. To avoid creating multiple Activity windows for each activity, Fragment objects can be used for this purpose. They ensure that the content of the screen changes and that the code is easily used (Trebilcox-Ruiz, 2016).

2.1.2. Libraries used

Ready-made external libraries were used to make the programming process easier and quicker:

- **Retrofi**t

  It is a library that converts REST API methods into corresponding Java objects and vice versa. It facilitates and accelerates the handling of requests made to HTTP REST services (GET, PUT, POST, DELETE). One of the main layers that the retrofit is responsible for is the POJO (Plain Old Java Object) model class. It is formed by field getters and setters, whose most important purpose is to store information. Such objects are easily converted from and to Json format (Chodorek, 2016).

- **RxAndroid and RxJava**

The Android system has the RxAndroid module, which is an extension of the RxJava (Reactive Extensions in Java) library. The library facilitates the development of programmes based on reactive programming, a type of programming that allows for asynchronous, non-blocking processing of data based on events, observed, as repeated sequences of (AndroidCode, 2019). One of its tasks is to handle multithreading, i.e. the ability to perform different tasks at the same time. For this, it uses various schedulers - objects that manage a pool of threads. Thanks to them, many activities are performed automatically and the user does not need to know the details of how this is done. This means that building and developing projects that process data asynchronously and multithreaded is easier thanks to ready-made solutions for handling communication between the main thread and background tasks. (RxJava in Android - Fundamentals. Ch. 1, 2019).

Android Studio allows projects to be imported and built through the **Gradle** tool. It has its own specific language that simplifies the solution of the tasks responsible for building a project. One of the basic actions that Gradle performs is the compilation of source code and the creation of a JAR (Java Archive) file that contains the compiled classes. Its operation is based on the configuration files that are associated with the objects that are created during the project building process. It consists of three stages (Pelgrims, 2015):

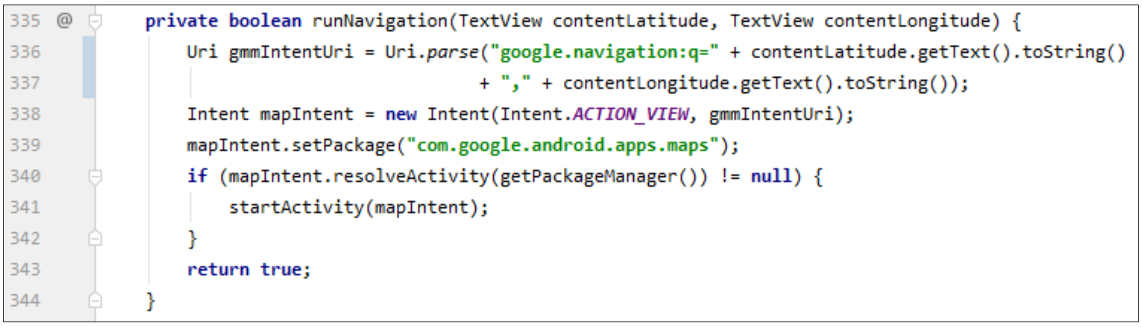
- initialisation - an object is created for each project and a settings.gradle file is created in which the project settings can be configured,

- configuration - objects created in the first stage are subject to configuration via build.gradle files,
- execution - in this phase, Gradle decides on the order and requirements of the tasks that were created in the previous stages.

2.1.3. Navigation and map service provider

The Android system has Google services specifically dedicated to it, one of which that has been used in this thesis is Google Maps (Android (operating system), 2020). The way the two programmes interact is by being able to run one application from the other, while passing additional parameters such as the coordinates of the destination to the Google Maps application (**Błąd! Nie można odnaleźć źródła odwołania.**). This uses objects of the Intent class, called Intents for short. The Android system, based on the information entered into the Intent, can launch another application located on the same device. In this case, it is about launching the Google Maps application and sending it the geographical coordinates of the destination. The coordinates of the place from which you start navigation are not sent to the application; if they are not, Google Maps will automatically use the current coordinates of your device. Intent is determined by the following parameters (Fig. 2.1) (Google Maps Intents for Android, 2020):

- action,
- The action to be performed by the view that the Intent will take us to. In this case it is ACTION_VIEW, this means that we want Google Maps to show us the route to a point,
- URI - the data on which the running view or application will operate. In this case, the URI is specified as google.navigation:q=widthGeographical,lengthGeographical. This means that our goal is to launch navigation to a point with the given coordinates,
- package - guarantees that the launched application will come from a specific package. Each application has its own package; by specifying it in the Intent, we guarantee that the Google Maps application will be launched, and not another application which has an action with the same name.



```
    private boolean runNavigation(TextView contentLatitude, TextView contentLongitude) {
        Uri gmmIntentUri = Uri.parse("google.navigation:q=" + contentLatitude.getText().toString()
                                            + "," + contentLongitude.getText().toString());
        Intent mapIntent = new Intent(Intent.ACTION_VIEW, gmmIntentUri);
        mapIntent.setPackage("com.google.android.apps.maps");
        if (mapIntent.resolveActivity(getPackageManager()) != null) {
            startActivity(mapIntent);
        }
        return true;
    }
```

Fig. 2.1. *Navigation to a specific point - application (own elaboration)*

2.2. Package diagram

The structure of the Metrica application is based on packages (Fig. 7.11). As already presented in the 'Server' chapter, packages consist of different classes and are responsible for all kinds of activities that are thematically related. In the application, we have three major packages, which are subdivided into smaller packages:

- java,
- manifesto,
- res.

The *java* package consists of smaller packages that are responsible for interacting with the server and the application view:

- api - responsible for the communication of the application with the server,
- models - contains the models used in the application,
- ui:
    - details - includes code responsible for displaying point details, photo galleries and topographical descriptions, as well as editing point information,
    - login - the package responsible for the login function in the application,
    - listPoints - contains a class that creates the home screen of the application, the list of points appears in it, which is managed by listPointsFragment,
    - mainMap - the package that is responsible for displaying points on the map,
    - aboutAplication - contains a class responsible for displaying information about the application.
- security - responsible for protecting stored data.

The next main package is the *manifest* package, it contains the androidManifest file responsible for configuring the application (e.g. setting the application window hierarchy).

The final package is the *res package,* which has also been divided into smaller packages:

- drawable - stores the image files used in the application,
- layout - contains .xml files that are responsible for the layout of the components and the entire visualisation of the application windows,
- menu - responsible for setting all the buttons on the top menu bar,
- values - manages styles allowing them to be edited in individual application windows from a single location.

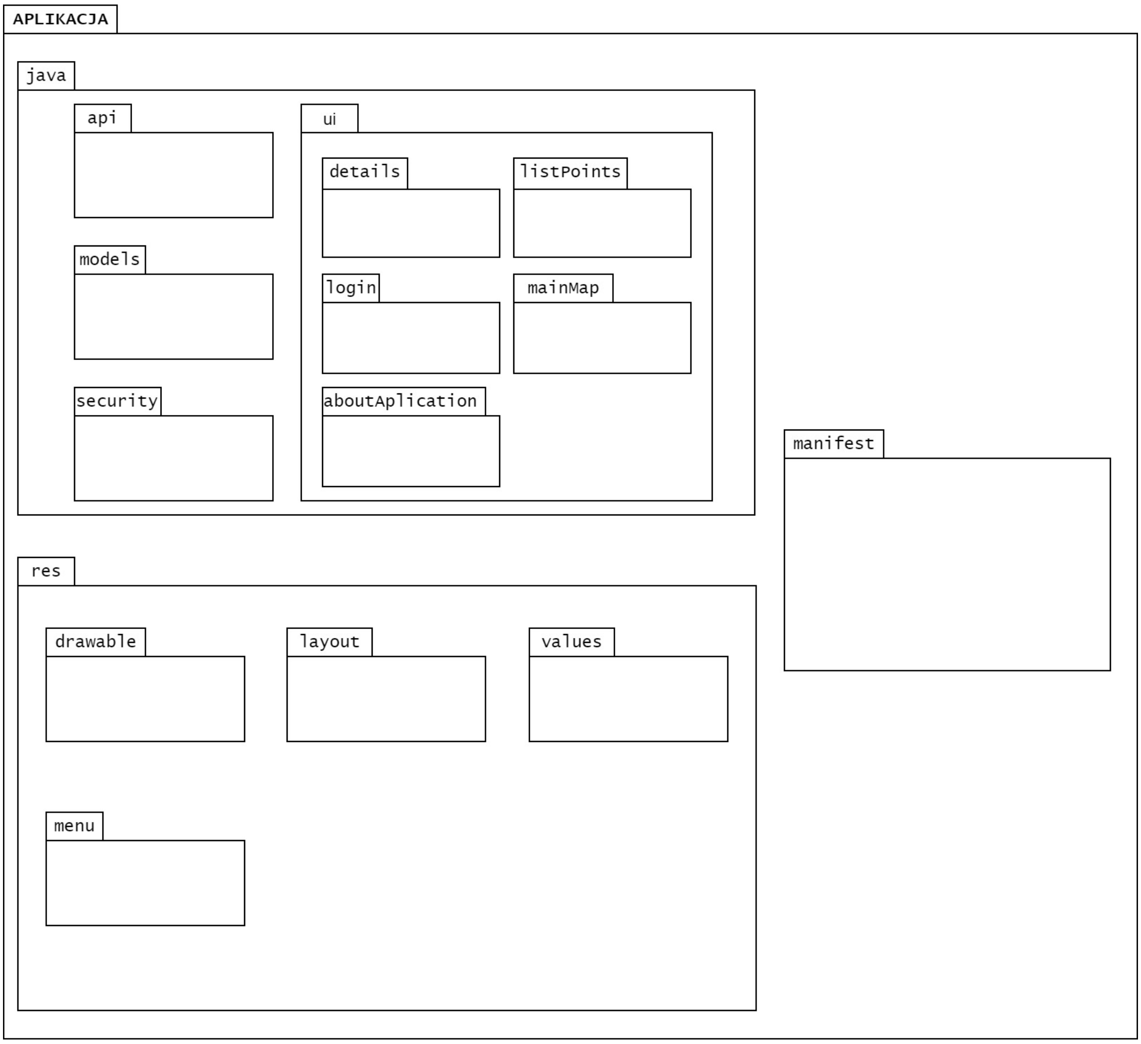


Fig. 2.2. *Application structure - package diagram (Own elaboration)*