Atlas: Software System for Monitoring and Reserving Free Parking Spaces

Ágnes-Kriszta Szabó
Babeş–Bolyai University
Cluj–Napoca, Romania
agnes_kriszta@yahoo.com

Zalán Ráduly
Codespring
Cluj–Napoca, Romania
raduly.zalan@codespring.ro

Örs-Krisztián Patakfalvi
Codespring
Cluj–Napoca, Romania
patakfalvi.krisztian@codespring.ro

Csaba Sulyok
Babeş–Bolyai University
Cluj–Napoca, Romania
sulyok.csaba@cs.ubbcluj.ro

Károly Simon
Babeş–Bolyai University
Cluj–Napoca, Romania
ksimon@cs.ubbcluj.ro

***Abstract*—Finding available parking spaces represents a significant problem for drivers in dense urban areas. The searching process results in substantial additional costs but also contributes considerably to pollution and road congestion. The Atlas project aims to facilitate the process of searching for parking spots.**

The software system uses data from existing surveillance cameras to list both designated and unassigned parking spaces while tracking the number of drivers navigating to a given spot, therefore it does not require expensive infrastructure. The main parts of the system are the central server, the mobile application and the computer vision module. The latter is responsible for the processing of images taken by urban surveillance cameras.

The paper presents the structure of the project, the details of the implementation, the tools and technologies used during the development process, and demonstrates the usage of the software.

***Index Terms*—parking space, parking spot, mobile application, surveillance camera, parking space detection**

I. Introduction

Drivers or public transport users may experience the inconvenience and frustration caused by traffic jams on a daily basis. From an environmental point of view, road congestion is a significant source of fuel and carbon dioxide emissions, as evidenced by a 2012 study by the International Parking Institute [1]. The same study estimates that "30% of the traffic in any city is people in cars searching for parking". According to a 2011 case study conducted by Siemens [2], "drivers looking for a parking space account for over 40 percent of all inner-city traffic" on average days, but this number may soar on the weekends or public holidays; around Christmas, it can reach even up to 90%.

There are already various attempts to solve the problem caused by the lack of parking spaces. Bosch uses ground-mounted ultrasonic sensors [3] to gather information about free parking spaces. Similar technology is proposed by a 2012 study [4], which aims to "find vacant spaces in a car park in a shorter time", as well as to detect car park occupancy and improper parking. Another main category of parking management systems is the image-based parking space detection, which may not require the installation of special devices. One representative of this category concentrates on tracking parking spaces in underground and indoor environments [5] "by fusing sensors already mounted on mass-produced vehicles". Another project intends to detect unoccupied parking spots "by using motion stereo-based 3D reconstruction" [6] paired with a rear-view fisheye camera mounted on automobiles. Besides reusable/global ventures, certain cities or districts may propose localized solutions. One such endeavour is the Cluj Parking [7] mobile application, initiated by the Mayor's Office of Cluj-Napoca, which provides real-time status information on seven barrier car parks within the city. Another mobile application, yeParking [8], allows parking lot owners with a subscription to temporarily lend their unused spot for free.

The Atlas project differs from these solution attempts in that it takes into consideration both designated and unassigned parking spaces at the roadside. Parking spots are detected with object recognition using neural networks, though the current paper focuses on the software system part of the project. The application makes it possible to list and reserve parking spaces by evaluating areas monitored by existing surveillance cameras, thus the installation does not require expensive infrastructure extensions, the creation of new parking spaces and/or a lot of human resources.

II. Functionalities

To access the services provided by the application, it is required of the user to log into the application with their Google account. Enabling the user to use their Google account speeds up the authentication process as it saves them from the registration steps and the creation and maintenance of a new account. Users of the application can have three different roles: guest, authenticated user and administrator.

The standard authenticated user has all the functionalities that can help in searching and reserving parking spots. The latter functionality consists of the reduction of available parking spots on the server-side based on the number of drivers navigating to the selected parking place. Among other things, authenticated users have the right to retrieve data about parking spaces and surveillance cameras, retrieve and modify data belonging to their users, and create or cancel reservations. They also have the option to filter the parking spaces and to retrieve the distance between their current position and their destination.

Besides the same general user rights, administrators may also add new cameras and parking spaces to the system or modify existing ones. Furthermore, they can also access user data and reservations if needed.

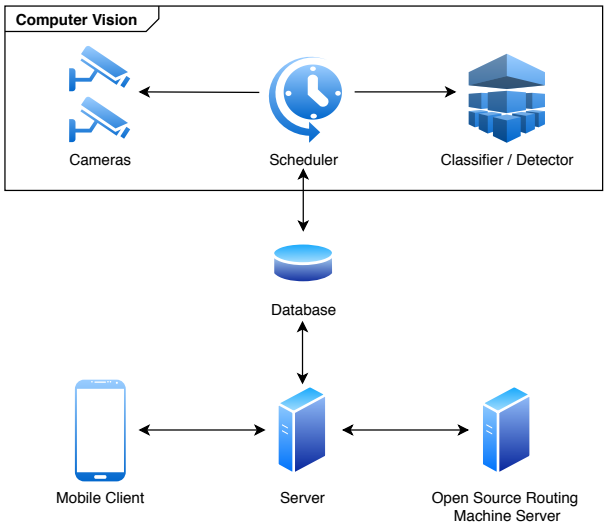

Fig. 1. The relationship between the modules of the software system.

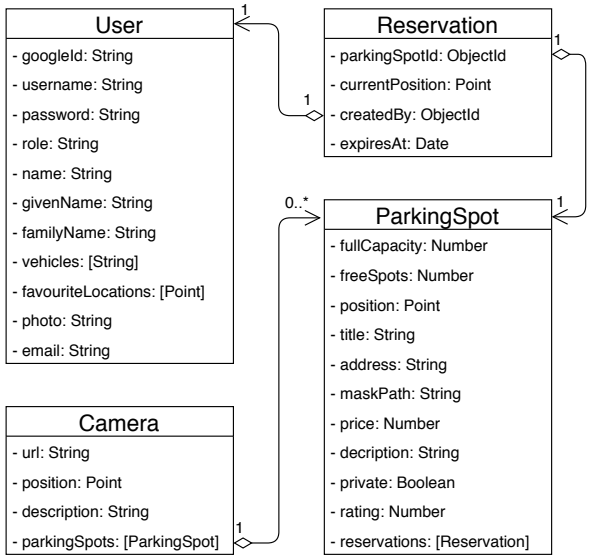

Fig. 2. Local objects corresponding to the four main schemas defined on server-side and the connections between them.

III. Architecture

The software system is built around three main parts: the central server, the mobile application, and the module responsible for computer vision. The relationships between the modules are illustrated in Fig. 1.

The current paper considers the computer vision module as an external one, the presentation of which is not included in details. By processing the images taken by urban surveillance cameras, there are two different approaches to detect free parking spots.

The first approach is to use a Convolutional Neural Network (CNN) in order to classify only a cropped out parking spot image from a camera, which concludes whether there is a car occupying the parking space or not. By processing every parking spot picture from an area, it can predict the total number of available spots. This solution leads to a more precise prediction in overall, because CNNs are accurate in large-scale video classification [9], but requires more time to process every cropped out parking spot image.

The other solution is to use object recognition with neural networks to predict the currently appearing cars at a parking area.

The scheduler component provides the images to the classifier or to the detector, retrieving them at intervals from the cameras entered in the database. After the predictions are done, the scheduler updates the parking spaces stored in the database with the number of currently available spots.

This database is used by the central server, which is responsible for serving the clients belonging to the system. The server is also responsible for authentication, for managing and forwarding the data stored in the database and for establishing a connection with the Open Source Routing Machine (OSRM) server. The mobile application included in the software system, which is available on both Android and iOS devices, allows users to list and reserve free parking spaces by communicating with the application server.

IV. The Application Server

The application server is a Representational State Transfer (REST) API [10], which is responsible for serving the parking space related requests from the client applications.

The structure of the server follows the principles of the multi-tier architecture [11], according to which the following layers can be separated: data model, data access layer and Application Programming Interface (API). By splitting the application server into layers, the development can be done in a modular way, so the system can be easily maintained and improved. Circular dependencies may also be avoided due to the layers communicating exclusively in one direction.

This chapter presents the different architectural layers and security details.

A. Data Model and Access

The data managed by the server is stored in a document-based database, namely in MongoDB [12]. To determine the structure of the documents storing the data, the following schemas are defined: User, Camera, ParkingSpot and Reservation; two auxiliary schemas are used to specify geographical positions (Point) and to describe polygons formed by coordinates (Polygon). The local objects corresponding to the four main schemas and their relationships are illustrated in Fig. 2.

The *Users* collection stores information about users signed in with a Google account. With their consent, Google Auth API provides their e-mail address, name, Google ID and profile picture. Additional fields defined in the schema are given by default or a value generated based on the previous ones: as a role, each newly logged-in user is given simple user rights, while their username is generated from their name.

The *Cameras* collection stores information about the surveillance cameras registered in the system, including the

geographical location of the camera, specified with a Point object, and a list of parking spaces under its monitoring.

The *ParkingSpots* collection contains common data which describes the parking spaces, such as GPS location, address, name, maximum capacity and the number of currently available spots. The latter is periodically set by the scheduler with the prediction of free spots evaluated by the detector or classifier.

The *Reservations* collection is defined by four properties: the parking space where the reservation is made, the user who made the reservation, the expiration date and the user's initial position. Using one of the advantages of the MongoDB, the data stored in the Reservations collection expires at the specified date, thus reservations are automatically removed and no further effort is required for their management.

Data access and manipulation is implemented by the Mongoose framework. This is an Object Document Mapping (ODM) framework, which allows performing operations on the server with objects that match the documents in the database. The created Mongoose models provide access to the data, thus no MongoDB scripts are required for the *Create, Read, Update, Delete* (CRUD) operations. These basic operations represent a convention system for accessing/modifying resources that facilitates persistent data storage. The implementation of the above-mentioned basic operations is performed by the controller layer, which also provides a response to the client in case of successful data retrieval or modification. The controllers are closely related to the decorator layer, which follows the decorator pattern. The task of such a component is to dynamically extend request of the client with the object to be modified or accessed, thereby reducing the number of tasks for the given controller.

B. API

Hypertext Transfer Protocol (HTTP) requests sent by the client are served by a RESTful API server. The benefits resulting from this communication mechanism include loose coupling and scalability.

Incoming requests are received by the routes layer, which forwards them to the decorators' and then to the controller layer. In terms of the REST architecture, each resource can be accessed through a unique URI. For every important collection stored on the backend, there is an API endpoint, which processes the incoming and outgoing requests with JavaScript Object Notation (JSON) format. To work with these API endpoints, authentication is necessary.

C. Security

To access the Atlas central API, a Google account sign-in is required, a process consisting of several steps as illustrated by Fig. 3. Clients have to send their Google token received from a Google authentication API to the server in order to validate this token. For this process, the server uses the *OAuth2* [13] authorization framework. In case the validation of the Google authentication token is successful, the server generates a JSON Web Token (JWT) which authenticates the requests

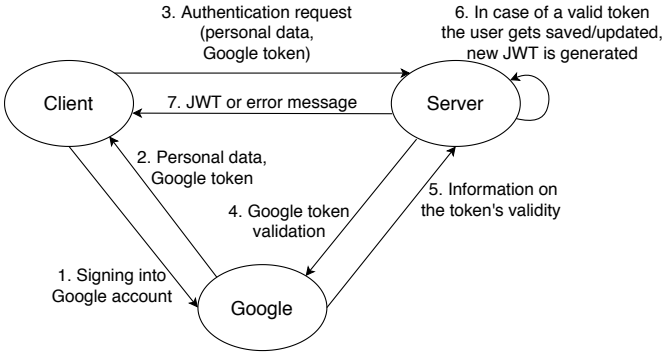

Fig. 3. The Google sign-in process consists of several steps in the background, just like in case of other third-party authentication service providers.

of the client-side user. For a user that already exists in the database, the server updates its data provided by the Google Auth API, otherwise, it gives them a unique username and sets their role to basic *user* by default.

After authentication, the API endpoints become available to the client if they have permission to access them. For each request received, the server uses a middleware in order to verify the JWT token and to check the authorization. Basic users are allowed to access data about the parking spots and cameras and also to manage their personal data which includes their reservations as well. The administrator has every access to all of the API endpoints.

V. The Mobile Client

The main goal of the mobile client is to communicate with the application server and display the user interface on Android and iOS devices. Similarly to the architecture of the server, the mobile client is also made up of several well-separated layers. Communication with the server and other business logic related tasks are handled by the *service* layer. The data received from the server is taken over by the *stores*, which are responsible for storing the information. The *screens* layer contains the views while using the common elements defined in the *components* layer.

The current chapter tackles the presentation of the communication with the server, the business logic layer, the display layer, and the user interface.

A. Communication with the server

The mobile application communicates with the central server via REST requests. The *axios* dependency is responsible for assembling the header and body of requests sent to the REST API.

Using the Axios library, communication over the network works asynchronously [14], so each request sent by the *service* layer returns a Promise, which makes it easy to handle and process asynchronous responses.

After logging in, the personal token returned from the server is placed in the default header of the Axios instance, thus it is included in every request. The server checks the validity of the token and then responds to the request depending on the

role of the user. In case of a logout, the contents of the header get deleted.

B. Business logic layer

In the case of the mobile application, the business logic is represented by the *services* layer. The services contain functions that implement requests to the server or perform calculations based on data already retrieved. The business logic is separated from the display layer as only service classes initiate requests to the server. Stores use an instance of these classes to retrieve the required data from the server.

The services achieve the actual Google authentication, while they are also accountable for preparing the parameter lists of certain requests and for checking the responses received from the server. Provided that the answers are alright, the received data gets stored in the appropriate store. Otherwise, an Error object is thrown by the method, which is then caught and handled properly by the upper layers.

C. Display layer

The display layer consists of three parts: *screens*, *components*, and *stores*. The screen layer contains the views that the user can navigate through. These views use different components, items that can be treated as a separate unit. The elements required for navigation that are permanently present on the screen, such as the bottom navigation bar, are also part of the components layer. For screens and components the data is provided by stores, which use service classes to communicate with the server.

React Native, the framework used by the mobile application provides several options to navigate between the screens with the *react-navigation* library. The Atlas mobile application's navigation is built upon a *Switch Navigator* which connects the *Login* screen, the *Welcome* screen and the logged-in parts so that they are only accessible in the specified order.

The map view, the primary screen of the application, provides the visualization of the available parking spots. For a consistent look and feel, both Android and iOS devices use the map provided by Google. Region changes inside the map view are automatically handled by a built-in function of the map, thus helping the app to provide a pleasing user experience. To increase efficiency, the map always retrieves from the server only the parking spaces located within the currently displayed region, and their positions are marked with custom markers. In order to cluster the markers shown on the map view that are covering each other, a dependency is used. which extends the basic React Native map. Additionally, the possibility offered by Google Maps to mark the congestion of the roads and the current traffic is also enabled.

The list view provides more detailed information about the parking spaces, including the user's distance from them. The distance calculated by the server is updated in five seconds interval.

Both the map and list screens use a modal that comes up from the bottom of the screen to provide detailed information about a specific parking space. Another common element of the two screens is the header. This component includes a hamburger icon which on press opens the side menu, and an icon to display the search header. The search is performed after each entered character in the text field and the filtering is based on the names and addresses of the parking spaces stored in the database.

The data used by the application are kept in stores. In these classes, the view-related data are marked with an `@observable` annotation, while the functions that can modify the values of the data are also flagged (`@action`) so that they notify the interested views about the change. Parts of the screen that contain store data and are marked with `@observer` annotation, are rerendered each time when the data is changed.

Since *MobX* is the state manager of the application, there are three central stores defined: *authentication store*, *map store*, and *parking spot store*. The authentication store manages the data of the logged-in user and it also establishes a connection with the authentication service. The map store is closely related to the map view, as it stores the region currently displayed by the map and the location of the user. The last one contains the parking spaces retrieved from the server.

D. User Interface

Since the React Native it is a component-based library, the views on the user interface are composed by different elements. The entire interface consists of React Native components and their extension classes, and the framework ensures that the appropriate parts of the interface get quickly updated as data changes. In addition to built-in elements, such as buttons or text boxes, own components can be created as well. Components' size, appearance, and position can be customized with style sheets.

Following the UI/UX principles [15], a key aspect of the project is to build a user-friendly application that is easy to use and shows consistency with frequently used products. This makes it easier for the user to learn about the new product without any additional learning cost. Keeping in mind the platform specific elements, the Native Base UI library is used, which provides components similar to the usual Android or iOS look and feel elements. The interface of the application is easy to use in order not to distract the driver's attention from the roadway, to require as little interaction as possible and to help with navigation. Following the *less is more* design principle, clarity, usability, and consistency are also important factors, thereby reducing the users' cognitive costs.

VI. Technologies

The central server uses Node.js [16], chosen for its efficiency in serving scalable web applications and also for being able to uniformly use the same programming language on the server-side as in case of the mobile client. As JavaScript is a loosely typed language, it can conveniently handle Not Only SQL (NoSQL) databases, such as the one chosen in case of the Atlas project, namely MongoDB. For the creation of the RESTful API, the minimalist and flexible Express.js

framework is implemented, paired with Axios, a promise-based HTTP client. The use of MongoDB is justified by storing geolocation data which is supported by default. The bridge between Node.js and MongoDB is formed by the Mongoose ODM [17] which solves the automatic mapping between database entries and local objects. For being able to calculate distances between parking spaces and the times required to cover the distances the Open Source Routing Machine (OSRM) [18] is put into use, which represents an important part of the architecture.

The mobile application uses React Native [19] as a cross-platform framework, which enables the development of complex, interactive user interfaces for both Android and iOS devices. MobX [20] is responsible for state management, chosen due to its simplicity and efficiency. The Native Base library borrows a native look on both iOS and Android devices, thus users are greeted by a pleasant and familiar user interface. Firebase [21], the mobile and web application development platform by Google, is used to solve the authentication of users with their Google account.

The GitLab repository management system is utilized for version control, project management and continuous integration and delivery. For deploying the application, Docker [22] is used for its efficient virtualization capabilities. Alongside Docker Compose [23] is used to connect the application server, the MongoDB database, the module that seeds the database with demo data, and the OSRM server. Quality assurance is achieved using ESLint as static code analysis tool, paired with Prettier plugins for code formatting.

VII. The Usage of the Mobile Application

When the mobile application starts, the sign-in page welcomes the user with a Google login button. Touching the button brings up a window where the user can select a Google account to use. Signing in for the first time, the app asks for permission to use the name, email address, and profile picture belonging to the account. By accepting these the login is successful and the page greets the user with a personalized message. An unsuccessful login may be caused by the lack of internet connection or server-side problems, in which case an error message appears on the screen.

To access certain features of the application, the application asks permission to geographical location of the device. To do this, a pop-up window appears for the user after logging in to grant access to their position. In case the user accepts it, the pop-up no longer appears at later logins. If the Global Positioning System (GPS) is not turned off for the device, the app will suggest to turn it on using a pop-up window again.

The logged-in user is navigated to the map view (Fig. 4.), which is the primary form of displaying parking spaces, based on their position. On the map, the roads are colored according to the current traffic, which helps the user to decide which route can be taken. In order to reduce the number of markers appearing on the screen, the parking spaces are clustered on the map. The clustering of the markers are processed dynamically in case of user interaction by zooming in and out

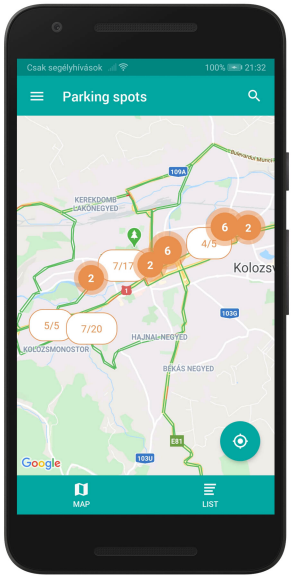

Fig. 4. The main screen of the application which displays available parking spots with the help of a map view.

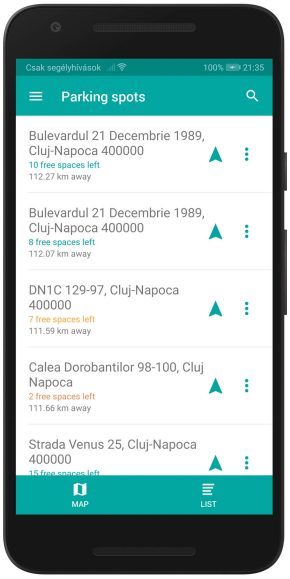

Fig. 5. The parking spots can also be viewed as a list with address, available spots and distance information.

the map. When touching a clustered marker, the map zooms in automatically. The non clustered markers show information about the total capacity of the parking lots and about how many of the spots are available. Navigating to the position of the user is possible by pressing the right bottom button with the current location icon.

The bottom navigation bar of the screen allows the user to navigate between the map and list views. More detailed information about the parking spaces in the region on the map can be viewed on the list view screen (Fig. 5.). The list items include the name of the given parking space, the number of free spots and the distance of the parking lot from the position of the user. The number of available spaces is colored depending on how many of them are left: dark red color indicates a number below four, orange marks less then eight unoccupied spots, otherwise turquoise color is used.

All information about a selected spot can be displayed by pressing on an item. In this way a modal appears as in the map view screen. This window contains the name and exact address of the given parking space, the parking fee, the rating of the spot, a small map of the exact position and two buttons.

It is possible to search between parking spots by pressing the search icon on the right side of the header and entering characters in the text field. Thereby a filtered result list appears which is updated at every change of the text field's value.

By pressing the top-left hamburger icon, a side navigation bar appears. In the sidebar further navigation items are available and also the logout can be carried out here. The other sidebar items are considered further development possibilities.

VIII. Conclusions and Further Development

Within the Atlas project, a software system has been created for simplifying the process of searching for parking spaces.

The central server of the system connects the other components and provides an API to its clients. Using this API, the mobile application displays the available parking spaces on a map or in a list view, as well as it offers the possibility to reserve parking spaces.

During the development of the software system, several further development possibilities have arisen.

One of the main priorities for future development would be the navigation to the selected parking spot by opening an external map application such as the Google Maps route planner. Furthermore, it would be useful for the application to send a notification to the user if the selected parking spot is no longer available.

Installing the currently inactive buttons in the side menu (*Profile*, *Favorite places*, *Settings*) would also be an important feature. A new screen could be created where the user can view and edit their personal information. Even vehicle data could be entered here, allowing the app to recommend parking spaces depending on the size of the vehicle and the size of the available space. It would be also possible to save favorite, frequently visited locations, allowing the user to search and book a parking space near the saved location.

Creating a web client would also be a useful extension, which would help administrators to maintain the surveillance cameras associated with the system and the locations they monitor. Moreover, further development plans include the possibility of logging in with Facebook, as well as the creation of an own registration system.

The implementation of a feedback form would be a useful addition as well, allowing users to report if the number of available parking spots indicated by the application does not correspond to reality, either due to the computer vision module's wrong prediction or a camera failure. It would also be desirable to mark and display parking spaces suitable for the disabled separately, as well as to install the daytime and nighttime theme of the app so that drivers would not be disturbed by the light emitted by the application in the dark.

REFERENCES

[1] I. P. Institute, "2012 emerging trends in parking," *URL https://www.parking.org/wp-content/uploads/2015/12/Emerging-Trends-2012.pdf*, 2012.

[2] Siemens, "Case studies for traffic solutions," *URL https://www.academia.edu/5300681/Case_studies_for_traffic_solutions_Modern_concepts_and_technologies_help_improve_efficiency*, 2011.

[3] S. Nordbruch, "Controlling a parking lot sensor," Dec. 26 2017. US Patent 9,852,623.

[4] A. Kianpisheh, N. Mustaffa, P. Limtrairut, and P. Keikhosrokiani, "Smart parking system (sps) architecture using ultrasonic detector," 2012.

[5] J. K. Suhr and H. G. Jung, "Automatic parking space detection and tracking for underground and indoor environments," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 9, pp. 5687–5698, 2016.

[6] J. K. Suhr, H. G. Jung, K. Bae, and J. Kim, "Automatic free parking space detection by using motion stereo-based 3d reconstruction," *Machine Vision and Applications*, 2010.

[7] E. Boc, "Green mobility and quality of life: Cluj-napoca case study," *Sustainable Development and Resilience of Local Communities and Public Sector Organizations*, p. 79, 2018.

[8] "Cluj-napoca: Aplicația yeparking e un park-sharing. poate pune capăt ocupării abuzive a parcărilor plătite?," 2018.

[9] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1725–1732, 2014.

[10] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*, vol. 7. University of California, Irvine Irvine, 2000.

[11] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[13] R. Yang, W. C. Lau, and S. Shi, "Breaking and fixing mobile app authentication with oauth2. 0-based protocols," in *International Conference on Applied Cryptography and Network Security*, pp. 313–335, Springer, 2017.

[14] A. Mardan, *Asynchronous Code in Node*, pp. 417–429. Berkeley, CA: Apress, 2018.

[15] J. Gothelf, *Lean UX: Applying lean principles to improve user experience*. " O'Reilly Media, Inc.", 2013.

[16] P. Teixeira, *Professional Node. js: Building Javascript based scalable software*. John Wiley & Sons, 2012.

[17] S. Holmes, *Mongoose for Application Development*. Packt Publishing Ltd, 2013.

[18] S. Huber and C. Rust, "Calculate travel time and distance with openstreetmap data using the open source routing machine (osrm)," *The Stata Journal*, vol. 16, no. 2, pp. 416–423, 2016.

[19] B. Eisenman, *Learning react native: Building native mobile apps with JavaScript*. " O'Reilly Media, Inc.", 2015.

[20] P. Podila and M. Weststrate, *MobX Quick Start Guide: Supercharge the client state in your React apps with MobX*. Packt Publishing Ltd, 2018.

[21] L. Moroney, Moroney, and Anglin, *Definitive Guide to Firebase*. Springer, 2017.

[22] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[23] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.