Cloud-based Serverless Solution for Facilitating the Organisation of Athletics Competitions

Tifani Franciska Nagy
Babes-Bolyai University
Cluj-Napoca, Romania
nagytf@gmail.com

Zsolt Csibi
Babes-Bolyai University
Cluj-Napoca, Romania
zsolcsib@gmail.com

Borbála Jánosi
Babes-Bolyai University
Cluj-Napoca, Romania
janosi.borbala@gmail.com

Károly Simon
Babes-Bolyai University
Cluj-Napoca, Romania
simon.karoly@codespring.ro

Hunor Hegedüs
Codespring
Cluj-Napoca, Romania
hegedus.hunor@codespring.ro

Erika Szász
Codespring
Cluj-Napoca, Romania
szasz.erika@codespring.ro

***Abstract*—The Athletimeter project was inspired by the annually held Béla Török Memorial athletics competition in Odorheiu Secuiesc, Romania, which brings together children from the surrounding areas to compete. So far, the organizers have not used any digital solution to maintain the data related to the competition. It was managed on paper, which did not provide the necessary transparency and secure maintenance for the organizers, and there was no opportunity for the spectators to follow the results.**

Therefore, the authors' aim was to develop a cloud-based system that could facilitate the data management needed to run an athletics competition by digitizing the process. The system provides a web and a mobile interface to its users, and is based on a cloud-based server that introduces the concept of serverless architecture. The server is only accessible during the competition, so the amount spent on resources is significantly reduced.

Through the web interface, organizers can manage age groups, contestants, events, and results. In addition, the platform allows spectators to track the results of the different events in real-time.

The mobile application provides an efficient alternative for recording results and identifying contestants. It is equipped with a built-in stopwatch to record the results of time-based events and a QR code reader that can associate a competitor with a result by scanning the code on the jersey.

The paper aims to present the system, detailing its functionalities, architecture, and different components, as well as the technologies and tools used.

I. Introduction

The Béla Török Memorial athletics competition is held annually in Odorheiu Secuiesc, Romania, where children from different localities compete against each other. Competitors take part in different events according to age group. The athlete's running time or distance in jumping or throwing is measured by the assigned organizer and recorded on a piece of paper. It is then the participant's responsibility to go to the appropriate desk and show the organizer, who summarizes the results on paper. After all the events have taken place, the completed tables are distributed to the representatives responsible for the children. This method makes it difficult for the organizers to manage and archive the data and for spectators to keep track of the results, which can lead to a significant number of human errors during the organization.

The Athletimeter software system was inspired by this competition and aims to facilitate the work of the organizers by digitally processing the data. The application can be divided into three components: a web interface for displaying and managing data, a mobile application for quick recording of results, and a cloud-based server that is available for the duration of the competition and is responsible for serving data to these platforms.

The organizers can manage the data via the web application. For the long-term preservation of the information, the platform provides the possibility to save the records in different formats, which can be later transmitted by the organizers to generate different statistics. As the server is deployed before the competition, organizers can quickly add data from previous competitions by uploading datasets in the proper format. The web application allows spectators to follow the results in real time, which can even be projected at the place of competition.

The mobile application contributes to the efficient recording of the results. This is equipped with a QR code reader, allowing organizers to identify the competitor by scanning the code on the clothing. A built-in stopwatch can be used for timed events. After the timer was stopped and the contestant was identified, the result is automatically saved.

Similar software systems already exist on the market to facilitate the organisation of athletics competitions, such as *Orgsu*[1] or *Athletics.app*[2], which in addition to data management, allow the advertising of different sporting events and the registration of athletes. Contrary to these solutions, the Athletimeter project is purely an administrative tool, which aims to facilitate the work of organizers, and thus it is sufficient for the software to be available only during the competition. The serverless technology greatly contributes to

[1]https://www.orgsu.com (accessed Aug. 3, 2022)
[2]https://www.athletics.app (accessed Aug. 3, 2022)

cost efficiency and allows the quick and easy automated deployment and removal of the resources, while the export to different formats ensures data preservation. The mentioned softwares also include a mobile application, for recording results in the case of *Athletics.app* and timing in the case of *Orgsu*. On the other hand, the Athletimeter application combines the mentioned functionalities with the possibility of quick identification of the athletes.

The paper is divided into 5 main sections: Section II. describes the user roles defined within the application and the associated functionalities. Section III. provides an overview of the communication model and discusses the different components of the architecture. Section IV. describes the technologies and tools used, and the last section draws conclusions and lists some possibilities for further development.

II. Roles and functionalities

The Athletimeter software system provides two user interfaces: a web application and a mobile application.

The features of the web application can be separated according to three user roles: guest users, organizers, and administrators. In the case of the mobile application, these are only available to registered users who act as organizers.

This section presents the main functionalities of the system based on the previously mentioned roles.

1) **Guest user**: The users who are not registered or logged in, are considered to be guests. Guest users are not able to use the mobile application, and the accessibility is limited in the web application as well, as only the homepage and the information page can be accessed. On the latter, guests can receive real-time information on the progress of the competition.
2) **Organizer**: The organizers are registered to the system by an administrator. After login, organizers can manage events, contestants, results, and age groups on the web application. On the pages dedicated to adding objects, besides filling out a form, the data can also be imported from previously saved JSON files. In addition, the saved contestants, events, and results can be exported for archival purposes to JSON and PDF format. The profile page is also accessible on the web application, where organizers can view and update profile information.
   On the mobile interface, the organizers' toolbox is extended with a built-in stopwatch as an alternative to adding results in timed events, and a built-in QR code reader for contestant identification.
   Multilingual support is available for both clients, currently supporting English and Hungarian.
3) **Administrator**: Besides the functionalities mentioned above, the administrator has also access to a web interface for managing the organizers of the competition.

III. Architecture

The Athletimeter software system provides the digital management of an athletics competition's data via a website and a mobile application connected to a cloud-based server.

The web client provides an easy-to-use platform for managing and viewing the data provided by the server (see Fig. 1). The web page is implemented using the **React.js** component-based JavaScript technology.

The mobile application developed in **React Native** communicates with the server and hosts a user interface for the organizers to provide easy data entry (see Fig. 4 and 5).

The server composed of five **Azure function applications** implemented in **.NET Core** is connected to the **Azure SQL** cloud database using services provided by the **Entity Framework Core**. The management of the data types defined in the system and the login are handled by separate function applications, which operate independently and receive requests through different URLs. Each function application provides a RESTful API to communicate with the clients.

The real-time delivery of new results to the web interface is implemented using the **Pusher Channels API**. When a result creation, deletion, or update operation is performed, the function application of the results publishes the information to the channel as a properly named event. If the web application is connected to the same channel, it can receive these events without performing a data request for synchronization.

Figure 2. shows the communication and architectural components of the software system, which are further detailed in the following subsections.

A. Server

1) Structure: The server can be divided into two main parts. The **application** part contains the function applications, as well as a **Main** module, which provides the data access layer and the classes implementing business logic and other data configurations. These are broken down into separate submodules, between which the dependency relationship is shown in Figure 3. The **configuration** part contains pre-written scripts for database creation and configuration.

Within the software system, five separate entities are defined: contestant, organizer, result, event, and age group. In the **Functions** component, there is a separate Azure function application responsible for the first four entities and the login, which contains the operations that manage the data for that entity and the authentication. There is no separate function application defined for age group management, as this can be performed through the functions related to organizers, events, and contestants. Each of the function applications runs parallel on separate ports and executes the tasks independently.

The **Models** module encompasses the representation of each entity and a factory class that configures the database and returns the objects representing the database tables. To increase security and minimize the amount of data to be sent, the **DTOs** module contains DTO (Data Transfer Object) classes defining different representations of the models. The conversion from model to DTO, and vice versa, is done by a mapper. The classes of the **Repository** layer follow the DAO design pattern and are responsible for implementing the CRUD (Create, Read, Update, Delete) operations related to the entities, using abstraction at two levels, thus increasing generality and

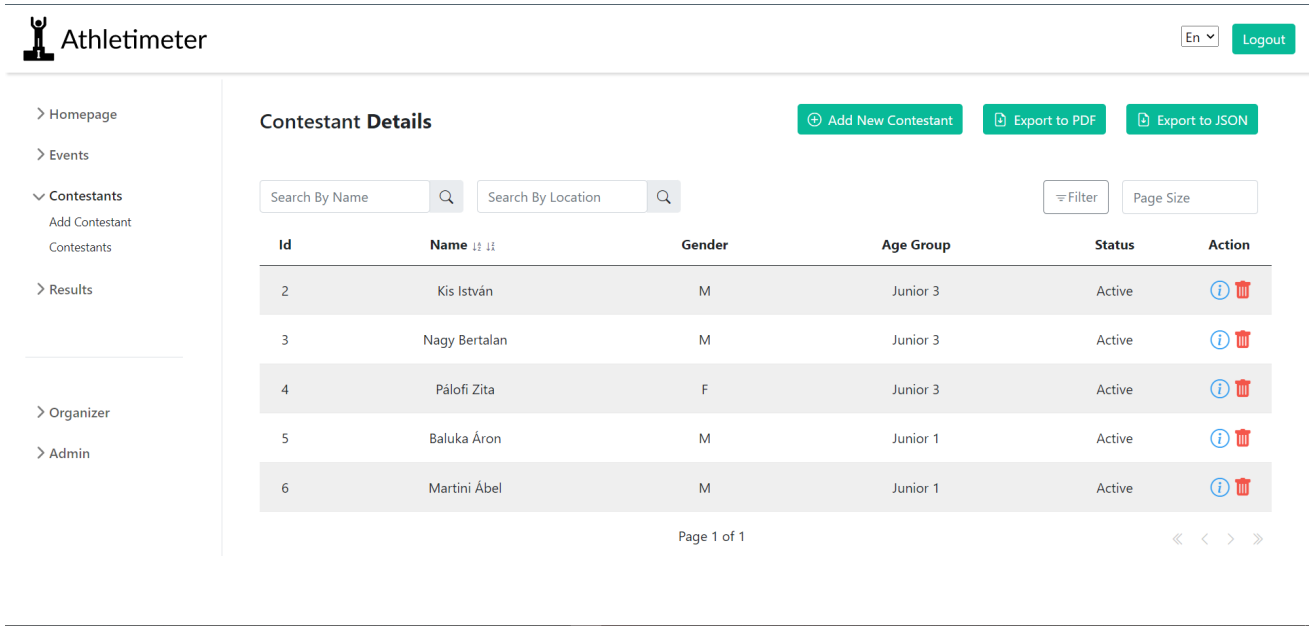

Fig. 1. The contestant listing page on the web application interface.

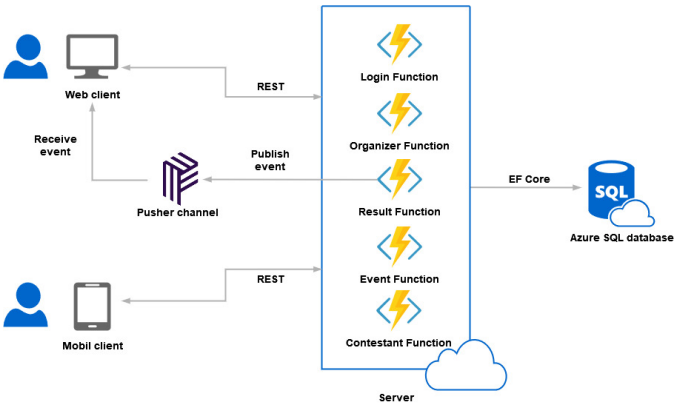

Fig. 2. The architecture of the Athletimeter software system.

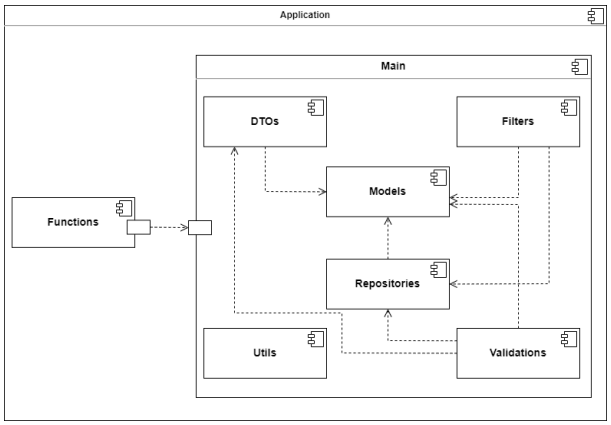

Fig. 3. The structure of the backend part of the application, the relationships between the components.

reusability. The retrieved information can be sorted and filtered according to different criteria. The **Filters** module contains separate filter classes for each entity. The **Validations** layer's classes are responsible for validating the objects received from the client platforms. The **Utils** component consists of not entity-specific functions, such as password hashing, JWT token generation, Pusher configuration, etc.

2) Security: As mentioned previously, the HTTP requests received from the client applications are served by the API server corresponding to the entity. The login operation is handled by a separate function application, which after receiving the authentication data, generates a **JSON Web Token (JWT)** for the user. The token includes the username, id, and role in the form of encrypted key-value pairs and is stored on the client side. For each request, the HTTP header will contain the generated token. If the user does not have permission to access a functionality, a proper HTTP status and message are sent according to the REST API conventions.

3) Serverless technology: As different types of data are typically handled during different phases of an athletics competition, the separation of the Azure function applications by entities provides a cost-effective solution, with no unnecessary resource reservation, as a function application is only active when it receives requests. In addition, efficiency is greatly enhanced by the rapid deployment of the system before the competition, as functions can be easily deployed and no other maintenance operations are required, as the resources are managed and scaled on demand by the cloud provider.

The REST API interfaces implemented by function applications are based on the use of *HTTP triggers*, which trigger the execution of functions when HTTP requests are received. Functions within an application have a unique name specified

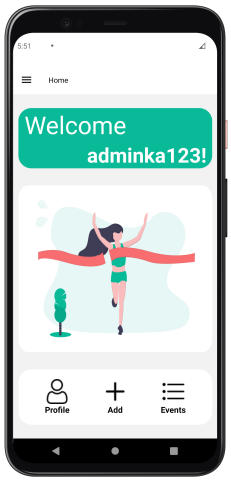

Fig. 4. (a) Homepage

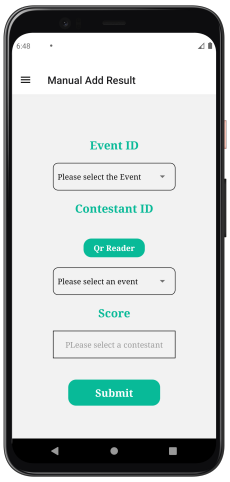

Fig. 5. (b) Add result page

as an attribute parameter of the *FunctionName* method. In addition, the parameter list also includes an *HttpTrigger* attribute, which indicates the type of triggering event and specifies the authentication level required to run the functions, the list of HTTP methods, and the route template that controls URLs. The data available during execution is represented by the functions' input parameters: the *HttpRequest* type parameter provides access to information related to the request, while the *ILogger* object is used to log operations. The output parameter is an object of type *IActionResult*, which encapsulates HTTP responses with different status codes and contents.

The deployed function applications are located within a shared *resource group*. Each is associated with an *app service plan*, which defines the needed computational resources, a *storage account* used by the provider to store temporary objects created during the execution, and an *application insights* service, which contributes to the monitoring of the functions.

B. Web

1) Structure: The structure of the website can be divided into several components. The **components** layer includes the page header, the navigation menu, and other elements that are used by the components defined in the **pages**. The **pages** module contains the main container for the frontend. The views displayed by the main are in separate submodules consisting of pages responsible for CRUD operations, and the information page. The **service** library contains *service* methods responsible for the communication with the server. The **config** package contains predefined constants and paths used by the components and the fetch methods, as well as the files implementing the internationalization. The **assets** module contains the style settings for the web interface, and a separate **icons** module contains the images and icons of the platform.

2) Architecture: Based on the communication model described at the beginning of this section, the website communicates with the server via RESTful APIs. While the server is only responsible for data handling, the clients update the page based on the information received and the events triggered on the user interface. Thus the application has a **rich client** architecture since the operations of the display and control components are defined based on the **MVC (Model-View-Controller)** pattern and implemented on the browser side.

The web client's *MainContainer* component displays the corresponding **pages** elements based on the option selected on the sidebar. Thus, the web client consists of a single page, and it can be considered a **Single-Page Application (SPA)**. As Figure 6. shows, there is no full page reload, the browser requests the HTML file that provides the single page of the application, as well as the associated static files and images, only on the first load, then it updates only those components of the page where it detects a change.

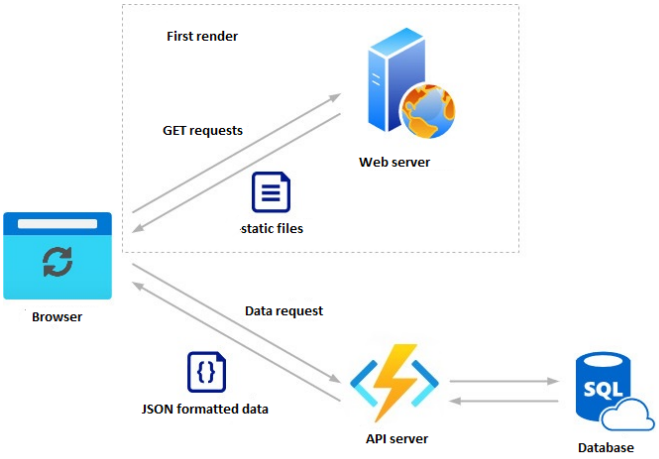

Fig. 6. Illustration of SPA.

3) Communication with the server: The communication between the client and server is implemented using **Fetch API**, which allows the resources to be fetched asynchronously. The API provides the generic *Request*, *Response* objects and the global *fetch* method to access the server-side REST API. The first parameter of the fetch method is a URL and the second is a JSON object containing the request header, type, and request body. In response, a JSON object is returned.

4) Components: The elements of the **components** and **pages** module are implemented in *.jsx* files. This extension provides the possibility to include both HTML and JavaScript code in the same file to describe a React component. However, browsers are not able to process this format, so React uses a compiler called **Babel** to convert *.jsx* files into JavaScript code. In the background, the compiler replaces the HTML code by passing the information defining the element as a parameter to the *React.createElement* function.

Elements are defined as **function components**, which are JavaScript functions that return React elements. Contrary to class components, functions do not have a *state* object whose modification causes the component to re-render. In this case, the state management is implemented using hooks.

5) Live update: The channel provided by the **Pusher Channels API** allows the website to receive real-time noti-

fications related to the results. The add, update and delete operations are indicated by the function application related to results in the form of named events. The client-side retrieves the new information via the *bind* method of the channel, whose parameters are the event name and a *callback* function, within which the update of the list based on the event is executed.

When the list of the results is updated, a reordering animation is displayed on the interface, whose implementation is based on the **FLIP** (First-Last-Invert-Play) list reordering animation principle. The first step consists of determining the current position of the result card components. After receiving new information from the channel and updating the list correspondingly, the positions of the card components may change, thus in the second step, the new coordinates are determined using the *getBoundingClientRect* method provided by JavaScript. After this, to ensure the animation, the inverse of the calculated offsets is applied, so that the browser is prevented from automatically changing position and the elements appear to remain in the original place. The function *requestAnimationFrame* is executed immediately before the redraw, and in the last step, the method given as a parameter to it sets an appropriate transition, which affects the motion of the position change on the interface.

6) Security: After successful authentication, the website receives the **JWT token** generated by the server, and stores it in a *browser cookie*, which is set and retrieved by the hooks provided by the *react-cookie* npm package. The hooks also allow the specification of keys used to defend against security vulnerabilities. The web client provides the *SameSite* key with the value *'Strict'*, which protects against CSRF attacks by preventing the transmission of a cookie in case of a request from another site. Furthermore, the *Secure* key is set to true so that the cookie is only sent if the request is directed to an HTTPS page. The application updates the options of the sidebar menu based on the token value and the role of the user. When a page is accessed, the system checks the content of the browser cookie, and if the user does not have the proper permissions, the application displays the login page instead.

C. Mobile

1) Structure: The layered structure of the mobile application is very similar to that of the web application. The function components in the **components** folder contain the pages to be displayed. Similar to the web application, the **services** module communicates with the server through the *fetch* method provided by the **Fetch API**. The constants in the **config** folder are used to specify the IP address to access the Azure functions stored in the cloud. Internationalization is implemented in a separate file, and each language has a dedicated *json* file in the *locales* subfolder, which contains the application's textual content in the respective language.

2) Navigation: Navigation between pages is provided by the **React Navigation** community package. In the App.js file, the application code is wrapped in a *NavigationContainer* component. The initial path is configured as a property of a *Drawer.Navigator* component and all the other required paths are properties of different *Drawer.Screen* elements. The pages visited are placed in a stack, which initially contains only the path to the default one. Navigating through pages, after each call to *navigation.navigate('route')*, the new path is added to the top of the stack. This makes it easy to implement navigation for back operations.

3) Security: Similar to the method mentioned in the section III-B6., the application sends the authentication data to the server, and receives an object containing a generated JWT token. However, contrary to web authentication, on the mobile side, the token is stored in the *SecureStore*.

IV. Tools and technologies

A. Server technologies and tools

The application server is developed using **ASP.NET Core** [1] which is a fast, cross-platform, secure framework that provides the possibility to develop a reliable cloud-based server. Another advantage of .NET Core is the seamless integration with Microsoft products. The code is written in C# with the help of *Visual Studio Code*. The **NuGet** package manager is entrusted to manage the .Net Core package dependencies in the project.

The server is based on **Azure Functions**, which is a serverless computing service offered by the **Microsoft Azure platform** that allows event-driven execution of application code, providing the required infrastructure. The necessary resources and all changes, settings, and incoming requests, can be viewed and easily managed on the **Azure Portal**.

The system's **Azure SQL** database and **Azure Functions** [2] communicate using **Entity Framework Core**, a lightweight and open-source object-relational mapping (ORM) system, which allows mapping an object model to a relational schema. The data is stored in records in the database, and each record is assigned to a .NET Core object, through which the database operations are performed.

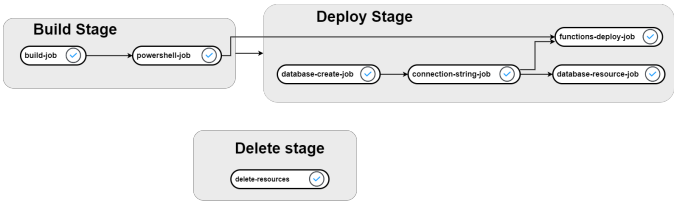

Fig. 7. Diagram of how the application server deploys and deletes scripts work.

The implementation of the PDF export is based on the **DinkToPdf** .NET package which is assigned to the project using the NuGet package manager and allows the conversion of HTML files to PDF format.

A **Gitlab CI pipeline** is used to automate the deployment of the database and the Azure function applications and create the required resources on the Azure platform. These operations are triggered by a *push* operation on a specific Gitlab *branch*, and performed in the order shown in Figure IV-A. As the

application must be only accessible during the competition, the deletion of the resources is also automated.

The pipeline can be divided into different stages consisting of jobs. The *build-job* within the **build** section is responsible for collecting the files and dependencies of the function applications. The *powershell-job* creates different .zip folders from these files for each function application separately. The second stage is the **deploy** stage, which is responsible for generating the resources on the Azure cloud provider side. The *database-create-job* creates the database resource group and the SQL server. The *connection-string-job* retrieves and completes the database connection link with the username and password of the administrator who is authorized to access the server. The *functions-deploy-job* job creates Azure function applications and the resources, uploads the zip files mentioned above, and sets up the configurations of the function applications. The *database-resource-job* runs the .Net project that creates the database tables and inserts the default age groups and the admin user. The **delete** stage is responsible for deleting resources on the Azure cloud provider side.

B. Web technologies and tools

The website is implemented using the **React.js** [4] JavaScript library developed by Meta Platforms for building web applications. The technology allows the efficient development of performant web applications by only reloading the components that have changed, thus avoiding unnecessary reloading of static components. It is based on the use of reusable elements, within which the principle of rendering logic is coupled with other user interface logic. The web application is developed in *Visual Studio Code* and **Node Package Manager (npm)** is used for the structural development and the management of the project dependencies. To facilitate the look and feel of the website, the **Bootstrap** open-source CSS framework and the *react-bootstrap* npm package is used.

The real-time update of the results on the web page is implemented using the **Pusher Channels API** [3]. The service forms a real-time communication layer in the form of channels, through which the server can signal to the client application in the form of system events.

The web application is deployed to the **Heroku** [5] container-based cloud platform, which allows the easy and free deployment and management of applications by providing the necessary infrastructure and software component maintenance. Similar to the server, the deployment is automatically executed within the deploy *job* of a *pipeline*. Whenever a *push* operation is applied to the *master* the remote repository of the service is synchronized with the current state of the project.

C. Mobile technologies and tools

The mobile application is also implemented in *Visual Studio Code* using the **React Native** [6] cross-platform JavaScript framework, which allows the simultaneous development of Android and iOS platform. The application is developed similarly to the webpage, as React Native is also a component-based technology, and supports the integration of the React framework within mobile applications by converting the components in the background into platform-specific native ones.

For the development and deployment of the mobile app, the **Expo** platform is used, which is a framework for facilitating the maintenance of React Native applications.

V. Conclusions and further development

The Athletimeter application is a three-component software system that facilitates the digital processing of data needed to manage an athletics competition.

As the application is used only during the contest, its permanent availability is not necessary. Using pre-written scripts, it is possible to automatically start the application server before the competition and quickly remove it after.

The system's web application provides a transparent and user-friendly interface for organizers to maintain competition data. In addition, the platform also provides spectators the possibility to follow the results in real-time.

Through the mobile application, organizers can access services that identify competitors and measure/save the results.

From the perspective of further development, on the server-side, the existing scripts could be extended to customize the resources stored in the cloud, so that multiple competitions could be organized simultaneously.

At present, the web application allows archiving data in different formats, but statistical processing is not ensured. The organizers could be given the possibility to view statistics and charts on a web page, based on the results stored and aggregated according to different criteria.

The addition of multiple views to the information page would offer different types of visualization of the results. The spectators could choose between different display formats of the data, which would make it more enjoyable to follow.

Some features of the mobile application are currently only available on Android, and the deployed version is only available on this platform. Adaptation to the iOS platform is therefore another option for further development.

As the QR code on the jersey may cause problems during competitions, an alternative to the identification of the contestants could be to use an NFC reader. Participants could be equipped with an NFC bracelet, which could be scanned by the organizers using the mobile application. This would also provide the possibility to automate the insertion of results for timed events, as NFC gates with sensors could be placed at the finish line to detect the arrival of the runner.

References

[1] A. Freeman, PRO ASP.NET Core MVC, Apress, Berkeley, CA., 2016.

[2] P. K. Sreeram, Azure Serverless Computing Cookbook, Packt Publishing Ltd, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK ,2020.

[3] J. Lengstorf and P. Leggetter, Realtime Web Apps, Apress, Berkeley, CA., 2013.

[4] D. Bugl, Learn React Hooks, Packt Publishing Ltd, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK ,2019.

[5] N. Middleton, R. Schneeman, Heroku: Up and Running, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2014.

[6] A. Boduch, React and React Native, Packt Publishing Ltd, Livery Place, 35 Livery Street, Birmingham B3 2PB, UK ,2017.