UDC 004.67

O. Tymchenko, T. Likhouzova

AN ARCHITECTURAL SOLUTION FOR DATA COLLECTION AND MONITORING SYSTEM SOFTWARE

Abstract: The article considers the problem of monitoring the condition of vehicles of organizations with a fleet. A comparative analysis of solutions available on the market is done. Requirements for software for data collection and monitoring systems that could be used by both large and small enterprises are formulated. An architectural solution is proposed, which is based on the ideas of service-oriented software architecture and cloud-first approach and allows you to easily scale and increase the functionality of the system, as well as reduce maintenance costs. Developed software for data collection and monitoring of vehicles, which differs from others by the ability to integrate with any vehicle.

Keywords: fleet management system, intelligent transport systems, distributed data processing systems, cloud computing, NoSQL.

Introduction

In the process of operation of vehicles inevitably have to face the need for timely maintenance and repair, control over the efficiency of use, it is often necessary to understand exactly where the vehicle is. The pretty obvious fact is that these operations are routine and typical. It is necessary to regularly check the condition of all systems, monitor the wear and tear of units, and there is no possibility of remote control over the location, fuel consumption, and other indicators. By transferring the responsibility for controlling the polling of various sensors of the vehicle to the hardware-software complex, we will be able to monitor the condition of the car from anywhere in the world in real-time via the Internet. Users of such a system can be both individual car owners and logistics companies with a different number of vehicles in the fleet. Therefore, it is important to strike a balance between functionality and ease of use, as well as provide quality work for customers with any number of vehicles.

Analysis of existing solutions

Fleet Management System (FMS) - a transport management system, usually understood as software that allows you to manage a fleet owned by a company, government organization, individual organization, etc [1]. These tasks can range from purchasing a vehicle to maintaining it and then even to disposing of it.

The main purpose of such software is to accumulate, store, process, monitor, and export information. Sources of information are limited only to the specific implementation, so it can be: databases of public administration, API provided by these authorities, databases of insurance companies, data from internal sources of organizations, such as accounting data organization or telemetry devices installed on the transport.

FMS should usually be able to manage tasks of any vehicle - cars, trucks, buses, trailers, etc. Vehicle management tasks generally include [1-3], but are not limited to:

- inventory of vehicles;
- vehicle maintenance;
- licensing, insurance, taxation;
- cost management and cost analysis;
- vehicle disposal.

Also, almost always such systems include functions of tracker and task manager for drivers. In addition, it is necessary to highlight the large range of tasks that are solved with the help of FMS - the collection of telemetry and monitoring. These tasks will be discussed in more detail in this paper. As a rule, tracking and monitoring tasks fall into several categories:

- telemetry collection;
- tracking and route planning;
- warning and notification of anomalies;
- event and operation log.

Many metrics can be collected, and they may vary for each system, but there is a specific standard that regulates such metrics. It is called Fleet management system interface (FMSI) [3]. This standard was founded in 2002 by six European manufacturers: Daimler AG, MAN AG, Scania, Volvo, IVECO, DAF Trucks. The standard allows the development of software that accepts telemetry regardless of the vehicle manufacturer.

To understand what functionality the existing systems on the market have, it is necessary to compare them (table 1). Fleetio [4], Momentum IoT [5], and Onfleet [6] are the most popular on the market.

From the table we can conclude that all of the reviewed products have advantages in their field, for example, Onfleet has a set of the necessary functionality to manage tasks, drivers, orders, and so on. It makes Onfleet very similar to a CRM system. Meanwhile, Fleetio and Momentum IoT have focused on ease of use and real-time information handling. Momentum IoT has a major advantage over its competitors - it has a tracker to collect data, so the telemetry is more detailed and accurate. Fleetio uses a mobile app to gather information, which has allowed it to develop functionality to file reports and get geolocation at the same time.

Table 1.

Comparison of FMS

	Feature	Fleetio	Momentum IoT	Onfleet
1	Free trial	+	-	+
2	Report generation	+-	-	+
3	Real-time information	+-	+	-

 ISSN 1560-8956

End to the table 1.

	Feature	Fleetio	Momentum IoT	Onfleet
4	Easy to use	+	+	-
5	Affordable price	+	+	-
6	Notifications	+	+	+
7	Geolocation	+-	+	-
8	Task/driver management	-	-	+
9	Device for collecting telemetry	-	+	-

Materials and methods

Information from research about similar systems was used to formulate both functional and non-functional requirements for the systems. During the research functionalities were compared, similarities were found, and requirements were synthesized on this basis. The **following** requirements have been identified as necessary:

- authorization and user authentication;
- ability to receive telemetry;
- ability to display new data in real-time;
- ability to generate reports;
- viewing data in graphs;
- adding and editing new vehicles;
- editing of user data;
- scheduling of services;
- possibility to add any number of vehicles;
- storage of detailed telemetry for up to 30 days, subsequent aggregation of daily data;
- real-time alerting via notifications, emails, and SMS messages;
- ability to view real-time geolocation;
- the possibility of tracing the route of each vehicle;
- ability to integrate with other systems through the OpenAPI specification;
- support for up to 500 vehicles simultaneously.

Non-functional interface requirements and hardware/software requirements include the following:

- availability of a unified API for communication with third-party systems;
- the systems with which the product will be developed should provide an HTTP interface or the ability to integrate through a message broker;
- the ability to be deployed in one of the cloud platforms.

- The description of operational requirements is more extensive because it includes many nuances. Security requirements:
- implementation of authorization and authentication using the OAuth2 protocol;
- all communication channels must be protected by encryption;
- logs must not contain PII (personal identification information).

For reliability requirements, we refer to an IBM article [7], which states that for dedicated systems the availability level should be at least 99%, which means 3.65 days per year is available for technical work.

Let's make a requirement for performance according to existing research. Article [8] states that the optimal speed of downloading a page ranges from 1 s to 3 s. Since it can be a challenge to ensure that data loads in such a short time frame, let's assume that the page load speed should be <=3 s and that the content should load in time frame up to 6 s after the page loads.

For the data storage policy (backups) let's take an interval of 24 hours with backups saved for 14 days.

In the matter of managing checks, all errors should be handled in a way to hide the internal implementation from the end-user.

Results

When the system under development is real and constantly expanding, the way the software components are organized and structured becomes one of the most important aspects determining the viability and cost of the software product.

To develop the right kind of software system architecture it is necessary to determine a specific set of properties, that should be provided by the system. According to existing data [9] and our considerations, let’s formulate a list of criteria:

- efficiency - must ensure the fulfillment of functional requirements under the given conditions, scalability, reliability, etc;
- flexibility of the system - the ease of change, the ability to defer key decisions;
- expandability - the ability to add new functions without modifying the existing ones, the implementation of the open/closed principle, the implementation of the YAGNI [10] principle;
- possibility of increasing the number of people in the development team;
- easy testing of the code;
- reuse of components.

The main goal in software architecture development is to reduce the complexity and intricacy of the system as a whole and each component in particular. The methods for solving this problem are decomposition and decoupling [11].

Figure 1 shows the developed hierarchy diagram of the software modules for monitoring and data collection. From the diagram, we can see that the entire software product was divided into two parts: the backend, and the frontend.

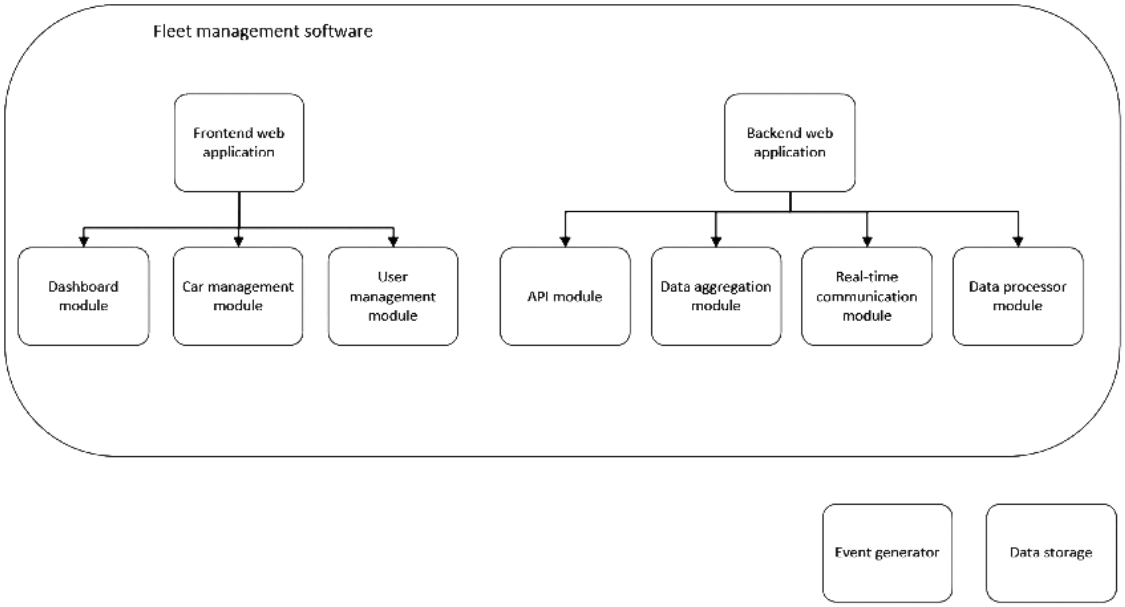

Figure 1. Hierarchy of software modules

The frontend part is generally responsible for the task of presenting information to the user. In the course of decomposition, the functionality was conditionally divided into three parts necessary for MVP: the dashboard module responsible for reports/graphics, the user management module, and the vehicle management module.

The backend is responsible for data reception, processing, aggregation, timely transfer of data to the frontend, and interaction with third-party systems. There is an API module that provides an interface to the application, a module for real-time communication, a module for data aggregation, and a module for data processing. The diagram also shows the data storage module and the event generator separately. They are not part of the software complex but must be present. An event generator is a device for collecting telemetry from a vehicle, but as part of this work, a program simulating such a device was developed to provide a test and demonstration of operation.

The interaction of web applications in real-time is implemented using the WebSockets approach, the principle of which is that a permanent connection is established between the server and the client, and then they exchange data. When the session ends, the connection is broken. WebSockets meets the requirements and is the recommended choice, so this approach was used in the development of the monitoring and data collection system. To facilitate the development process and ensure stable communication, we used a solid off-the-shelf solution, SignalR. SignalR is a library that provides a convenient API that provides a wrapper for working with WebSocket. The advantage of this library is that it supports all kinds of approaches described above. The idea is to use the best option even if you can't connect via WebSocket. Also one of the big advantages is the availability of the library for different platforms, which allows you to choose any technology to implement the software.

Almost any software stores some information or state in some data storage. Usually, databases are used to store this information or state. Considering all the advantages and disadvantages [12-14], it was decided to use the NoSQL database Azure Cosmos DB for this system. This provided the right level of bandwidth because of the ability to handle large amounts of data, made it very easy to change the type of collected telemetry, made it possible to scale and support a large number of vehicles and users. Diagram of the database developed for the software system for monitoring and data collection of vehicles is shown in Fig. 2. Since one of the requirements is to provide scalability, a serverless technology Azure functions, was used to develop the backend. It is an event-oriented platform that allows you to easily and conveniently develop applications that interact with each other in real-time.

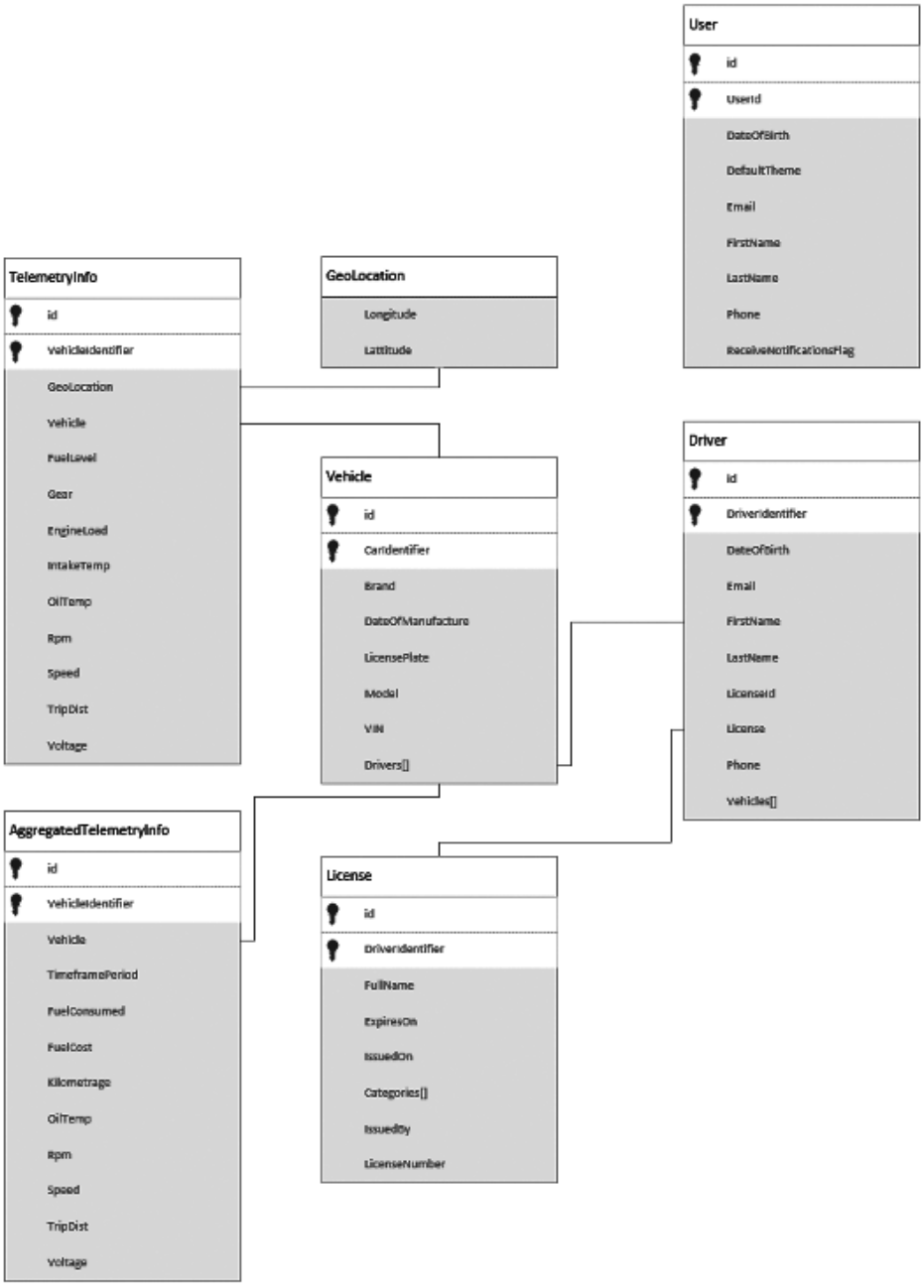

Figure 2. Scheme of the database

 ISSN 1560-8956

The Onion architecture was chosen to organize the code in the backend. This architecture consists of layers, each layer should be as independent of each other as possible, and cohesion grows closer to the center. This architecture has several advantages over the classic three-layer architecture for instance this architecture has better performed principle of control inversion because the different layers interact with each other only through interfaces, and the implementation is determined during operation. The code is easier to test, there are no complex dependencies between layers, there is no need for joint projects.

SPA application was developed as the frontend part of the software. SPA approach allows building a user-friendly and pleasant interface by updating only the components of the page instead of a full update, which reduces the amount of transferred data and makes the data update process almost invisible to the user.

Fig. 3 shows the system modules, separated into individual applications, and the basic infrastructure required for communication and operation.

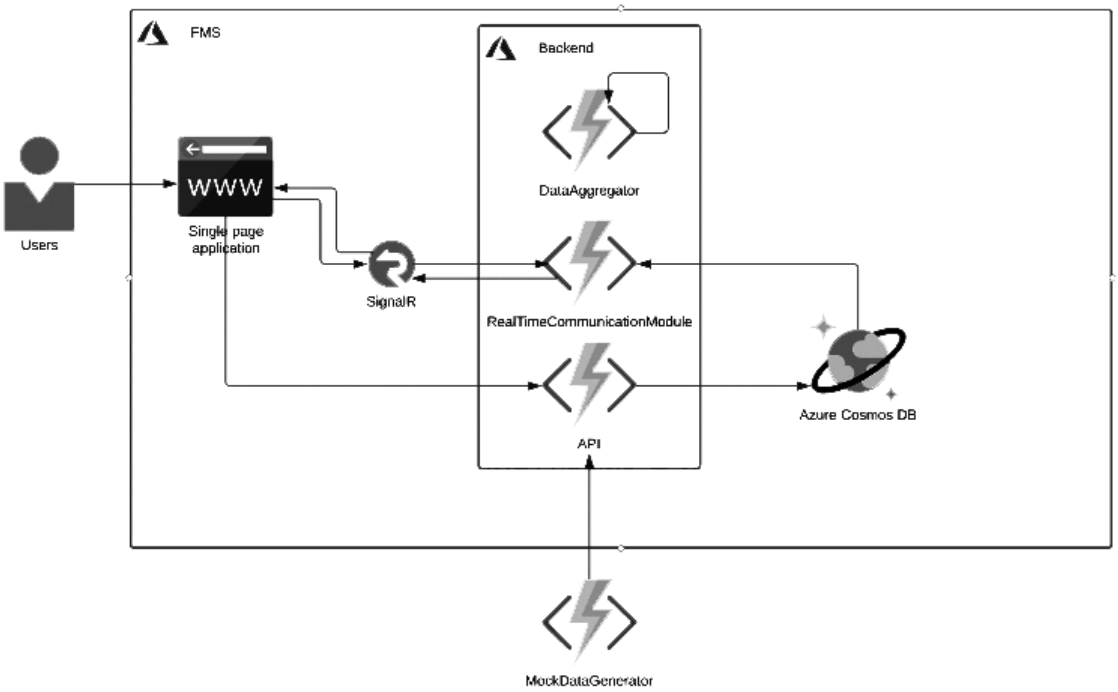

Figure 3. System modules

To solve this problem, it will be advisable to use video cameras, as this method is low-cost compared to others, and also gives fairly accurate results without the need to mark each individual car with labels or additional devices. To implement this method, you can use several approaches:

To properly evaluate the proposed software, it is necessary to formulate the criteria by which it will be evaluated. Today there are many approaches to software quality evaluation, but I believe that the most appropriate and adequate is the ISO/IEC 25010 standard. According to the standard, 8 aspects of software deserve evaluation. Let's evaluate the software for each of these points, on a scale of 0 to 5.

Table 2.

Final quality assessments of the proposed software

Quality aspect	Grade	Notes
Functional suitability	4	limited amount of functionality
Compatibility	5	widely used protocols for communication
Ease of use	3	needs of people with disabilities
Reliability	5	geo-replication, scaling
Security	3	not given due attention during development
Convenience of support	5	onion architecture, frontend structure
Portability	2	extensive use of Azure-specific services

The ability to perform the task as required under different loads was measured to test the effectiveness of the proposed software. In order to assess how efficiently the task was performed, the timeframe was measured from the moment when the request with the new telemetry update was received by the software until the software displays the result in a certain form on the client application.

According to the functional requirements, the MVP implementation must support the simultaneous operation of 500 vehicles. To obtain the result, 5 measurements were performed for 5, 50, 100, 500 telemetry generators. There was a pause after each measurement. The result was calculated as the arithmetic mean for each measurement.

From the results, you can see that the first measurement for each successive test series is, on average, larger than the other tests for a particular test series. Then the measurement results come to a plateau and even decrease for the last measurement. It can be assumed that the delay on the first request is the result of the so-called cold-start problem. This problem is characterized by the fact that applications deployed in the cloud need some time to initialize after a certain amount of downtime. Since there was a long pause between each series of measurements (an average of 25 minutes) it takes some time to "warm up" when receiving the application request.

Overall, the delay for each of the test series is more than acceptable and ensures that the user is comfortable using the software. So, for 5, 10, and 100 generators the average delay has a value of about 1 s, which is more than enough. For 500 generators, the average value is about 2.5 seconds, which is primarily due to a cold start. You can also see from the graph that for 500 generators the delay decreases over time, this is due to the automatic scaling of deployed resources.

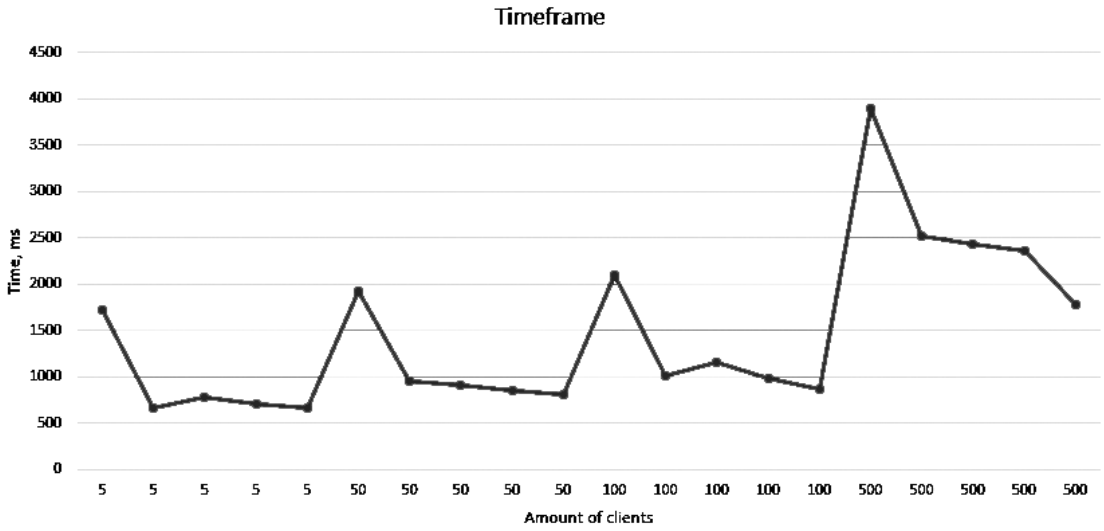

Figure 4. Measurement results

Conclusion

The implementation of software for data collection and monitoring systems has developed further. Proposed software system for data collection and monitoring of vehicle status is different from others with the ability to integrate with any vehicles and focused on the use of both individual users and businesses.

REFERENCES

1. Fan, Y., Khattak, A. J., & Shay, E. (2007). Intelligent transportation systems: What do publications and patents tell us? Journal of Intelligent Transportation Systems: Technology, Planning, and Operations, 11(2), 91-103.

2. Rubel H.L., Likhouzova T.A. Problems with effective traffic management // Inter-branch scientific and technological digest «Adaptive systems of automatic control» № 1(38), 2021 – p.62-67

3. Kuznetsov Denis, Zakharova Maria, Liuta Maiia (2021) // Criteria for evaluation of efficiency of remote administration software // Young Scientist #1 (89) January, 2021, pp 129-132

4. Fleetio: Fleet Management Software and Maintenance System // URL: https://www.fleetio.com/ [accessed 23.10.2021]

5. Momentum IoT // URL: https://momentumiot.com/ [accessed 23.10.2021]

6. Onfleet | Power your deliveries // URL: https://onfleet.com/ [accessed 23.10.2021]

7. IBM Availability levels // URL: https://www.ibm.com/docs/en/i/7.3?topic=roadmap-deciding-what-level-availability-you-need [accessed: 2021-11-01]

8. Krzysztof Zatwarnicki, Anna Zatwarnicka (2012) // Estimation of Web Page Download Time // International Conference on Computer Networks (2012) // Poland, Opole

9. Ingeno, Joseph (2018) // Software Architect's Handbook // Packt Publishing. p. 175. ISBN 178862406-8.

10.Robert C. Martin (2004) // YAGNI // URL: https://www.artima.com/weblogs/viewpost.jsp?thread=36529 [accessed 23.10.2021]

11.Jens Coldewey (2010) // Decoupling of Object-Oriented Systems p.4-6. // Coldewey Consulting, Munich, Germany

12.Seth Gilbert, Nancy A. Lynch (2012) // Perspectives on the CAP Theorem // Computer Magazine pp.30-36 vol.45 no. 02

13.CosmosDB Overview // URL: https://azure.microsoft.com/ru-ru/services/cosmos-db/#overview [accessed 23.10.2021]

14.Robert T Mason (2015) // NoSQL Databases and Data Modeling Techniques for a Document-oriented NoSQL Database // Proceedings of Informing Science & IT Education Conference (InSITE), 2015