

ITC 2/49 Information Technology and Control Vol. 49 / No. 2 / 2020 pp. 206-223 DOI 10.5755/j01.itc.49.2.23757	An API-first Methodology for Designing a Microservice-based Backend as a Service Platform	
	Received 2019/07/02	Accepted after revision 2020/01/27
	 http://dx.doi.org/10.5755/j01.itc.49.2.23757	

HOW TO CITE: Dudjak, M., Martinović, G. (2020). An API-first Methodology for Designing a Microservice-based Backend as a Service Platform. *Information Technology and Control*, 49(2), 206-223. <https://doi.org/10.5755/j01.itc.49.2.23757>

An API-first Methodology for Designing a Microservice-based Backend as a Service Platform

Mario Dudjak, Goran Martinović

Faculty of Electrical Engineering, Computer Science and Information Technology; J. J. Strossmayer University of Osijek; Kneza Trpimira 2B, 31000, Osijek, Croatia;
phone: +385 95 828 3101; e-mails: {mario.dudjak, goran.martinovic}@ferit.hr

Corresponding author: mario.dudjak@ferit.hr

Over the last several years, cloud computing has grown into a major paradigm in software development by providing computer resources over the Internet. Among various cloud service models, Backend as a Service (BaaS) stands out as a model that targets the specific needs of web and mobile developers. By providing the backend for applications, it facilitates and expedites the software development process. In order to prevent major problems with the use of third-party BaaS providers, this paper advocates building your own BaaS platform, as well as several works ahead of it. However, the development of a BaaS platform carries various challenges regarding architecture and design. This paper strives to define the core service offerings of a BaaS platform and to propose a method for providing an architectural design of a BaaS platform based on a microservice architecture. Microservice architecture is the preferred architectural style for cloud solutions since it promotes loose coupling, ease of scaling and integration with third-party services, which are fundamental stipulations of BaaS platforms. The methodology adopted in designing a microservice-based BaaS platform was formed in accordance with an Application Programming Interface (API)-first approach, which strives to design a suitable, representative API of the platform. To the best of authors' knowledge, this paper proposes the lowest-level design of a BaaS platform so far, describing the entity relations, integration patterns, and communication styles. Ultimately, the proposed design was implemented and tested for its functional requirements. In that regard, specific test cases that mirror the actual workflow of the BaaS platform were constructed.

KEYWORDS: API-first approach, API testing, Backend as a Service (BaaS), cloud computing, microservice architecture.

1. Introduction

Cloud computing is becoming mainstream in the area of Information Technology (IT) infrastructures, offering many diverse services that effectuate IT-related tasks for enterprises. One of the reasons why each generation of IT infrastructures appeared was the need for increased speed to market [35]. Within the cloud computing paradigm, the emergence of novel cloud service models introduces additional layers of abstraction to facilitate and expedite IT-related tasks. In the domain of web and mobile application development, the latest cloud service model which tends to increase speed to market is Backend as a Service (BaaS). BaaS allows developers to focus on application features by replacing backend development with connecting to an Application Programming Interface (API). In this decade, the BaaS market has been grown considerably as more and more developers adopt BaaS services. Demand for rapid deployment and development is one of the major drivers responsible for the growth of the BaaS market [23].

On the other hand, the utilization of a BaaS platform in application development can result in major problems. Use of third-party services carries drawbacks such as questionable security, vendor lock-in, and platform shutdown. Although the BaaS cloud service model has only recently been introduced, several major providers have already announced the shutdown of their platform. The earliest BaaS platform provider, Parse, shut down its platform in 2017 and thus jeopardized businesses that based their applications on the platform [27]. Likewise, another major BaaS platform provider, Apigee, announced the end of life for its platform in the middle of 2019 [4]. Considering that a small number of key vendors hold most of the BaaS market, closing any of them could cause the downfall of the BaaS cloud service model, regardless of its undisputed advantages. Overall, in order for small businesses to take advantage of such a model, they must develop their own platform. Development of your own BaaS platform eliminates the pointed drawbacks and at the same time enables all benefits of that service model. However, unlike writing an application-specific backend, BaaS services must be uniform and reusable, which requires a set of specific design patterns.

Typically, the BaaS platform consists of several independent service offerings [9, 10] and the main ar-

chitectural issue is to design a mechanism for their communication and synchronization. Given that a particular service acts as data storage for the overall platform, application-specific models and relations need to be abstracted in order to be used from other services. Recent propositions of the BaaS platform design are made up of either single service offering [10], or a number of services coupled in monolithic architectural style [9]. The former does not address the complication of data context sharing while the latter is troublesome to extend in case of adding new service offerings. This paper proposes the appropriate architectural design of a BaaS platform. Considered design patterns originate from a microservice architecture which is justified as the most suitable architecture pattern for the cloud solutions due to the promotion of loose coupling between services and independent scaling capabilities. Nonetheless, studied patterns need to be adjusted to the BaaS domain. The proposed design patterns define data sharing, messaging, and orchestration processes.

Overall, researchers have not treated the design of a BaaS platform in much detail. For instance, the BaaS cloud service model constitutes the foundation of the recently proposed frameworks in healthcare [18], wildlife conservation [8], mobile banking [20], education [42] and smart city domains [1, 15]. However, in all proposed solutions the mere introduction of a BaaS platform is considered as a contribution, regardless of the architecture and the way of implementing such a platform. On the other hand, authors in [38], found that mobile applications largely depend on platform-specific APIs, and concluded that the extent of dependence on obscure platforms may be an indicator of poor software quality. The objective of this paper is three-fold: (1) a review and definition of the core service offerings of a BaaS platform, (2) a method for providing an architectural design of a BaaS platform based on a microservice architecture, and (3) the platform implementation and the design of test cases that mirror the actual functional workflow of the platform. The contribution of the paper is a proposal for the architecture and design of the BaaS cloud service model which consists of three common service offerings, in the hope of helping small businesses in developing their own BaaS platform. The

novelty of this work is that it follows an API-first approach, by first comparing the offerings of commercial BaaS platforms and identifying key services, then designing suitable APIs, and ultimately proposing a design-supporting architecture. In addition, the platform consists of as many as three services, unlike most previous works that only exhibited one.

The remaining part of this paper has been divided into four parts. Section 2 begins with establishing foundations of the BaaS platform and a microservice architecture. Section 3 is concerned with the methodology used for the design of a BaaS platform. In Section 4, the functionality of the designed platform is evaluated by performing API testing and the client-level comparison was conducted. Finally, some conclusions are drawn in Section 5.

2. Foundations

2.1. Backend as a Service

Backend as a Service (BaaS) is a cloud service model which provides a way of connecting mobile and web applications with cloud-based backend services [21]. The most prevalent such services are data storage, user management, file storage, geolocation, push notifications, social integration, and analytics. BaaS delivers an infrastructure that can be automatically scaled and optimized, linked with a set of backend services. Therefore, BaaS represents an extension of the Platform as a Service (PaaS) model, specialized in simplifying the development of mobile and web applications. The abstraction of infrastructure management in such a way enables developers to focus only on building application features. For this reason, the main benefits of BaaS are increased speed to market, lower development cost, and higher scalability.

The early leader in providing cloud-based backend services in the form of a platform was Parse, which was later acquired by Facebook and ultimately shut down. Some of the largest BaaS platforms that have appeared after Parse are Google Firebase, Kinvey, Appcelerator Cloud and Backendless. Each of the above providers offers distinct advantages over others, although most of them possess the equivalent set of service offerings. At its core, BaaS eases linking applications to backend cloud storage and pairs it with the administration and authentication tools. That being the case, data storage, user and application manage-

ment services can be defined as the core service offerings of a BaaS platform. Tan et. al [40], conducted a study on three mobile BaaS (mBaaS) providers (Kinvey, App42 and Backendless), evaluating five different metrics, with an emphasis on data storage, user management and push notifications. The requirements set for each individual service during this study, have been taken into account when defining functional requirements of core service offerings in this paper. The five evaluated metrics were availability, processing services, computing services, portability, and reconfiguration, which were considered both quantitatively and qualitatively. Performed comparative study has shown that all three services are quite similar. Moreover, Colombo-Mendoza et. al [12] extended the PaaS cloud service model over a mobile ecosystem in the form of the novel platform. The proposed platform was then validated by performing qualitative-comparative evaluation and measuring three metrics - ease of learning, ease of use as well as current knowledge and skills of developers. In a similar fashion, this paper proposes the lowest-level design of a BaaS platform so far and validates it quantitatively by measuring commonly used metrics in object-oriented design. Very little was found in the literature on the question of designing a BaaS platform. Thus far, few studies have suggested that such a platform ought to be distributed in accordance with its service offerings. However, what they lack is a proposition of design patterns used for solving common design challenges specific to BaaS, on top of the proposed architecture. In his exploration of the concepts of a BaaS platform, Carter [10] defined, architected and designed heterogeneous micro-applications based platform. Designed platform acted as an API gateway and each micro-application was implemented as a Representational State Transfer (REST) API. Unlike the large commercial platforms that offer a wide range of services, this platform consists of a data storage service only. Given that the platform consisted only of data storage service offering, the research did not inform on means of communication, integration and sharing data context with forthcoming services. As mentioned in the future state section of the paper, BaaS platform should be composed of applications on different platforms, which signified the transition from micro-applications to microservice architecture. Carranza-García et al. [9] introduced a framework intended to facilitate the development of BaaS plat-

forms that target various Internet of Things (IoT) systems. The introduced framework enables developers to specify the behavior of desired web services and automatically generates compatible models, communication services and documentation. Unlike work in [10], the proposed framework contains a set of pre-defined services that are most common in BaaS service models, such as data storage, user management, and file storage. However, the framework is based on prototypical service-oriented architecture (SOA 2.0), which is increasingly being replaced with microservice architecture when building cloud solutions. The proposed framework was implemented as a Web platform, making it difficult to add new services. Although it enables the defining of custom services, their architecture is limited within the default technologies and specifications of the platform design. Gropengießer et al. [16], presented core components required for implementing Database Backend as a Service model in a cloud and introduced the overall framework based on model-driven software development techniques. As in [10], the only system component is data storage, which is also implemented as an independent service with its own database, Object-Relational Mapping (ORM) framework and REST interface. The paper describes deployment and monitoring patterns in much more detail but use relational database for storing user-specified conceptual schema. By infrastructure, the proposed DBaaS platform is most similar to the Alibaba Cloud DBaaS service, which only facilitates database hosting and maintenance. Unlike the work in [10], this paper swaps relational for non-relational database, introduces two additional services and proposes integration patterns as well as communication styles for both internal and external communication.

The main drawback of previous works on the topic of BaaS platform design is that they propose a single-service platform, although they state that such platforms must be heterogeneous and extensible. The only multi-service platform is the one proposed in [9], but it is based on the SOA architecture and implemented as a web platform, making it difficult to expand with new services. Furthermore, the DBaaS platform proposed in [16] provides exclusively structured data storage, with the developer still having to define the data relationships himself. This paper proposes a new method for designing a BaaS platform that employs an API-first approach. The API-first approach is used in this paper to establish the design of a BaaS platform that

provides the most significant service offerings of commercial platforms, which architectures and designs are difficult to obtain. In addition, the designed BaaS platform is based on a microservice architecture that allows for the easy addition of new technology-independent services. Unlike previous works, this method proposes integration and communication patterns for the consolidated operation of microservices.

2.2. Microservice Architecture

Microservice architecture has turned out to be a significant architectural style for building distributed applications. In a microservice architecture, a single application is built as a collection of small services, each operating in its own process and communicating with various mechanisms [41]. Moreover, such an architecture promotes loose coupling, i.e. minimizing the dependencies between two or more services. Considering that cloud computing solutions require a loosely coupled architecture [19], microservice architecture is the preferred architectural style for the cloud. In a traditional monolithic architectural style, an application is built as a single component in which the slightest change requires rebuilding and re-deployment to make changes come into effect. In response to a growing amount of the work, monolithic applications can be scaled horizontally by replicating an entire application on multiple servers. On the other hand, due to the separation of application functionalities into services, microservice architecture scales by distributing services across servers, recreating as required. Villamizar et al. [43] evaluated the implications of using microservice architecture by comparing performance tests executed on two equivalent applications developed on monolithic and microservice architecture, respectively. They concluded that microservice architecture did not considerably impact the latency of responses due to the use of more hosts and suggested that microservices should be utilized in applications with hundreds of thousands or millions of users because each microservice can scale independently using different policies. Furthermore, microservice architecture is often misinterpreted as service oriented architecture (SOA), which is commonly utilized in maintenance systems in communication networks [22, 34]. Nonetheless, services in a microservice architecture can operate independently of other services, unlike in SOA, which makes new services easier to deploy and scale.

Several systematic reviews of microservice architecture have been undertaken. In their systematic mapping study, Taibi et. al. [39] extracted common patterns and principles employed in the adoption of the microservice architectural style. Extracted patterns were classified into three subsections: orchestration, data storage, and deployment patterns. In his discussion on the design of microservices, Sill [36] pointed out that putting a microservice architecture in practice, requires proper addressing of issues such as data exchange, messaging and orchestration. Furthermore, he referred to existing standards that provide the basis for resolving mentioned issues. Together these studies provide important insights into the typical design patterns for a microservice architecture. Those insights have been taken into considerations in this paper when integrating microservices that arose from an API-first approach.

3. Proposed Method

The methodology adopted in designing a microservice-based BaaS platform was formed in accordance with an API-first approach. This approach suggests that the software development process ought to start with designing and implementing appurtenant APIs [7]. An API-first approach facilitates decomposing of an application into autonomous microservices and is therefore exceptionally helpful for applications that require loose coupling [33]. As a result, each microservice is represented with a unique API but can be efficiently developed for various devices. The method proposed for designing a microservice-based BaaS platform decomposes the API into individual services and integrates them into a microservice architecture. The proposed method consists of four steps, which are described in more detail below.

Step 1: A design of a BaaS platform is required to be coherent and extensible in case of adding new service offerings. Therefore, only core service offerings were selected to be provided by a proposed platform. As mentioned in Section 2, core service offerings of a BaaS platform are data storage, user management, and app management. In this step, a comparison of core service offerings between major existing platforms, Firebase, Kinvey, Backendless, and Alibaba Cloud, was drawn. Backendless is a platform most similar

in concept to the BaaS cloud model, and all platform services can be classified into 2 categories: user management and data storage. User management service enables user registration, login, logout and password recovery functionalities, while data storage service provides both SQL-driven relations and NoSQL schema management. Security of the Backendless platform is role-based, where every single role has a set of permissions of each API. Alibaba Cloud and Firebase platforms provide a comprehensive infrastructure of global cloud computing services, some of which are common services of the BaaS platform. Alibaba Cloud platform provides various cloud computing models such as IaaS, PaaS, DBaaS, and SaaS, which can be managed from the administration dashboard of the platform. Of the models mentioned, the closest to the BaaS model is DBaaS which only facilitates database hosting and maintenance. On the other hand, Firebase provides backend that facilitates storing and syncing data between users using a cloud-hosted NoSQL database, managing user authentication and storing and sharing user-generated files.

During the comparison, mutual functional requirements were derived. Considering the huge heterogeneous user population of a BaaS platform, detailed specification of requirements is of great relevance in order to develop this type of system [2].

As a result, Table 1 presents the functional requirements of each core service offering that must be fulfilled.

Table 1

Functional requirements of service offerings

Service offering	Functional requirements
Data storage	<ul style="list-style-type: none"> - CRUD operations and filtering on collections - Defining validation rules for entities in a collection - CRUD operations on arbitrary data
User management	<ul style="list-style-type: none"> - CRUD operations on user resource: register, login, password recovery, logout - Managing user access by defining roles and permissions - CRUD operations on roles - CRUD operations on permissions
App management	<ul style="list-style-type: none"> - Registering new application - CRUD operations on application resource

Step 2: Throughout this step, functional requirements of service offerings were mapped into API capabilities which were then turned into API contracts. Each of the corresponding contracts was designed as REST API, considering its ease of connection with mobile and web applications. While developing the API contracts, a methodology called spec-driven development was employed. API specification provides a thorough insight into API behavior and its linkage with other APIs [37]. Such specification consists of design specifications for three different concepts of an API: resources, actions, and security. For the purpose of writing an API specification, SwaggerHub tool and Swagger 2.0 specification language were used.

The relationships between the resources of the BaaS platform are shown in Figure 1. Each resource belongs to a particular application. The basic unit of data storage is an entity, and entities of the same type are organized in collections. In addition, one user can be linked to many entities, which are created by that user. In the user management service, every resource has a many-to-many relationship with others, which is the foundation for role-based security.

Figure 1

Relationships between resources of the BaaS platform

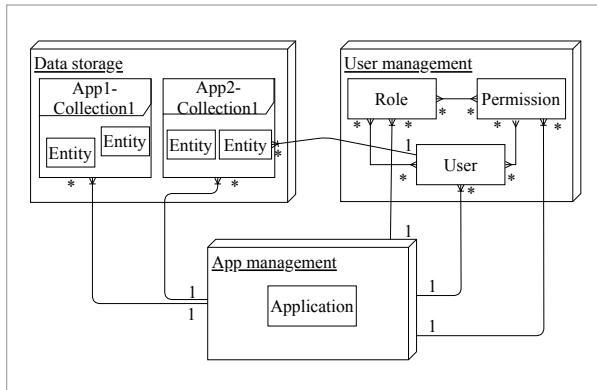


Table 2 shows the design of role-based security of the BaaS platform, in terms of authorization and authentication styles. BaaS platform manages application users and may group them into roles based on security permissions they share. Permissions provide users with access to perform actions on specific resources. Roles and permissions are managed by application admins. Therefore, the platform supports two levels of authentication: application user and application

Table 2

Security design of the BaaS platform

Security design		
	Authorization type	OAuth2
Authorization	Access control entities	<ul style="list-style-type: none"> - Permissions – holding definition of access rules specified by API endpoint - Roles – a group of permissions associated with a user
	Authentication type	<ul style="list-style-type: none"> - Bearer token in Authorization header of the request
Authentication	Authentication levels	<ul style="list-style-type: none"> - Application user – user access on application resources, based on permissions and roles - Application admin – full access to application resources

admin. Application user has access to data storage and user management services, as regulated by permission rules. Application admin has full access to application-related resources and to the application management service.

Each service offering contains resource models, which define how data is stored within a service, but also how data is transferred between services of the BaaS platform. Given the REST architectural style of each service offering, resources were modeled as JavaScript Object Notation (JSON) objects. In that way, resources are technology agnostic and available through Hypertext Transfer Protocol (HTTP) requests. Data storage service allows saving arbitrary data, modeled as JSON objects, in the property named data of an entity resource. Entity resource also contains user_id property, which binds it to a user. Knowing which resources belong to which user is of great importance when a user wants to update his model or perform bulk actions on his entities. Collection resource holds schema property, which determines whether the platform should validate the creation and updating of entities and discard them if they do not comply with the schema. Besides entity, each model contains application property, which is used to distinct resources on the application level. Application resource incorporates publicKey and privateKey properties, which are used to compose HTTP requests and manage application data, respectively.

Table 3 presents an overview of actions from each service offering of the BaaS platform. In accordance with functional requirements, each API provides cre-

ate, read, update and delete (CRUD) operations for predetermined resources. In this respect, the BaaS platform differs from a backend framework, cover-

Table 3

Designed actions of the BaaS platform

	API Endpoint	HTTP Method	Semantics
Data storage	/data	GET	Retrieve all collections
		POST	Create a collection
	/data?searchQuery={searchQuery}	GET	Query collections
		GET	Get the collection by id
		PUT	Update the collection
	/data{id}	DELETE	Delete the collection
		GET	Retrieve all entities from the collection
		POST	Create an entity in the collection
	/data/{coll_name}?searchQuery={searchQuery}	GET	Query entities in the collection
		GET	Get the entity by id
User management	/data/{coll_name}/{id}	PUT	Update the entity
		DELETE	Delete the entity
	/users	GET	Retrieve all users
		POST	Create a user
	/users/login	POST	Login a user
	/users/password-recovery	POST	Initiates a password recover process for the user
	/users/{id}	GET	Get the user by id
		PUT	Update the user
		DELETE	Delete the user
	/roles	GET	Retrieve all roles
		POST	Create a role
	/roles/{id}	DELETE	Delete the role
	/roles/{id}/users	GET	Get users in the role
		POST	Add a user to the role
App management	/roles/{id}/users/{user_id}	DELETE	Delete the user from the role
	/roles/{id}/permissions	GET	Get permissions in the role
		POST	Add permission in the role
		DELETE	Delete permission to the role
	/management/apps	POST	Create an application
	/management/apps/{id}	GET	Get the application by id
		PUT	Update the application

ing most of the configuration process of the backend services. In contrast to backend framework where developers have to create tables, define relationships and develop interfaces, the BaaS platform requires only resource models to be provided through the pre-defined interfaces and the platform automatically generates relationships between them and takes account of scalability. The provided operations were designed as asynchronous, to avoid blocking client applications. All stated API endpoints are relative to the API entry point: /{app_key}. Thereby, each request is application-specific.

According to [30], not all BaaS providers offer separate access for application developers and users, thus leaving end user' data at risk and application vulnerable for data manipulation, exploitation, and misuse. Actions of managing users, collections and applications only need to be taken by developers through separate access channels, or in the proposed design, through the application admin authentication level.

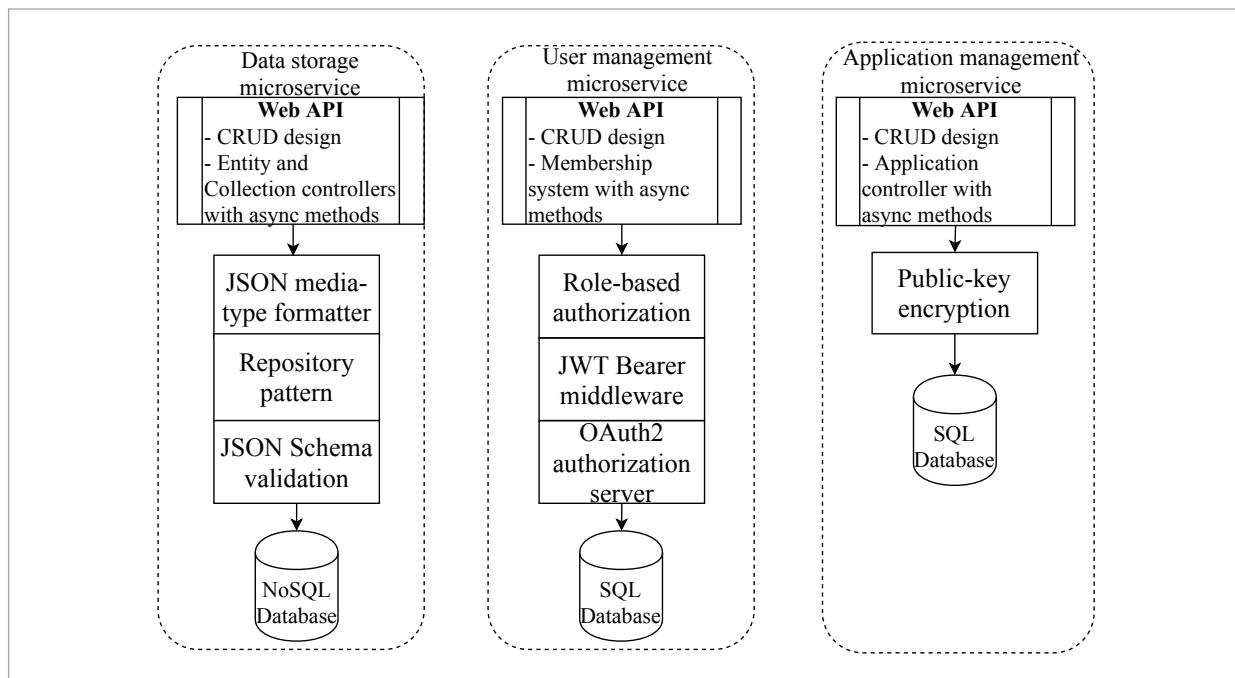
Step 3: Upon the design of API contracts, a further step was to decompose those contracts into microservices. Many different languages and frameworks for the implementation of each individual microservice

were available. As a starting point in this step, API implementation stubs for ASP.NET Core 2.0 platform were generated by utilizing SwaggerHub tool. Generated stubs consisted only of methods that do not contain any programming logic. Thereafter, each service offering was implemented individually as a web API service on the selected platform. Depending on the defined requirements, each web API service was composed of various architecture patterns and technologies, which ultimately leads to having a polyglot microservice architecture.

Figure 2 presents the multi-architectural patterns of implemented microservices. A non-relational (NoSQL) database is selected for the data storage microservice, given that this service ought to enable storing arbitrary data of a flexible structure. Even though the main advantage of NoSQL databases is horizontal scaling, the database design was conducted bearing in mind that the BaaS platform can be simultaneously used by multiple applications. For the sake of business requirements, the platform cannot employ a single collection to store data coming from different applications. Developers on top of the BaaS platform have the ability to define validation schemes

Figure 2

The internal architectural patterns of the implemented microservices



for their collections, and the platform should not allow the imposition of the same validation rules in another application. Therefore, each client has the ability to create collections and store data at his own choice. User management microservice provides the developers with the basic authentication, authorization and user management methods in the form of a membership system. The service act as both resource and authorization server, according to Open Authorization (OAuth2) protocol terminology. Finally, application management microservice allows application management to anyone with an application private key, which is produced by performing public-key encryption of a client-defined public key. The technology selection should not in any means disrupt the proposed design method, as long as stated implementation requirements are fulfilled.

Step 4: Integration of microservices is indispensable for their consolidated work and is considered as the most important aspect of a microservice architecture [27]. In the previous steps, the business modeling of the autonomous microservices was carried out by mapping them to the service offerings. The obtained degrees of cohesiveness and coupling are the consequences of the previous steps. In this step, design patterns for different styles of microservices integration were customized to the domain of the designed BaaS platform. Prior to customization, the design patterns were derived from the studied literature. In order to select the appropriate style of the microservices integration, two presumptions stemming from the previous design steps were set out:

- 1 Data of each microservice are kept private to that service, i.e. architecture implements database per service pattern.

- 2 Each microservice communicates with client applications and other microservices asynchronously.

These underlying presumptions inevitably point to a certain style of integration. The microservices of BaaS platform were integrated by combining the two integration styles. Business processes were managed by a choreography system while an orchestrator system supported cross-cutting concerns, such as authentication.

A microservice architecture differentiates between two types of communication: client-to-service and inter-service. In the case of internal communication between the microservices of the BaaS platform, the event-based style of collaboration was employed. Owing to the prior microservices modeling, strong data consistency in the BaaS platform is not required. Of all BaaS platform business processes, only the process of deleting a user span across the boundary of the associated microservice. Figure 3 shows the implementation of an event-based communication for the DeleteUser operation. After deleting a user, the user management service publishes the DeleteUser event, and data storage service must subscribe to that event to delete related entities. The publish and subscribe system is performed on an event bus. Since the operation of deleting a user is not tightly coupled with the operation of deleting related entities, there is no need to block the former while waiting on the latter. Therefore, by employing an event-based style of collaboration, loose coupling between the microservices was preserved.

In the case of client-to-microservice communication, the request-response style of collaboration was employed. Given the asynchronous way of communica-

Figure 3

Event-based communication for the *DeleteUser* operation

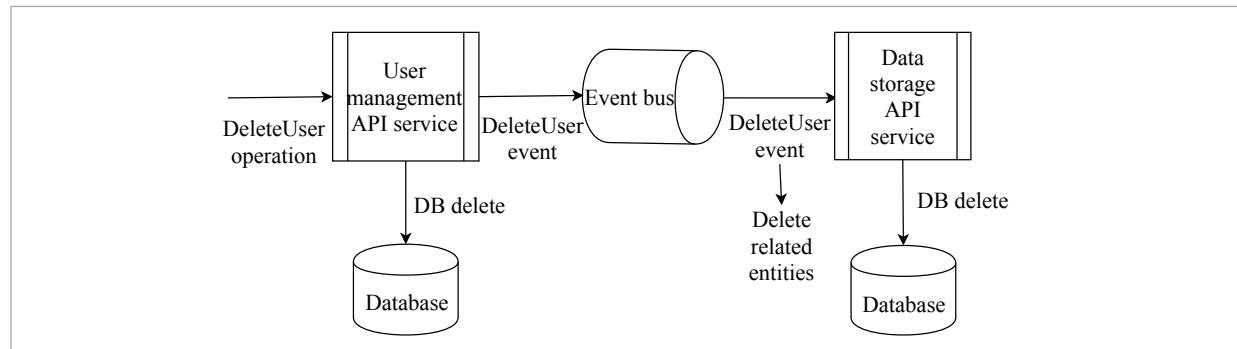
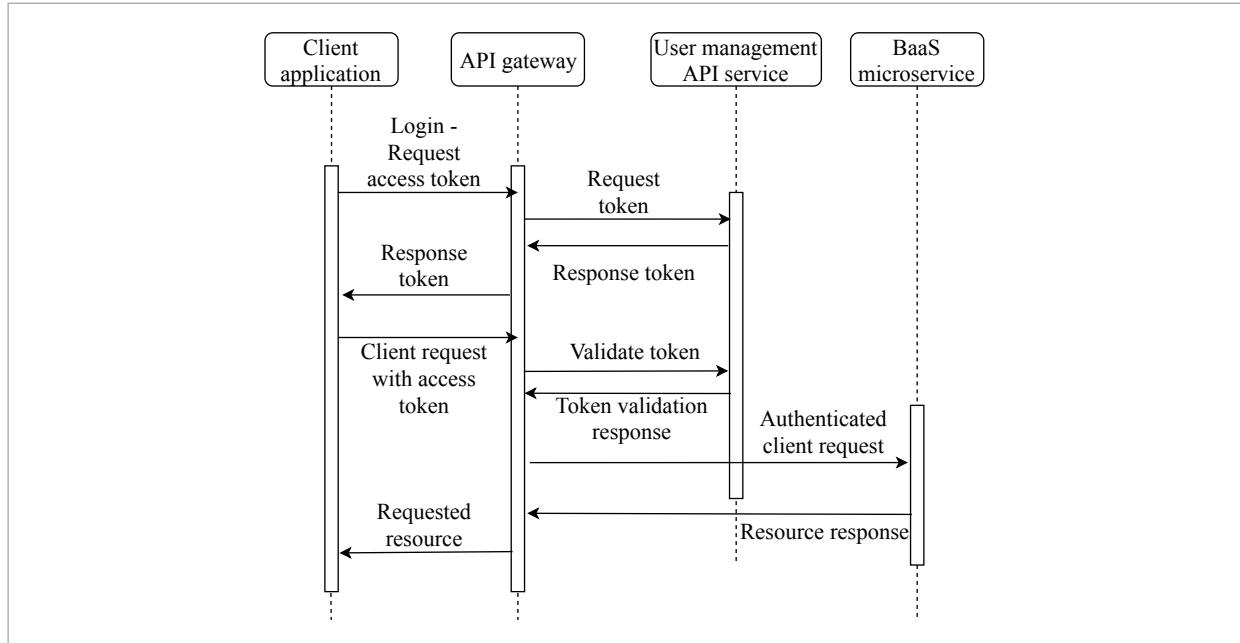


Figure 4

Authentication process in the BaaS platform



tion, client application dispatches a request and registers for a callback which notifies the client when the request has completed. To minimize the latency which occurs by a client sending multiple requests directly to microservices, the unique entry point for client requests in the form of an API gateway was implemented. API gateway of the BaaS platform efficiently aggregates responses of the overall backend services, by handling cross-cutting concerns. Figure 4 presents the authentication process in the BaaS platform that takes place through the API gateway. The user management microservice was defined as the identity service in the API gateway and it is invoked to perform the authentication and authorization process. To maintain statelessness, token authentication system was implemented. Upon successful login, the user obtains a JSON Web Token (JWT) that must be attached on subsequent requests. In this way, the API gateway establishes the user's identity and permissions.

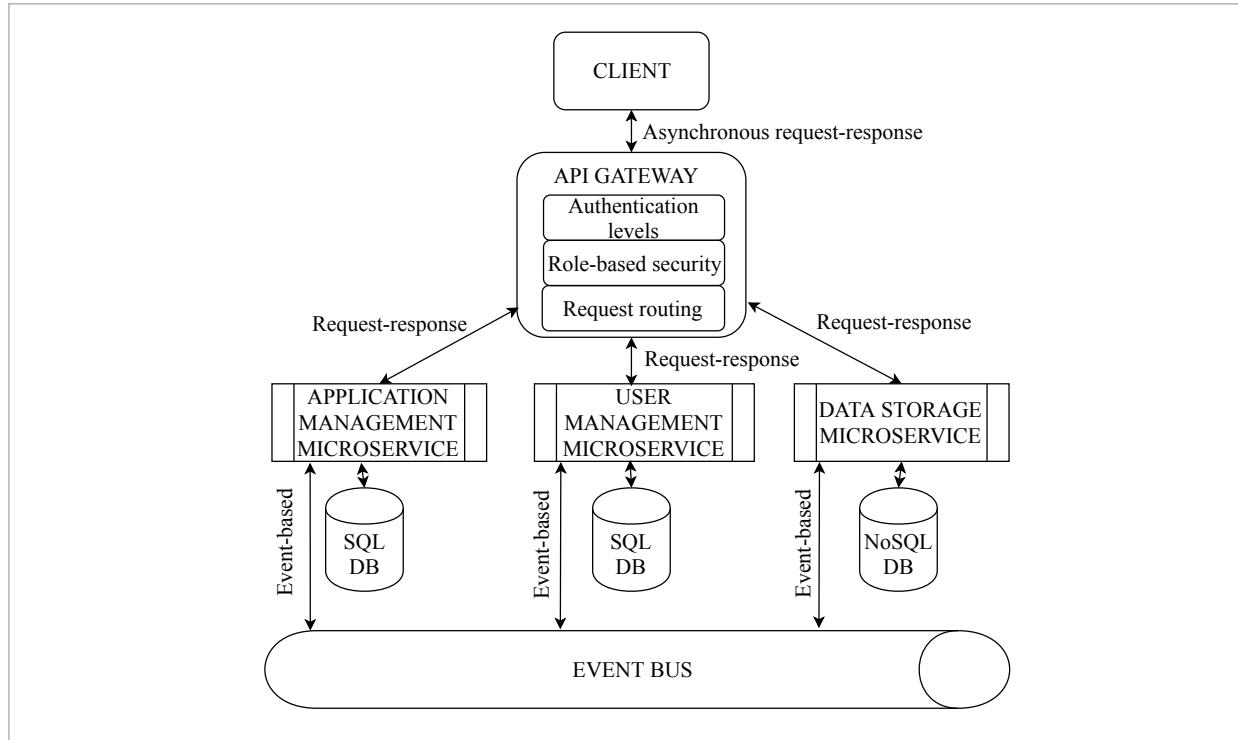
Another advantage of using the API gateway design pattern is that it allows the distribution of platform interaction with clients, depending on the type of client application. For example, processing APIs from web and mobile applications may result in dif-

ferent performance indicators [6]. This is mainly because mobile applications present the same data less elaborate than web applications, given their physical limitations in screen size. Network performance is another aspect of the difference between the two types of applications, with the mobile network typically being much slower and having a much higher delay than the non-mobile network [32]. An additional backend in the form of the BaaS platform can preprocess API responses and provide customized, highly optimized protocols and data formats for communication with the mobile device. This demonstrates the need for different API gateways of one BaaS platform tailored for different types of clients. The proposed platform design contains only one gateway for all devices, but due to the use of this design pattern, the platform can easily be extended with additional gateways.

The overall architecture is presented in Figure 5. API gateway serves as an access point for all client requests returning the responses asynchronously. It routes requests to the individual microservices, calling the user management microservice to determine identity. The microservices interact with each other through the event bus.

Figure 5

The architecture of the designed BaaS platform



4. Testing and Analysis

As was stated in the beginning, the main part of the paper is the design of a microservice-based BaaS platform. Considering that the API-first approach was employed in the design of the BaaS platform, proposed design method was proven by performing API testing. Unlike traditional software testing, API testing is performed without the use of GUI based libraries but using API client libraries that directly test APIs in isolation [31]. More importantly, being a part of integration testing, API testing is more suitable for testing backend services associated with a microservice architecture. Among many types of API testing, functional testing verifies the functional correctness of the system and was therefore selected for the evaluation of the BaaS platform functionality.

Furthermore, this section evaluates the impact of using a developed BaaS platform in developing a client web application. As a mediator between the BaaS and

the web application, the Software Development Kit (SDK) was developed for a widely-used JavaScript framework Angular. The developed SDK eases a connection between a web application and the BaaS platform and hides the platform's internal structure from the end user. In order to assess the obtained speed and simplicity of development, a simple web application was developed in two ways:

- 1 Without BaaS platform.
- 2 On top of the designed BaaS platform.

In each of these modes of development, certain quality measurements, commonly used in object-oriented design to quantify the complexity or quality of web application, were conducted. The goal of this analysis is to determine the share of backend services in the developed web application to clarify the speed increase and complexity decrease of a web development process.

4.1. Implementation

The first prototype of the BaaS platform was implemented using the technologies listed in Table 4. Prior to the development of the autonomous microservices, SwaggerHub tool was used to generate the empty server stubs for ASP.NET Core framework. The other technologies were selected as supplementary to the framework, which results in facilitated integration.

Table 4

The implementation details of the BaaS platform

Subsystem	Software framework	Database mapping framework	Database
Data storage microservice	ASP.NET Core 2.0	MongoDB ODM	MongoDB 4.0
User management microservice	ASP.NET Core 2.0	Entity Framework Core	SQL Server 2017
	ASP.NET Core Identity		
App management microservice	ASP.NET Core 2.0	Entity Framework Core	SQL Server 2017
API gateway	Ocelot	-	-
Event bus	RabbitMQ	-	-

The client web application used for assessment of the impact of BaaS platform is developed in Angular 6 framework and represents a web gallery-like application. The developed web gallery meets the basic requirements of an application of this type, such as the ability to register users, create and differentiate user albums, upload photos, update album and photo information, and more. The developed application consists of four Angular components and of 15 functions. More implementation details and insights into application user interfaces can be found in [14], which is the author's previous work on the use of the BaaS system when developing web applications.

4.2. Test Cases

The first step of API testing was setting up a testing environment, i.e. setting up software and hardware required to execute test cases. The API testing was

conducted using Postman – a complete API development environment. Test cases were created as collections in Postman, given that they were composed of multiple requests. By writing pre-request and test scripts for each request in a collection, requests can be chained together thus creating a collection workflow. Responses from some requests were set in environment variables that other requests use in their URL, body or test scripts. Test scripts in Postman environment were written in JavaScript scripting language.

To evaluate platform functionality, two different test cases that mirror an actual workflow for both admins and users were defined. Admin's workflow consists of operations that an admin must perform to initialize the application. Upon registering, an admin needs to create an application, select a desired application public key and define application roles and collections. Actions of managing users, collections and applications are inherent only to the admins. User's workflow consists of nearly all platform operations that the user has access to, organized in a logical sequence. After setting up an account, the user manages his entities by performing various CRUD operations on an entity resource. Firstly, a user accesses an application through the application user level and enters desired account information. After that, he is immediately redirected to the login, upon which the user_id and access token are stored as environment variables. If the login process has passed successfully, the user can create, retrieve, update and delete the entities he owns.

4.3. Traditional Complexity Metrics of Web Application

The analysis of the developed web gallery application intends to determine the distribution of backend and frontend services in the entire application. Determining the share of background services in the web application can provide insight into how much the development process accelerates and how much its complexity decreases if an application is developed on top of the BaaS platform. The first step in web gallery analysis is to define the complexity metrics that simplify the way to determine the complexity and size of a web application. The metrics observed in this paper stem from work in [25], which proposes a set of metrics for size and complexity of web application derived from traditional object-oriented metrics.

In the conducted analysis, the following three metrics were selected to evaluate the complexity and size of the developed web application:

- Lines of Code (LOC) – the number of lines of code in a given class or file [6].
- Response for Class (RFC) – the number of methods that are called when a certain operation within a class is invoked [11].
- McCabe Cyclomatic Complexity Model (CCN) – the number of distinct paths through which a code segment can run [24].

Selected metrics help to determine the level of complexity and size of a web application, as well as normalize other measurements. For example, the very high cyclomatic complexity of a method suggests a high complexity of that method, but only if the method has a lower LOC value than other methods in the class. This is one of the reasons why it is important to collect measurements of all these metrics. The measurement was performed using the Visual Studio Code editor which features a Code Metrics Tool that in a simple way eases measurement of the aforementioned metrics. Measurement was performed both on file and function level for the corresponding functions in both backend and frontend code segments. The comparison of measured metrics is performed in such a way that each function of the web gallery is compared to the function of the BaaS platform that it calls. If some function of the web interface calls multiple BaaS functions, then the measurements for those functions are added.

4.4. Cognitive Complexity Metrics of Web Application

The traditional complexity measures listed are the most widely used complexity metrics for object-oriented software [29]. However, these metrics do not fully demonstrate the complexity of modern applications. The traditional complexity metrics are generally criticized for their utilization of mathematical models that do not take into account the relative complexity of certain code sequences from a programmer's perspective. Although they provide the measurements required to evaluate the software, they fail to indicate the reasons behind the obtained complexity. For this reason, Wang [44] introduces a new set of complexity metrics called cognitive complex-

ity metrics that use human judgment to assess how structures should be scored. In order to determine the complexity ratio of the BaaS platform and the web application from a cognitive complexity standpoint, several suitable metrics were selected. The cognitive complexity metrics observed in this paper are derived from the work in [26], which proposes a metric suite for evaluating the cognitive complexity of object-oriented software. In such metrics, specific weight is defined for each basic control sequence, indicating the complexity of its implementation.

In the analysis performed, the following three metrics were selected to evaluate the cognitive complexity of the developed web application:

- Attribute Complexity (AC) - the total number of attributes associated with a class [17].
- Method Complexity (MC) - the sum of cognitive weights of basic control sequences in a method. In the analysis, this metric is observed at the class level, so that the weights of all methods are summarized [13].
- Class Complexity (CLC) - the sum of the attribute complexity and all method complexities of a class [3].

All three metrics of cognitive complexity were measured at the class level. In the case of a developed web gallery, the class represents an individual Angular module or component. Given that in modern web applications, frontend development also stems from object-oriented principles, it is feasible to compare classes of web gallery and BaaS platform in this analysis. On the other hand, a class of the BaaS platform consists of one microservice, which is in fact a web API, and all the functions from the lower layers that class invokes. The metrics are compared by comparing each class of the web gallery with the class of BaaS platform that performs apposite backend operations. The SDK services are considered as function calls from the frontend part of the application.

4.5. Discussion of Results

The conducted functional API testing yielded results structured as a set of returned status code, duration, and size of each HTTP request in the defined workflows. All the operations returned the expected status codes, which means that the workflow was not compromised and that the designed platform was functional. Given the fact that the requests were chained

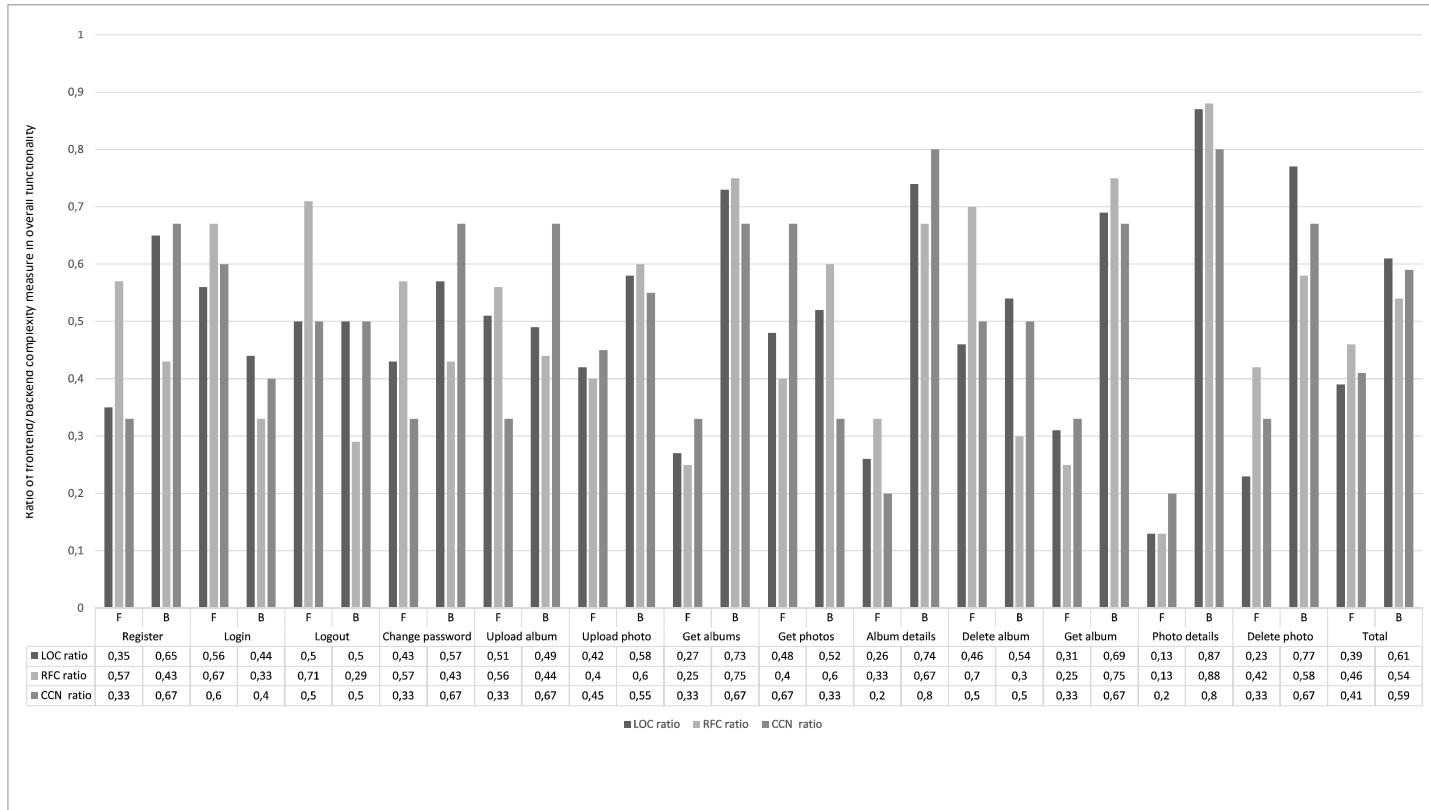
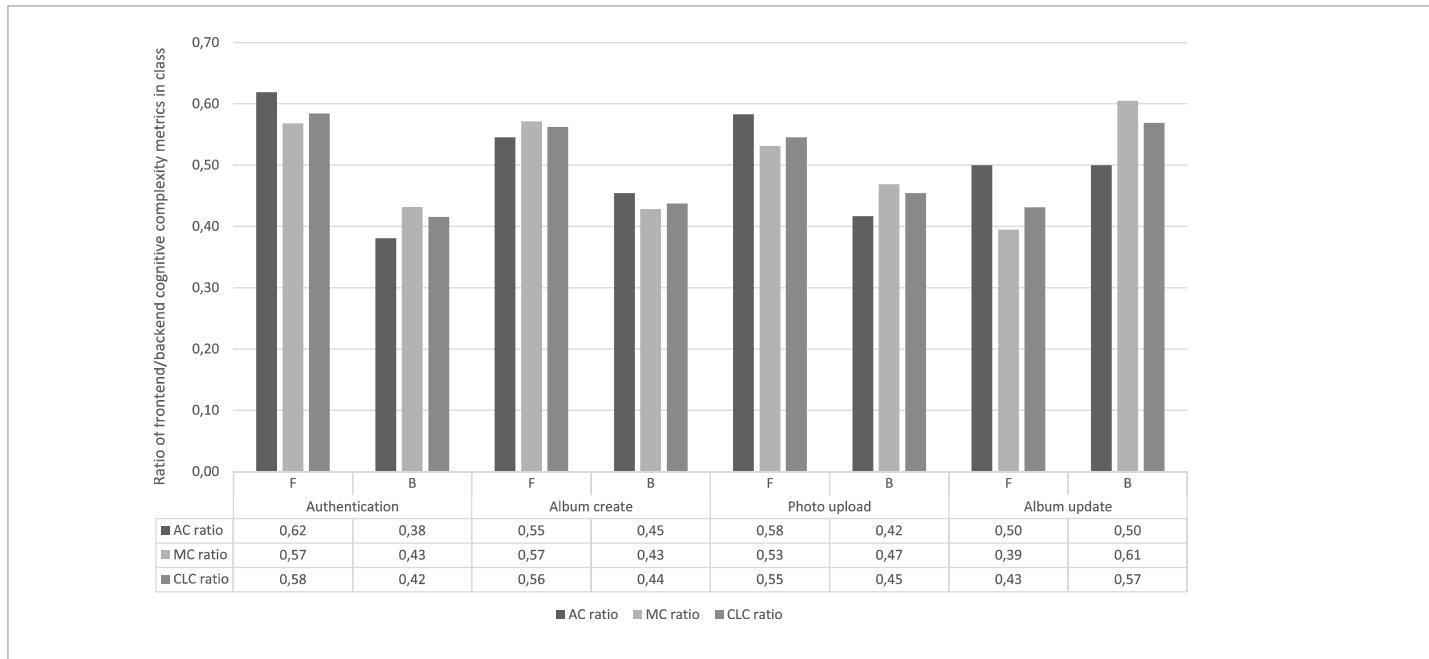
in a single workflow, failure of one request would indicate that either the failed request is inadequately constructed, or the previous requests failed to set the required environment variables. Therefore, in the conducted API functional testing, only the status codes were verified knowing in advance the structure of the status codes foreseen to be returned by the platform. By stating that all requests returned the expected status codes, it can be concluded that all requests were properly constructed and that they set the required environment variables. Taken together, these results demonstrate that the designed platform provides the ability to easily and quickly set up the backend of mobile or web application through several actions that the application admin must take. After that, an application user can configure his account and perform basic CRUD operations on his own resources through a client application, all supported by the BaaS platform.

Since the developers are the ultimate users of the BaaS platform, the purpose of this analysis was to take their stand in trying to determine how much these platforms contribute to the development process. Figure 6 displays the measurements of the selected software complexity metrics for both backend and frontend of the developed web gallery, where the backend is constituted with the services of the BaaS platform. The letter F denotes the frontend part of the developed application while B stands for the backend. Measurement comparison for the corresponding BaaS platform functions and web application functions can to a certain extent determine their complexity and the size ratio. The ratio of the measured quantity between the application backend and frontend is calculated for each metric. The first metric of importance is the LOC in a particular method. By observing the total sum of all code lines, it can be concluded that the share of backend code needed for web gallery to work is about 61% of the total code. Similarly applies to the RFC (54%) and CCN (59%) metrics. It is important to note that a certain part of the backend-specific code is replaced with the general uniform functions of the BaaS platform. Likewise, a certain code segment of the frontend is formed by invoking the services of the BaaS platform in the correct manner and in the appropriate order. The given RFC ratio of the total code indicates that a slightly larger number of methods

can be invoked from backend classes, thus the utilization of the BaaS platform could obviate more than half of debugging and testing efforts. Moreover, the calculated ratio of the CCN metric also indicates the possibility of halving the required number of tests by developing on top of the BaaS platform.

Figure 7 demonstrates the measurements of the selected cognitive complexity metrics for both backend and frontend of the developed web gallery. The web application consists of four separate components: authentication, album create, photo upload and album update component, as described in [14]. The ratios of cognitive complexity metrics are calculated at the class level, with the four components consisting of the functions shown in Figure 6 and additional functions. The ratios of values of cognitive complexity metrics are slightly different from those of traditional metrics. According to the cognitive complexity metrics, three of the four frontend components are more complex than the corresponding backend classes. One reason for this may be that some of the frontend functions operate solely on the client-side and do not perform function calls of BaaS services. On the other hand, these results indicate that the frontend components have multiple attributes and that their functions are composed of basic control sequences of higher weights. However, an album update component that incorporates more complex data storage logic results in higher values of MC and CLC metrics for the backend side.

It is very important to emphasize here that the results obtained depend largely on the implementation of the functions and classes presented. However, object-oriented standards were adhered to when implementing the web gallery, and efforts were made to develop features that resemble CRUD operations as closely as possible. From the results of traditional complexity metrics presented, it can be concluded that the backend part of the application is larger and more complex than the frontend part. However, in terms of the development effort from a programmer's perspective, the frontend part of the application has evinced to be more complex. These results support the stated benefits of a BaaS platform that seeks to reduce the superfluous backend that is recurring in most applications while allowing developers to focus on the development of frontend features.

Figure 6. Ratio of complexity metrics for BaaS platform and web gallery application**Figure 7.** Ratio of cognitive complexity metrics for BaaS platform and web gallery application

5. Conclusion

The aim of this paper was to propose a method for designing a microservice-based BaaS platform. By analyzing several platforms of major BaaS providers, it was concluded that most of them possess the equivalent set of service offerings. However, very little was found in the literature on the question of designing a BaaS platform. Several previous works proposed a high-level design, focusing mainly on data storage service offering without defining a proper integration and communication styles. To the best of authors' knowledge, this paper proposes the lowest-level design of a BaaS platform so far, describing the entity relations, integration patterns, and communication styles.

Microservice architecture proved to be a natural choice for the architecture of a BaaS platform because each service offering fits an individual micro-service. In this way, microservices can be scaled individually, depending on clients' needs. Additionally, a new service offering can be added as a new micro-service in the platform, without affecting the existing architecture. That is why the designed platform consists only of core service offerings that are present in each such platform. Results of the functional API testing indicated that the designed platform performs requested functionalities, designed specifically to mirror the admin and user ways of using the platform. The constructed test cases can be used to test future designs of a BaaS platform. This kind of BaaS platform testing has not yet been carried out in the literature, mainly due to the diversity of services provided by such platforms and could therefore be identified as a future research problem. In order for the design platform to be fully compared with the existing commercial platforms, a similar study should be carried out as described in Section 2, but this goes beyond the scope of this work. The emphasis here was solely on the low-level design of a functional platform. The proposed method contributes to the problem of designing a BaaS platform and provide a basis for adding new service offerings in existing architecture.

The final analysis of the size and complexity of the client web application shown that utilization of the proposed BaaS platform considerably facilitates and accelerates the web development process. About

60% of the developed web application is constituted by its backend. On the other hand, when looking at cognitive complexity, it can be concluded that the frontend part of the application is more complex in terms of development effort from a programmer's perspective. It is important to note that the acquired results depend majorly on the context of the web development process, which consists of the application domain, functionalities, development frameworks, languages, and other development tools. The terms in which the results of the conducted client-level comparison may be considered relevant for a future research imply a client-like application of similar functionality and program code such as a developed web photo gallery. The analysis carried out may be a reference for testing the advantages of the BaaS system, but the performer of the evaluation must be able to assess the weight of the influence of said development parameters on the results of the analysis. This study is limited by the lack of information on the architecture of the major platforms of the BaaS market. Notwithstanding these limitations, the proposed method may be a suitable starting point for small businesses that want to take advantage of a BaaS model while avoiding the risks of vendor lock-in and platform shutdown. Since this method begins by defining the backend features and then gives a detailed list of steps all independent of the choice of technology, it can be recommended for all those systems that depend on the third-party BaaS platform, to avoid potential vendor lock-in whilst acquiring trustworthy backend for future use. This research has thrown up many questions in need of further investigation. A further study could assess the adaptation of additional design patterns for a microservice architecture to achieve increased performance and scalability of a BaaS platform. Likewise, further research might explore novel service offerings, such as machine learning features for mobile and web use cases.

Acknowledgment

This work was supported by the European Regional Development Fund under the grants KK.01.1.1.0009 (DATACROSS), and KK.01.2.1.01.0127.

References

1. Abir, H., Khaled, R. A Smart City System Using Backend as a Service Approach: Biskra City Case Study. Proceedings of 5th International Symposium on Innovation in Information and Communication Technology (ISIICT), Jordan, India, October 31 - November 1, 2018, 1-7. <https://doi.org/10.1109/ISIICT.2018.8613725>
2. Aguilar, J. A., Garrigos, I., Mazon, J. N. Requirements Engineering in the Development Process of Web Systems: A Systematic Literature Review. *Acta Polytechnica Hungarica*, 2016, 13(3), 61-80. <https://doi.org/10.12700/APH.13.3.2016.3.4>
3. Akbarinasaji, S., Soltanifar, B., Çağlayan, B., Bener, A.B., Miranskyy, A., Filiz, A., Kramer, B.M., Tosun, A. A Metric Suite Proposal for Logical Dependency. Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, Austin, Texas, USA, May 14, 2016, 57-63. <https://doi.org/10.1145/2897695.2897704>
4. Apigee. API BaaS Deprecation and End of Life. <https://docs.apigee.com/release/notes/api-baas-eol>. Accessed on June 19, 2019.
5. Basili, V. R., Perricone, B. T. Software Errors and Complexity: An Empirical Investigation, 1983. <https://dl.acm.org/doi/10.1145/69605.2085>
6. Bermbach, D., Wittern, E. Benchmarking Web API Quality. Proceedings of the International Conference on Web Engineering, Lugano, Switzerland, June 6-9, 2016, 188-206. https://doi.org/10.1007/978-3-319-38791-8_11
7. Bloch, J. How to Design a Good API and Why It Matters. Proceedings of Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, Portland Oregon, USA, October 22-26, 2006, 506-507. <https://doi.org/10.1145/1176617.1176622>
8. Bonacic, C., Neyem, A., Vasquez, A. Live ANDES: Mobile-cloud Shared Workspace for Citizen Science and Wildlife Conservation. Proceedings of IEEE 11th International Conference on e-Science, Munich, Germany, August 31 - September 4, 2017, 29-30. <https://doi.org/10.1109/eScience.2015.64>
9. Carranza-García, F., Rodríguez-Domínguez, C., Garido, J. L., Guerrero-Contreras, G. BaaS-4US: A Framework to Develop Standard Backends as a Service for Ubiquitous Applications. Proceedings of 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS), Granada, Spain, December 14-16, 2016. <https://doi.org/10.1109/IUCC-CSS.2016.012>
10. Carter, B. Grow Your Own Backends-as-a-Service (BaaS) Platform. Proceedings of the GOCICT Conference College of Information & Computer Technology, Sullivan University, Louisville, USA, 2015.
11. Chidamber, S.R., Kemerer, C.F. Towards a metrics suite for object oriented design. Proceedings of the Conference on Object-oriented programming systems, languages, and applications, Phoenix Arizona, USA, November, 1991. <https://doi.org/10.1145/117954.117970>
12. Colombo-Mendoza, L. O., Alor-Hernández, G., Rodríguez-González, A., Valencia-García, R. MobiCloUP!: A PaaS for Cloud Services-based Mobile Applications. *Automated Software Engineering*, 2014, 21(3), 391-437. <https://doi.org/10.1007/s10515-014-0143-5>
13. Crasso, M., Mateos, C., Zunino, A., Misra, S., Polvorín, P. Assessing Cognitive Complexity in Java-based Object-oriented Systems: Metrics and Tool Support. *Computing and Informatics*, 2016, 35(3), 497-527.
14. Dudjak, M. Development of BaaS (Backend as a Service) System for Web Applications. Master thesis, Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek, Department of Software Engineering, 2018.
15. Fornia, A., Napoli, C., Tramontana, E. Cloud Services for On-Demand Vehicles Management. *Information Technology and Control*, 2017, 46(4), 484-498. <https://doi.org/10.5755/j01.itc.46.4.17331>
16. Gropengießer, F., Sattler, K. U. Database Backend as a Service: Automatic Generation, Deployment, and Management of Database Backends for Mobile Applications. *Datenbank-Spektrum*, 2014, 14(2), 85-95. <https://doi.org/10.1007/s13222-014-0157-y>
17. Husein, S., Oxley, A. A Coupling and Cohesion Metrics Suite for Object-oriented Software. Proceedings of International Conference on Computer Technology and Development, Kota Kinabalu, Malaysia, November 13- 15, 2009, 421-425. <https://doi.org/10.1109/ICCTD.2009.209>
18. Kato, T., Tanaka, T., Sugihara, S., Shimizu, K., Kudo, N. Trial Operation of a Cloud Service-based Three-dimensional Virtual Reality Tele-rehabilitation System for Stroke Patients. Proceedings of 11th International Conference on Computer Science & Education (ICCSE), Nagoya, Japan, August 23-25, 2016, 285-290. <https://doi.org/10.1109/ICCSE.2016.7581595>

19. Kavis, M. J. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014. <https://doi.org/10.1002/9781118691779>
20. Kumar, P.J., Sundaram, S. M. Enabling Cloud Adoption Based Secured Mobile Banking through Backend as a Service. Proceedings of the International Conference on Security in Cloud Computing ICSCC, 2017, 29-30.
21. Lane, K. Overview of the Backend as a Service (BaaS) Space. API Evangelist, 2015.
22. Lojka, T., Bundzel, M., Zolotová, I. Service-oriented Architecture and Cloud Manufacturing. *Acta Polytechnica Hungarica*, 2016, 13(6), 25-44. <https://doi.org/10.12700/APH.13.6.2016.6.2>
23. MarketsandMarkets. Backend as a Service (BaaS) Market worth 28. 10 Billion USD by 2020. <http://www.marketsandmarkets.com/PressReleases/baas.asp>. Accessed on June 19, 2019.
24. McCabe, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*, 1976, SE-2(4), 308-320. <https://doi.org/10.1109/TSE.1976.233837>
25. McNinch, C.A. *Measuring and Quantifying Web Application Design*. 2012.
26. Misra, S., Adewumi, A., Fernandez-Sanz, L., Damasevicius, R. A Suite of Object-oriented Cognitive Complexity Metrics. *IEEE Access*, 2018, 6, 8782-8796. <https://doi.org/10.1109/ACCESS.2018.2791344>
27. Newman, S. *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc., 2015.
28. Nguyen, P. *Mobile Backend as a Service: The Pros and Cons of Parse*, 2016.
29. Pressman, R. S. *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 2005.
30. Rasthofer, S., Arzt, S., Hahn, R., Kolhagen, M., Bodden, E. *(In)Security of Backend-as-a-Service*. 2015.
31. Richardson A. *Automating & Testing a REST API. Compendium Developments*. Great Britain, 2017.
32. Richardson, C. *Microservices Patterns*. Shelter Island: Manning Publications, 2018.
33. Rivero, J. M., Heil, S., Grigera, J., Gaedke, M., Rossi, G. MockAPI: An Agile Approach Supporting API-first Web Application Development. Proceedings of International Conference on Web Engineering, Berlin Heidelberg, July 8-12, 2013, 7-21. https://doi.org/10.1007/978-3-642-39200-9_4
34. Rodriguez-Martinez, L. C., Duran-Limon, H. A., Mora, M., Rodriguez, F.A. SOCA-DSEM: A Well-structured SOCA Development Systems Engineering Methodology. *Computer Science and Information Systems*, 2019, 16(1), 19-44. <https://doi.org/10.2298/CSIS170703035R>
35. Sareen, P. Cloud Computing: Types, Architecture, Applications, Concerns, Virtualization and Role of IT Governance in Cloud. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2013, 3(3).
36. Sill, A. The Design and Architecture of Microservices. *IEEE Cloud Computing*, 2016, 3(5), 76-80. <https://doi.org/10.1109/MCC.2016.111>
37. Stowe, M. *Undisturbed REST: A Guide to Designing the Perfect API*. Lulu.com, 2015.
38. Syer, M. D., Nagappan, M., Adams, B., Hassan, A. E. Studying the Relationship Between Source Code Quality and Mobile Platform Dependence. *Software Quality Journal*, 2015, 23(4), 485-508. <https://doi.org/10.1007/s11219-014-9238-2>
39. Taibi, D., Lenarduzzi, V., Pahl, C. Architectural Patterns for Microservices: A Systematic Mapping Study. *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, Madeira, Portugal, March 19-21, 2018, 221-232. <https://doi.org/10.5220/0006798302210232>
40. Tan, D. J., Tzi, G. Y., Lau, S. L. A Study on Cloud-based Backend for Crowd-sourced Sensor Data Collection Apps. *IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*, Langkawi, Malaysia, October 10-12, 2016, 46-51. <https://doi.org/10.1109/IC3e.2016.8009038>
41. Thönes, J. *Microservices*. *IEEE Software*, 2016, 32(1), 116-116. <https://doi.org/10.1109/MS.2015.11>
42. Vásquez-Ramírez, R., Bustos-Lopez, M., Alor-Hernández, G., Sanchez-Ramírez, C., García-Alcaraz, J.L. AthenaCloud: A Cloud-based Platform for Multi-device Educational Software Generation. *Computer Science and Information Systems*, 2016, 13(3), 957-981. <https://doi.org/10.2298/CSIS160807037V>
43. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S. Evaluating the Monolithic and Microservice Architecture Pattern to Deploy Web Applications in the Cloud. *Proceedings of the 10th Computing Colombian Conference (10CCC)*, Bogota, Colombia, November 23, 2015. <https://doi.org/10.1109/ColumbianCC.2015.7333476>
44. Wang, Y. On the Cognitive Informatics Foundations of Software Engineering. *Proceedings of the 3rd IEEE International Conference on Cognitive Informatics*, Victoria, Canada, August 17, 2004, 22-31. <https://doi.org/10.1109/COGINF.2004.1327456>