Micro-frontends: application of microservices to web front-ends

Article · May 2020

DOI: 10.22667/JISIS.2020.05.31.049

CITATIONS
18

READS
4,045

4 authors:

Andrei Pavlenko
Innopolis University
2 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Nursultan Askarbekuly
Innopolis University
16 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Swati Megha
Innopolis University
13 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Manuel Mazzara
Innopolis University
443 PUBLICATIONS **5,562** CITATIONS
SEE PROFILE

Micro-frontends: application of microservices to web front-ends

Andrey Pavlenko[1,2]*, Nursultan Askarbekuly[1], Swati Megha[1], and Manuel Mazzara[1]
[1]Innopolis University, Innopolis, Russia
a.pavlenko@innopolis.ru, {n.askarbekuly,s.megha}@innopolis.university, m.mazzara@innopolis.ru
[2]Sitronics Telecom Solutions (MTS Group), Innopolis, Russia

Abstract

This paper explores the concept of micro-frontends in web-development. Micro-frontends are the logical evolution of architecture for the front-end side of web-applications. It was evolved in conditions of continuous development browser platform that allowed to create rich applications that are easy to distribute and deliver to the user and at the same time provide an experience close to desktop applications. The article begins with an overview of how web-development and the architectural approaches within it evolved, including emergence of micro-services concept in back-end development, and existing researches on micro-frontends. Then, a case study on implementing the concept in an actual project is presented and issues risen during development are discussed. Topics of routing, static assets serving, and repository organization in the projects are discussed based on the experience of development. As a result of the case study, the micro-frontends architecture is found to better suit complex large-scale projects and big teams with experienced developers. The approach increases the complexity of project structure and development. At the moment of publication, the community does not have any ready-to-use solutions that can bootstrap the project and handle the additional complexity.

Keywords: web-development, frontend, micro-frontends, architecture

1 Introduction

Web Applications, as software systems, are traditionally separated into two parts: back-end (server), that is in many cases responsible for data processing, and front-end (client), that is needed to provide a convenient interface for interaction between users and system.

Back-end components could be developed using different architectural approaches, one of them being Microservices, which is most suitable for scalable systems. The idea of the approach is based on the concept of separation of logically independent parts of a system. It is used instead of monolithic architectural styles because those are difficult to scale (and impossible to scale only particular parts of it) and to develop different parts of it concurrently, whereas these issues are solved in microservices.

Speaking about the world of client-side applications, their size, as well as complexity, is also growing (especially in web development). That is why architecture starts to play a more significant role. Front-end community came to the idea of micro-frontends, which is similar to the microservices concept and involves the separation of one big application into smaller parts that leads to the possibility of concurrent development, faster download time, and greater performance.

The main goal of this paper is to examine the micro-frontends approach in the context of single-page applications (SPA). This paper follows a case study of applying microservices architecture in the design and development of a web application's front-end. The case study project is a SPA for online courses aggregator service. Then it proceeds to the discussion around the advantages and disadvantages of the approach based on the case study experiment.

Journal of Internet Services and Information Security (JISIS), volume: 10, number: 2 (May 2020), pp. 49-66
DOI: *10.22667/JISIS.2020.05.31.049*

*Corresponding author: Universitetskaya St, 1, Innopolis, Republic of Tatarstan, 420500, Russia, Tel: +7-987-4100455

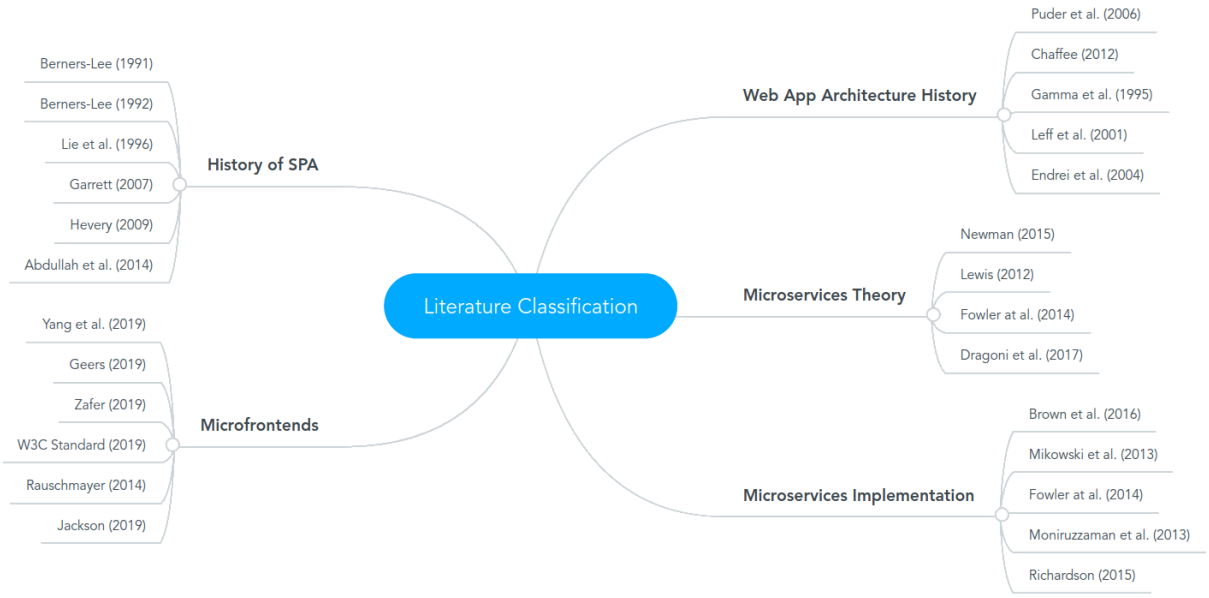


Figure 1: Literature classification and categorization

2 Literature Review

This chapter contains a review of the literature and related work. It starts from a historical reference that covers classical approaches to building web applications that were used in the past decades. Then the microservices topic is covered with related definitions and discussion about its efficiency, positive and negative sides. After that, web frontend is discussed starting with reasons of why industry came to the idea of using Single Page Applications (SPAs) and how the micro-frontends architectural style appeared.

2.1 Web Application Architecture: Historical Overview

The **client-server model** is a type of software architecture model where two parties are involved: service providers, called servers, and service users, called clients [20]. Typically, a server owns some resources that can be retrieved by a client. Web Application is a kind of system that is built using the client-server architecture where all the communication happens through the Internet. There is not any strict definition, however, the concept was defined in Java Servlet Technology Specification as a collection of servlets, HTML pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors and are rooted at a specific path within a web server [5]. Since that time technologies did step forward and now a lot of different variants of implementation of Web Applications are available in addition to Java Servlet.

During a long period applications built on top of the client-server model exploited the so-called **Model-View-Controller** (MVC) pattern [9]. The pattern considers the separation of an application into three parts: a **model** that describes how data is presented inside and how it is processed, the business logic of the application, a **view** that is responsible for displaying the data on a screen, the user interface of the application, and a **controller** that handles interaction between a user and the View and connects these actions to the Model. This pattern was used in the development of Web Applications as well [14].

The popularity of Web Applications had been growing during the first decade of the 21st century, mainly because of interest that appeared from the business side. The penetration of the Internet started to grow and the population used this type of communication channel more. Lots of Web Applications

appeared, but because of some problems with services development, the **Service-Oriented Architecture** was introduced [7]. This approach allowed the IT industry to reduce cost because of reuse of services, increase the speed of development through the combination of existing services in the creation of new solutions, manage the complexity of a system and make simpler integrations between its parts.

2.2 Microservices on Back-end Side

Service-Oriented Architecture played an important role in the development of Software Architecture and the next step in its evolution was an introduction of Microservices architectural style. The term **Microservice** was firstly introduced by j. Lewis [15] in 2012 and later described in detail by M. Fowler and J. Lewis [8]. This topic is still being researched and papers are published every year.

One of the recent researches provided by N. Dragoni et al. [6] define **Microservice** as 'a cohesive, independent process interacting via messages' that means this process does not have anything in common with the other processes and should retrieve resources that are needed to perform its communication with other processes using some messaging protocol. The same paper defines **Microservice Architecture** as 'a distributed application where all its modules are microservices' considering that every part of the application, regardless of either it is a business logic unit or it is an auxiliary technical component, can be developed, deployed and maintained independently and then composed into a system that solves a concrete problem.

The paper mentioned above highlights challenges of a classical monolith type of Software Architecture that includes difficulties in maintainability introduced by high complexity, "dependency hell" problem, high downtime during system's reboot even for small updates, deployment issues connected with hardware environment that can not fit requirements of all the parts of an application, limited scalability and impossibility of usage of different technology stack in different parts of the system. They can be solved by the introduction of a microservices architecture into the project. The small size of a microservice helps to manage complexity problem, independent deployment solves issues of downtime, because small parts are faster to reboot, both deployment, because of possibility to use a machine that best fits a piece of software that is installed, scalability, because there can be deployed more than one instance of each particular process on different machines without affecting other parts of the system, and technologies used, except a communication protocol, because every microservice is developed independently now.

The researchers also touch the organizational part of the development of microservices. They state that the best approach to organize people is to create teams that will be responsible for their services. It also was said in [8] and noted that the team should be involved in the full cycle of life of service, including development and maintenance. They also highlight the importance of deployment automation including continuous integration and continuous deployment meaning that it is hard to manage the process by hands, especially for large-size applications. Questions of the way of inter-process communication and quality assurance are also raised and discussed.

2.3 Practical Side of Microservices

A review of different patterns and approaches of building systems using microservices style can be found in [4]. There are four groups of patterns discussed:

(1) **Modern Web Architecture Patterns**, that cover client-side of a system responsible for the interaction with a user and issue connected with the speed of data delivery to this part of a system;
(2) **Microservices Architecture Patterns**, that cover types of microservices and ways of optimizing their work;

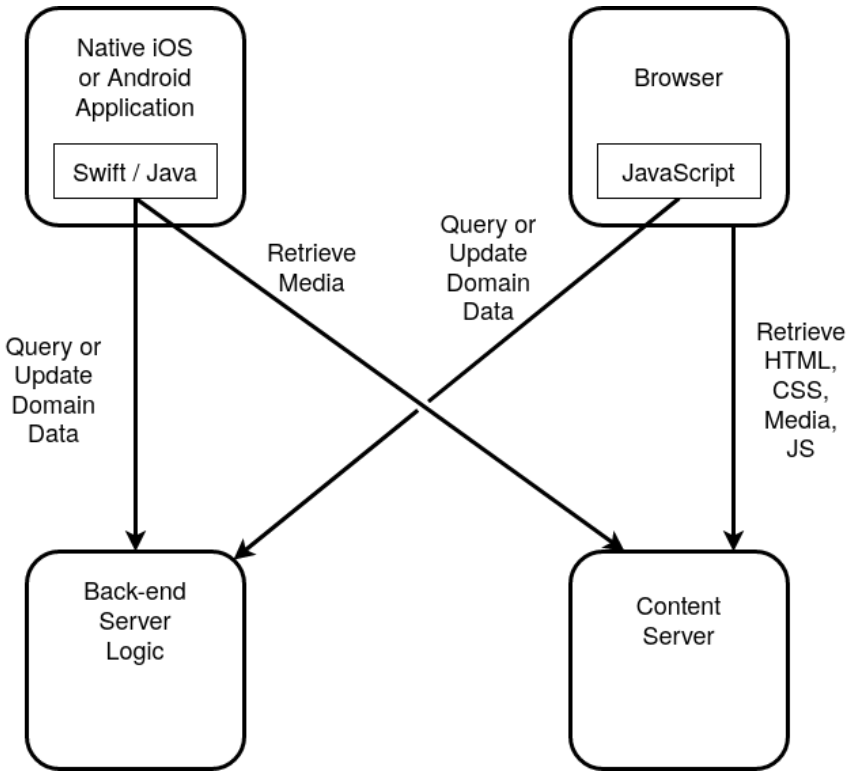


Figure 2: Modern Web Architecture Components

(3) **Scalable Store Patterns**, that should give a possibility to store data in a way that allows convenient access and opportunity to scale it;
(4) **Microservices DevOps Patterns**, that help to develop and maintain the system.

2.3.1 Modern Web Architecture Patterns

The first group includes Modern Web Architecture itself, then Single Page Applications, Native Mobile Applications, and Near Cache.

Modern Web Architecture considers replacement of so-called page-at-a-time dynamic page solutions, when a view of a result of a query is generated by a server and returned once this query was processed, by separately developed and deployed front-end components that can be more intelligent and sophisticated, that is also important for an end-user. This approach also introduces the principle of separation of concerns that means the server now should not bother about how to display the result of the query.

Authors suggest using **Single Page Applications** and **Native Mobile Applications** as a front-end of a system. Single Page Applications are written in JavaScript, work in an environment of web-browsers without reloading page during its use [17]. Native Mobile Applications are written in Objective-C/Swift for iOS and in Java/Kotlin for Android. They are proposed because it is possible to create a more powerful and intuitive GUI that, if designed and developed in the right way, will give a better experience to a user. Figure 2 shows high-level interaction schema of a system. There is also a Content Server placed as a separate block that is responsible for serving static content such as elements of GUI, text entities, style sheets, etc.

Near Cache is aimed to solve the problem of a high amount of requests to the back-end servers. This pattern considers having a lightweight cache inside of a client application that will serve data that is

frequently needed but rarely updated. In this case, data should be downloaded once, stored in the cache, and retrieved from it when is needed.

2.3.2 Microservices Architecture Patterns

Microservices Architecture Patterns group includes Microservice Architecture itself, claiming that a system should be represented as a set of microservices satisfying all microservices principles, Business Microservice, Adapter Microservice, Back-end for Front-end, Result and Page Caches Patterns.

Business Microservice is a type of service that includes business logic. There is a restriction that one such microservice should solve only one business problem and finally come to a single component that can be deployed. Another important point applicable to all types of microservices stated in [8] and highlighted in [4] is that they should not introduce indirect communication through a database.

Adapter Microservice is designed to help to embed an external system into the microservices environment in case if the protocol of communication is not compatible. It maps requests based on the common protocol of communication into a structure that is accepted by the system and vice versa for a response.

Back-end for Front-end is a microservice that serves as a gateway for requests coming from the client-side. The main tasks of this type are **orchestration**, that means performing requests to multiple microservices and combining responses in case if one client's action leads to this case, **translation**, that is a conversion of a response into a form that is more convenient for a client, and **filtering**, that is deleting all the data, that client does not need.

Cache patterns are needed to optimize the way of retrieving data from storage. **Result Cache** is intended to store data that was recently queried and can be requested in the nearest future again. **Page Cache** is used for data that represents listed entities and is passed to the front-end in form of pages, in this case, all queried data can be stored in the cache and if the user requests another page it will be quickly sent. Both the Result and the Page caches usually have a relatively short lifetime (minutes), but it is can be changed by a decision of a developer.

2.3.3 Scalable Store Patterns

Scalable Store Patterns are the Key-Value Store and Document Store [18]. **Key-Value Store** is usually a simple in-memory store that performs search operation with the complexity that is closed to O(1) and can be used for creating caches and storing data that has a relatively small lifetime. In contrast, **Document Store** allows saving data for a longer period. Its main advantage is the possibility to store data in flexible structures, but the main drawback is that it has problems with relational structures.

2.3.4 Microservices DevOps Patterns

And the last but not the least meaningful group of patterns is Microservices DevOps Patterns Group. It includes the **Log Aggregator** pattern that is needed because of the reason that a system consists of dozens of microservices, and it collects all the logs of the system and stores it in one place to make it easier to deal with it. Also, there is the **Correlation ID** pattern that helps to identify requests and responses belonging to one chain in the logs. And the third pattern introduced in the chapter is the **Service Registry** that is aimed to assist with the service discovery [22] process when URLs of microservices are not typed in the code but requested at a time when they are needed allowing to manipulate with deployment model without rebuilding parts of a system after changes in it.

2.4 Static Sites and Single Page Applications

Web Graphic User Interfaces were developing in parallel to the Back-end side of systems. The starting point is introduction of **HTML** by Tim Berners-Lee [2][3]. This language allowed to markup a text, highlighting its different parts and their actual purposes, and link different documents. Then **CSS** was introduced [16], allowing to format and prettify HTML documents, followed by JavaScript, that gives the possibility to manipulate documents and make them dynamic.

HTML documents were static files that were uploaded to the server by an administrator and then retrieved using any web browser. After some time engineers came to the idea of the generation of documents on the fly (page-at-a-time) when it is requested by users. For example, Facebook [1] used PHP programming language to generate pages on their service by user's request.

The next step was to move from server-side document generation to client-side. And later **AJAX** (Asynchronous JavaScript and XML) came into play [10]. It introduced a new level of User Experience and allowed to create dynamic user interfaces for complex systems rater than simple documents. AJAX itself was not a technology, it was a combination of several different technologies that helped to develop web applications.

AJAX included (X)HTML and CSS as standards for presentation of interfaces and Document Object Model (DOM) as a way of handling dynamic aspects of data display. XML and XLST as an instrument for data representation were used along with XMLHttpRequest that helped to retrieve any data from a server using JavaScript. In this new model, the HTML page was downloaded once, and then attached scripts, in response to the user's actions, communicated with a server and updated structure of the page.

Industry liked to use this way of building GUIs, and the popularity of the approach started to grow. A lot of new libraries appeared in the ecosystem of JavaScript, web pages were getting more and more functional and more and more complex. There was introduced **AngularJS** [12] that was aimed to make web-development simpler. Later this library developed into the framework that, along with other popular frameworks, is used for the creation of Single Page Applications. Developers created lots of different things from calculators to games with front-ends written in JavaScript using AJAX concepts and JS libraries and frameworks. And now interfaces are getting even more complex and look like desktop applications (e.g. Google Docs, email web clients, etc.).

2.5 Micro-frontends: New Approach to Building SPAs

Development of Single Page Applications now leads almost to the same problems as the development of servers. Monolith frontend applications in big systems are bloated that become a reason for problems with scalability, deployment, and especially maintenance [24], as it was described in section 2.2. That's why engineers come to the idea of using concepts of microservices in the development of front-ends, calling this approach micro-frontends.

The idea of **micro-frontends** is in the separation of a single front-end application into a combination of multiple small applications that satisfy main principles of microservices: be small, focused on one task, and autonomous [19]. Considering it, M. Geers describes the following concepts that lay in the foundation of Micro-frontends [11]:

(1) **Technology stack independence**. Teams should decide on their technology stack without any impact from the others.
(2) **Code isolation**. Runtime should not be shared and applications should be self-contained.
(3) **Team prefixes**. All the parts, that cannot be isolated, should be prefixed to avoid collisions.
(4) **Native browser API preference**. The default browser's APIs should be preferred for the development of custom ways of inter-application communication.
(5) **Resilience**. The feature should be useful even in the case if JavaScript failed or did not run.

However, this architectural style brings new issues to be solved along with the solution of existing problems [25]:

(1) **Orchestration** - how to deal with loading applications when they are needed.
(2) **Routing** - how to manage routes in the application and decide which app to load.
(3) **Isolation** - how to bound to avoid collisions in the environment.
(4) **Communication** - how to provide applications the way they can communicate with each other.
(5) **Consistency of UI/UX** - how to manage common styles and make sure that UX is consistent.
(6) **Dependency management** - how to avoid one library to be loaded more than once.

2.5.1 Microfrontends: Code Separation and Loading

There are five different approaches to implement Microfrontends in practice [13].

Server-side template composition manages orchestrating applications on the server during HTML page generation. The main drawback of this approach is that it is the old way when the HTML page is generated after each user's request and it vanishes all benefits that were introduced by Single Page Applications.

The **build-time integration** approach considers that each micro-frontend is delivered as a separate JavaScript package and integrated during the final compilation of modules. The main drawback of this approach is that all the micro-frontends should be recompiled if one of them is changed.

Run-time integration via iframes utilizes an iframe tag to organize the isolation of applications. But the main benefit is also the main drawback: it is impossible to share common dependencies across different iframes, so, an application can have a larger size and greater download time. Also, the only way of inter-application communication, in this case, is the API of iframe, which offers less flexibility than the next techniques.

Run-time integration via JavaScript is one of the optimal solutions that consider that each application is built as a separate bundle and is loaded and mounted on the page only if it is needed. In this case, we can pre-load common libraries and styles, that will reduce the final size of the application. Also, each bundle is deployed separately which allows teams to update functionalities independently on others.

Run-time integration via Web Components is a new approach that becomes possible because of new HTML standards issued by W3C [23]. The standards allow us to create custom HTML elements, define their behavior, and load their code dynamically. Isolation, in this case, is also kept by the browser as it was with iframe. But this approach provides a more convenient way of communication between components. The main issue here is that the standards are new and not fully implemented in all browsers yet. Developers can use polyfills, which are small utility libraries that implement new features of JS using old versions of language [21], to make them work, but this will increase the size of content loaded and lead to greater downloading time.

There is also the approach of distribution of applications through web-server when different applications are loaded depending on the route that was requested [24]. But this approach leads to page reloading when a user needs to open another application, thus it is not considered here.

2.5.2 Microfrontends: Common Styles and UI/UX Consistency

The problem of UI/UX consistency can be solved by **shared component libraries** [13] that contains reusable UI elements. Each team combines components from the library to construct the interface that they need. The library can include both simple components, like icons, labels, buttons, and complex components, that contain their logic and UI consisting of a bunch of small elements.

One of the most problematic parts of this type of shared resource is that it API of each component can change after release because of clarification of requirements and appearance of new ones because

there can be issues that were not considered before the start of usage. The other issue is that a team that develops the library can create components that nobody will use.

Another important issue is the way of integration of components in applications. There are three ways available: using Web Components, using frameworks like Angular, React, etc., and using pure JS and CSS. As it was said earlier, Web Components are not well supported by browsers yet. Libraries implemented with the usage of frameworks are limited to work only with these frameworks, so, each component needs to be implemented as many times as the number of different frameworks used, what does transforms independence of technology stack in a drawback rather than a benefit. In the last option, there is a higher possibility of collisions between different parts of applications, so, isolation issues should be considered during development.

3 System Architecture

The Micro-frontends topic discussed in this thesis is explored in the context of developing Education Hub System. The system is the platform that aggregates online courses from different online course providers to serve as a single entry-point and search engine for users to find a course that best fits their needs and wishes.

The system architecture will be presented in this chapter as well as requirements that cover its functional and non-functional parameters and directly affect different aspects of the system's design.

3.1 Overview of System Requirements

The following requirements are presented in form of shall statements equipped with the priority of implementation in the following form:

- (M) - Must - The implementation of the functionality is critical for the system's operation;
- (S) - Should - The implementation of the functionality is not obligatory but highly recommended;
- (C) - Could - The implementation of the functionality is optional.

There are functional and non-functional requirements below in the list:

(1) Front-end of the System
   (a) Shall display a list of courses and their details (M)
   (b) Shall display feedback for courses (M)
   (c) Shall display a list of universities and their details (C)
   (d) Shall display a list of educational events and their details (C)
   (e) Shall display top-5 courses (S)
   (f) Shall perform parametrized and textual search in courses (M), universities (C), events (C)
   (g) Shall allow a user to authenticate using OAuth (C)
   (h) Shall allow a user to authenticate using Login / Password (S)
   (i) Shall allow a user to leave feedback (S)
   (j) Shall allow a user to mark a course, university or event as the favorite (C)
   (k) Shall allow admin to authenticate using login and password (M)
   (l) Shall allow admin to perform CRUD operations on courses, universities, and events (M)

(m) Shall allow admin to moderate feedback (S)

(2) Back-end of the System

(a) Shall provide API for any actions performed by the front-end (priorities like in the previous section)

(b) Shall gather statistics on user requests (S)

(3) Non-functional Requirements

(a) The system shall flexibly store courses (there should be the possibility to add new fields for some courses on the fly) (M)

(b) The system shall be available more than 95% of the time (S)

(c) The system shall reliably store data (data should not be lost) (S)

(d) The system shall perform any search request within less than 5 seconds (S)

(e) The system shall gather and update information about courses automatically (S)

3.1.1 Development View

Figure 3 shows the Development View of the system in the form of components diagram.

First of all, the system architecture is built on top of the concept of multilayered architecture. Particularly, this is three-tier architecture with classical tiers: presentation, business logic, and data. On the diagram, each of them is highlighted by its color.

The presentation tier is highlighted using red color and mostly presented by components with type FrontService. This tier consists of four components that are micro-frontends, i.e. independent small front-end applications that compose into a bigger one. The micro-frontends aim to perform requests to the business logic tier and show its responses in a user-friendly manner. All the micro-frontends communicate with the business-logic tier using a single entry point - API Gateway.

API Gateway in our case is part of the business logic tier. It is the implementation of concepts of Back-end for Front-end and Service Registry[4]. The only goal of the component is to provide a single entry point for front-end, i.e. presentation tier, take its requests, and redirect them to corresponding microservices on the back-end side, i.e. business logic tier. However, this part of the system is the potential bottleneck, because all the requests will pass through it. That means the implementation should be reliable and efficient enough to tolerate high load.

All other components of the business logic tear are highlighted using green color and have type Service. They will handle all the operations on data transformation and processing, such as storing it into DB, performing search requests, checking user rights, etc. Particular functions of the Services will be described a bit later

Data-tier is highlighted using blue color and its only purpose to serve requests to store and retrieve data from long-term memory. Particularly, there are three databases, to serve corresponding services, inside of one database server.

Talking about functionality, each service exposes several interfaces that are delegated to API Gateway, as it was said earlier.

The ElasticSearch service is a special service that is an instance of ElasticSearch. The main purpose of the component is to perform a fast search on arrays of data. It is highlighted using both green and blue colors because of this piece of software stores all the data by itself. The interface ISearchProcessing represents all the API methods provided by ElasticSearch for developers and is used mainly by SearchService.

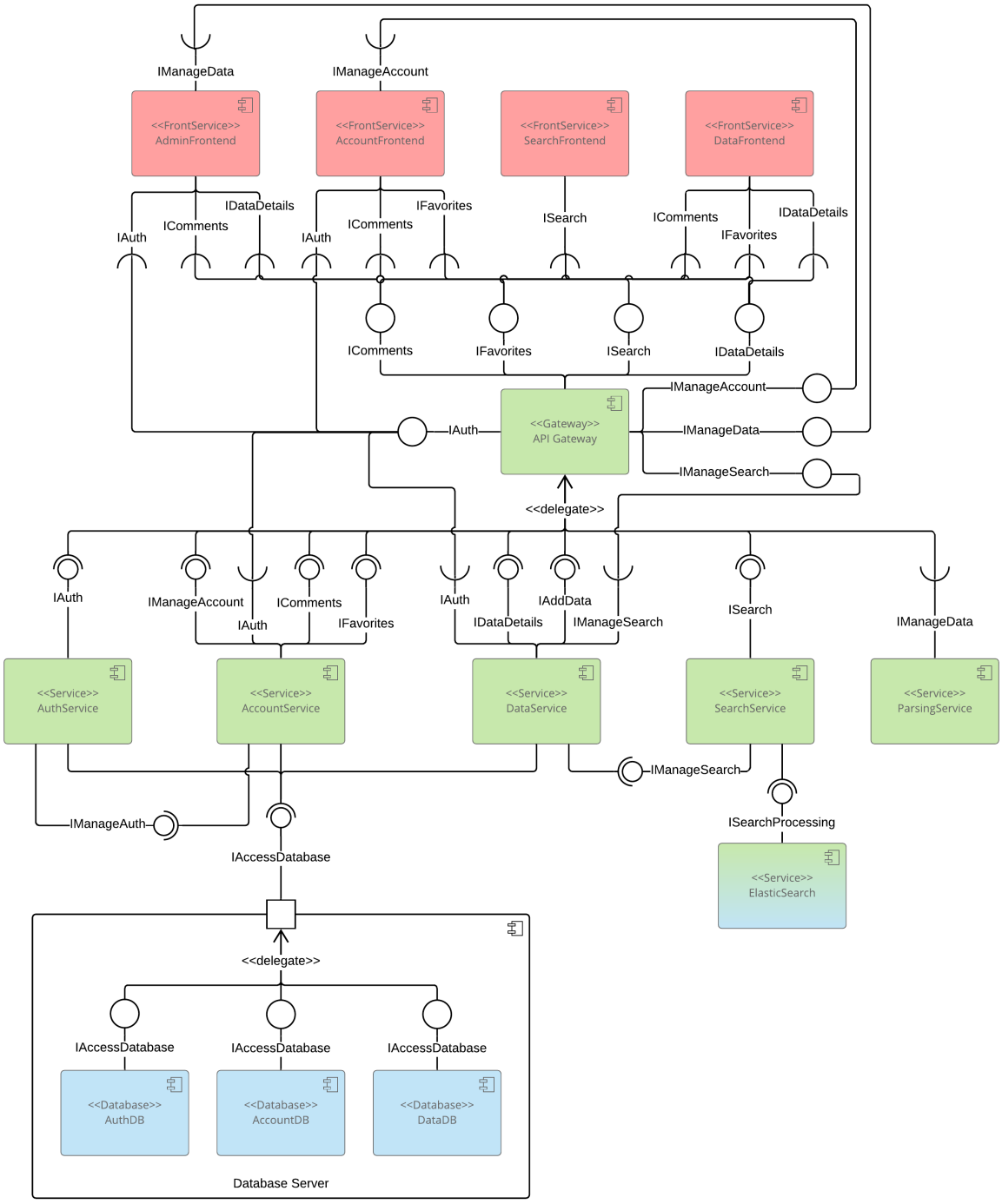


Figure 3: Development View - Components Diagram

The SearchService encapsulates the API of ElasticSearch. Its main purposes are to transform requests from front-end into the ones suitable for ElasticSearch, preprocess its responses by deleting redundant and technical information, and manage data that is put into it. It exposes two interfaces:

- **ISearch** - the interface provides methods for performing the textual and parametric search in arrays of data;
- **IManageSearch** - the interface provides methods that help to add, update and delete entities in ElasticSearch by other services, particularly, by DataService. It is available only for internal usage and not accessible through API Gateway.

The DataService processes data incoming to and outcoming from the DataDB, that is intended to store Courses, Universities, and Events. It exposes two interfaces:

- **IManageData** - the interface provides methods that help to add, update, and delete entities of types Course, University, and Event to the system.
- **IDataDetails** - the interface provides ways of accessing data in the system.

The service also manages all the data in search using the IManageSearch interface provided by SearchService in case if methods of IManageData are invoked. Both the interfaces are available through API Gateway, however, the IManageSearch interface requires authorization to perform any actions.

The AccountService handles all the actions on data of users of the system, such as accounts' data, lists of favorite Courses, Universities and Events, and comments to them. It exposes three interfaces:

- **IManageAccount** - the interface provides methods that provide Create, Read, Update, and Delete (CRUD) operations on the user's profile;
- **IComments** - the interface provides methods for searching in user's feedback and performing CRUD operations on it;
- **IFavorites** - the interface provides methods that help to put entities into a list of favorites and remove them from it.

The methods are accessible through API Gateway, however, they all require authorization to perform any operation on data because they all work with the data that should not be available publicly. Also, the service uses the IManageAuth interface to control the user's credentials.

The AuthService handles issues related to access control to the system. It exposes two interfaces:

- **IAuth** - the interface contains a method that checks a user's credentials and issues authorization tokens and method that checks token and rights of the user;
- **IManageAuth** - the interface helps to perform CRUD operations on the credentials of users.

Methods of the IAuth are accessible through API Gateway and is used by presentation tier, particularly, token issuing and token checking, and by business logic tier, as well, particularly, token and rights checking. Methods of IManageAuth available only internally in the business logic tier and are used primarily by AccountService.

There is also a special service that does not expose any interfaces and only uses IManageData that is called ParsingService. In the future, this service can be used for automatic data collection from external resources, however, it is not going to be implemented in the current version.

It is also can be noticed that the diagram can be viewed not only horizontally, in terms of different tiers, but vertically well, in terms of functional units:

- DataFrontend, DataService, ParsingService, and DataDB refer to DataService from the Logical View;
- SearchFrontend, SearchService, and ElasticSearch refer to SearchService from Logical View;
- AccountFrontend, AccountService, AuthService, AccoundDB, and AuthDB refer to Account service from Logical View.

API Gateway is a supporting service that can be both included and not included in each of these groups because it only provides a convenient way of accessing methods for developers.

AdminFrontend can not be included in any of these groups because of the reason that it communicates to all of the parts of the system. Its main purpose is to provide a readable UI that helps to manage the system manually.

4 Implementation Stack of Front-end

Due to the dynamic nature of the front-end application, it was decided to use React as the rendering library. There are several reasons for the decision:

(1) The library itself is easy and its main purposes are to manage rendering and simple state of the front-end application;
(2) Several boilerplates exist that makes the start of development simpler comparing to set up from scratch;
(3) TypeScript support is also available in the boilerplates like create-react-app;
(4) The library is based on the component-oriented approach to the development of applications what, in theory, increase the level of reusability of the codebase and reduces duplications;
(5) There are a lot of utility libraries developed for React exclusively that helps to solve routine problems within an acceptable time;
(6) Due to popularity and good support of the community, there are lots of books, articles, discussions, and other materials that can help to solve any problems rose during development.

React Router library was used for creating seamless client-side transitions between pages in the browser for the micro-front-ends that have internal routing. This is the library created especially for React. It directly manipulates the history state of the browser window to achieve the effect of changing URL when the page is not reloaded.

EmotionJS manages CSS styles during the application's runtime. It helps to avoid unnecessary problems with additional assets management in a micro-frontends application, that could rise in case if all CSS code is bundled in .css files.

Webpack utility with a set of plugins and loaders is used for processing and bundling final builds of micro-frontends. It is configured to pass all the .ts files through the TypeScript compiler and then pack all the results into one single file that is then loaded by the user's browser.

SingleSPA library was used to manage to switch between different micro-frontends. This library manages actions connected with registration of micro-frontends, managing routes, mounting and unmounting micro-frontend applications into DOM-tree of the browser.

5 Micro-frontends Concepts Analysis

5.1 Different Approaches

Generally, there are three approaches to building GUI in web-applications now: Server-Rendered Pages, Single Page Applications, and Single Page Applications with Chunk Splitting.

Server-Rendered Pages approach proposes the usage of template engines, and markups are usually embedded in controllers on the back-end side of the application. GUI in this case is static, however, there can be dynamic parts injected and loaded as separate static JavaScript files. Back-end generates pages when they are requested by a user and delivers them with corresponding resources. The development of the front-end part is not dedicated to a separate team, so, the front-end does not have an impact on the possibility of simultaneous work by different teams. The setup of pages generation is simple unless developers try to tune the template engine. However, the complexity of maintenance is not an obvious question here, because templates can be distributed between different parts of the system, and the right template should be found before it can be fixed.

Single Page Applications are front-ends developed with JavaScript and delivered to a user as a single bundled .js file. They are dynamic and work without page reloads. It is easy to set up because of boilerplates' availability. However, it is quite hard to distribute work across different teams and complex to maintain an application if it has a large size.

Single Page Applications with Chunk Splitting are the same, as the previous one. The only difference here is that builder is configured in the way that its output is a set of files that are loaded by request. The files contain a subset of the functionality of applications. This option helps to speed up startup time and improve user experience.

And finally, the subject of the research, Micro-frontends architecture is an approach that is built on top of Single Page Applications. It is implemented correctly, different frameworks can be used for different parts of an application. The parts are loaded by request and even can have chunk splitting enabled. Micro-frontends allow different teams to work simultaneously on different parts without major problems with synchronization. They are hard to set up and maintain because most of the work should be done manually, however, when frameworks and bootstrap kits appear, it should become easier because they will take care of repeating operations and boilerplate code.

	Server-Rendered Pages	**Single Page Application**	**SPA with Chunk Splitting**	**Micro-frontends**
Static or dynamic GUI	Typically static with dynamic parts	Typically dynamic	Typically dynamic	Typically dynamic
Framework restrictions	Without framework	Single framework	Single framework	Any number of frameworks
Delivery to the client	Loads one page at a time	Loads fully, in the beginning	Loads partially, on request	Loads partially, on request
Simultaneous work	Possible	Usually, impossible	Usually, impossible	Possible
Setup complexity	Simple	Simple	Normal	Complex
Maintenance complexity	Normal	Normal/Complex	Normal/Complex	Normal/Complex

Table 1: Comparison of Approaches to Building Front-end

5.2 Routing Issues

Routing is one of the main concerns in Single Page Applications. The reason we want to change the address is that often they include lots of different features that should be split into different logical units, and state of the application should be saved in case if a user needs to reload the page for some reason

or to share the link to particular feature with other people. However, the size of an application usually much more than the size of a page rendered on the server, and taking into consideration the growth of usage of mobile devices and poor coverage of high-speed mobile internet, any reload of the page should be avoided.

At the beginning of the development of such applications, Location object was used for these purposes that are accessible through ‘window.location‘ property in browser platform JavaScript. The goal of the object is to represent detailed information about the current location that is opened in the browser. There is ’hash’ property in the object that creates part of a route that does not reload the page when it is changed.

When the HTML5 standard appeared, a new way of dealing with routing was provided to the developers. History object that is accessible through ‘window.history‘ property gives access to a new API that helps to manage seamless transactions across history, back and forth.

It may seem like there are no problems with routing at all in Single Page Applications development. However, with the micro-frontends concept being introduced, the issue of dealing with routing inside and across different micro-frontends appears. As micro-frontends should be independent and isolated by nature, they should be developed without prior knowledge of how they will be deployed, including the absence of predefined routes for micro-frontends. Additionally, the issue of management of the current active micro-frontend inside of the page should be also solved, and it can be done through routing, as well.

The possible solution is to use custom system-wide routing that will manage currently loaded micro-frontend and be shared across them. And this part of the system can be based on History API provided by browsers, taking into consideration the context of the current micro-frontend and their configuration. Ready-to-use libraries do not exist for the moment. SingleSPA library is used to deal with the management of currently loaded applications in the project that is described in the actual paper, however, it relies on user-defined activation functions and does not provide any routing solution.

5.3 Static Assets Issues

Serving static assets is one of the most important things in SPA deployment because images, styles, JavaScript files, and other resources are very important to make the application work and look as it was intended to be. The current model of working with resources suppose that developers know how the system is deployed because all the work with them from JavaScript and HTML is done through URL.

This is an issue because of the independence of the applications inside of the system and independent deployment that consider that deployment schema should be changeable without any consequences. It makes the process of work with static assets complicated, so, developers need to choose either to use pre-defined deployment schema where all the files are placed in exact folders or to create a system-wide shared function that will return base address or address of a resource depending on the context and state of the system.

5.4 Project Organization

The structure of the project highly affects the speed and efficiency of development. Two of the parts tightly connected to the structure are versioning and sharing code across developers. Small or monolith projects can be stored in one repository inside of Version Control System (VCS), however big modular systems require another approach to be applied to the development process.

There are three ways for micro-services (and micro-frontends) projects: Mono-repository, Multi-repository, Multi-repository with git-repo tool.

	Mono-repository	Multi-repository	Multi-repository with git-repo tool
VCS	Any	Any	Git
Additional tools required	No	No	git-repo
Package registry required	No	Yes[1]	No
Partial codebase downloading	Impossible	Possible	Possible
Setup complexity	Simple	Complex	Complex
Checkout speed	Slow	Fast	Fast

Table 2: Comparison of Different Repository Organization Structures

The simplest way is to choose mono-repository and put all the modules of the project in one single repo somehow structured, e.g. one module per folder. The main problems of this variant are that all the codebase should be downloaded before development can be started, even if you do not need to have the code of all the micro-frontends, branch checkouts and synchronizations can take a lot of time in case if the project is intensively developed by a big group of developers.

The second possible alternative is to have separate repositories for different micro-frontends, and separate repositories for common parts. In this case, all the parts of the project can be published as packages in the private package registry and linked to the project using it. But that leads to the increased complexity of developer tools setup and maintenance. Also, additional actions are needed to update packages in the registry and all of the repositories. Workflow can be configured without package registry, but it makes setup process even more complex and every new developer will need to link folders on their local machines before the first run ([1]).

The third variant is to have multiple repositories as it was mentioned above, but manage them using the git-repo tool provided by Google. The tool takes a configuration file that describes what set of repositories is needed to be downloaded and in what way they should be structured, and performs all these actions. In this case amount of repositories, that are needed to be download from a remote host, is configurable, and only necessary part of the codebase can be cloned, comparing to mono-repository, and the problem of slow checkouts is also removed because you need to synchronize only repositories that you currently use. At the same time, you do not need to have additional servers for the package registry however complexity of the environment setup is still higher.

So, there is a trade-off between the complexity of environment setup and speed of work of the Version Control System, and developers need to decide what is acceptable in their concrete case.

6 Conclusion

In its essence, the concept of micro-frontends architecture is the micro-services approach adopted to front-end development, and is a new alternative way for the development of modular front-end applications. This architectural style is a response of the community to the increasing complexity of web-applications. In general, it best fits if it is obvious that the system will have a lot of business logic on the front-end side and will be developed by a big team of developers in the long run. In this case, all the complexity, that is added by instruments that help to support the development process and connect all the micro-frontends, will be compensated by decreased coupling, i.e. dependency of modules on each other, and management efforts, because development can be easily split into separate teams.

However, as became evident with case study project, for small projects or projects with a small number of developers, this kind of architecture does not fit well because most of the efforts of developers are spent on support of architecture instead of feature development and overall time of development is increased. Moreover, increased complexity cannot be managed by a small group of people effectively, which leads to an overhead, boilerplate code, and an increased probability of having additional bugs. As was described in the previous section, the issue of the long-lasting project setup arose during the initial development stage of the case-study project. It consisted of taking the decision on the appropriate structure of the project and way of delivery of static assets to the user's browser, and took considerably more time than expected.

Time was also spent to prepare tools that helped to manage applications delivery, their mounting and unmounting from DOM-tree, and few other similar issues. This leads to another important point: the community still does not have fully ready-to-use solutions that include all the necessary functionality to support the micro-frontends architecture. That means the team should have experienced developer who can setup project from scratch and resolve issues appearing before the start and during the development cycle, e.g. setup of Continuous Integration and Continuous Delivery pipeline that will support chosen architecture, prepare development and deployment processes considering issues described earlier, choose tools for implementation, and configure these tools. Of course, that takes more time than the preparation of a project without micro-frontends, so the release of features for users is delayed.

Some libraries solve part of the issues separately and help to build an application based on micro-frontends architecture. However, many of them are not frameworks that provide such a tool-set that developers can take and start building features for end-users. For example, the SingleSPA library was used in the case study project, but it is only the library that manages the appearance of micro-applications in browser's DOM and does not solve such problems as loading these applications from the server, providing tools for communication between them, and handling routing in a browser. All the aforementioned factors lead up to the following conclusion: for the micro-frontends to be a feasible option for small projects and teams there has to be a comprehensive tool and library support which will simplify and speed up the architecture's implementation process.

References

[1] H. M. Abdullah and A. M. Zeki. Frontend and backend web technologies in social networking sites: Facebook as an example. In *Proc. of the 2014 3rd International Conference on Advanced Computer Science Applications and Technologies (ACSAT'14), Amman, Jordan*, pages 85–89. IEEE, December 2014.

[2] T. Berners-Lee. www-talk from september to october 1991. `http://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html` [Online; accessed on October 7, 2019], October 1991.

[3] T. Berners-Lee. Tags used in html. `https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html` [Online; accessed on October 7, 2019], 1992.

[4] K. Brown and B. Woolf. Implementation patterns for microservices architectures. In *Proc. of the 23rd Conference on Pattern Language of Programs (PLoP'16), Monticello, Illinois, USA*, pages 1–35. The Hillside Group, October 2016.

[5] A. Chaffee. What is a web application (or "webapp")? `http://www.jguru.com/faq/view.jsp?EID=129328` [Online; accessed on October 22, 2019], May 2012.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: yesterday, today, and tomorrow.* Springer, Cham, September 2017.

[7] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. *IBM: Patterns: service-oriented architecture and web services*. IBM Redbooks, April 2004.

[8] M. Fowler and J. Lewis. Microservices. `https://martinfowler.com/articles/microservices.html` [Online; accessed on October 5, 2019], March 2014.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1995.

[10] J. J. Garrett. Ajax: A new approach to web applications. `https://www.semanticscholar.org/paper/Ajax%3A-A-New-Approach-to-Web-Applications-Garrett/c440ae765ff19ddd3deda24a92ac39cef9570f1e` [Online; accessed on May 20, 2020], 2007.

[11] M. Geers. Micro frontends - extending the microservice idea to frontend development. `https://micro-frontends.org/` [Online; accessed on October 13, 2019], 2019.

[12] M. Hevery. Hello world, ¡angular/¿ is here. `http://misko.hevery.com/2009/09/28/hello-world-angular-is-here/` [Online; accessed on October 7, 2019], September 2009.

[13] C. Jackson. Micro frontends. `https://martinfowler.com/articles/micro-frontends.html` [Online; accessed on October 13, 2019], June 2019.

[14] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proc. of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'01), Seattle, Washington, USA*, pages 118–127. IEEE, February 2001.

[15] J. Lewis. Micro services - java, the unix way. `http://2012.33degree.org/talk/show/67` [Online; accessed on October 5, 2019], 2012.

[16] H. W. Lie and B. Bos. Cascading style sheets, level 1. `https://www.w3.org/TR/CSS1/` [Online ; accessed on October 7, 2019], December 1996.

[17] M. S. Mikowski and J. C. Powell. *Single Page Web Applications*. Shelter Island: Manning, September 2013.

[18] A. B. M. Moniruzzaman and S. A. Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191:1–14, June 2013.

[19] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition, February 2015.

[20] A. Puder, K. Römer, and F. Pilhofer. *Distributed Systems Architecture*. New York: Elsevier, January 2006.

[21] A. Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers*. O'Reilly Media, 1st edition, February 2014.

[22] C. Richardson. Service discovery in a microservices architecture. `https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture` [Online; accessed on October 7, 2019], October 2015.

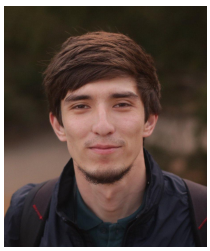
[23] w3c. w3c/webcomponents. `https://github.com/w3c/webcomponents` [Online; accessed on October 13, 2019], 2019.

[24] C. Yang, C. Liu, and Z. Su. Research and application of micro frontends. *IOP Conference Series: Materials Science and Engineering*, 490(1):1–8, April 2019.

[25] Ö. Zafer. Understanding micro frontends. `https://hackernoon.com/understanding-micro-frontends-b1c11585a297` [Online; accessed on October 13, 2019], January 2019.

Author Biography

Andrey Pavlenko finishes his B.S. in Computer Science at Innopolis University. His area of interest and specialization is practical software engineering and related areas. Currently he is a software engineer at Sitronics Telecom Solutions (MTS Group) and works mainly on front-end development of web-services.



Nursultan Askarbekuly is a junior researcher at Innopolis University (Russia). Nursultan is a software engineer with 5 years of experience as a developer and project manager. His research interest includes Software Design, User Experience and Requirements Engineering.

Swati Megha is a junior researcher at Innopolis University (Russia). Swati holds a Masters Degree in Information technology from the Innopolis University. She has 4 years of work experience in IT industry as Backend developer. Her research interest is focused around Software engineering and Block chain technology.

Manuel Mazzara is a professor of Computer Science at Innopolis University (Russia) with a research background in software engineering, service-oriented architectures and programming, concurrency theory, formal methods and software verification. Manuel received a PhD in computing science from the University of Bologna and cooperated with European and US industry, plus governmental and inter governmental organizations such as the United Nations, always at the edge between science and software production. The work conducted by Manuel and his team in recent years focuses on the development of theories, methods, tools and programs covering the two major aspects of software engineering: the process side, describing how we develop software, and the product side, describing the results of this process.