CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering
Department of Computer Science

Asynchronous communication using WebSockets

Bachelor thesis

Vít Šesták

Supervisors: Ing. Martin Komárek, CTU
Jason Hibbeler, Ph.D., UVM

Study program: Software Engineering and Technology

Prague, 2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šesták** Jméno: **Vít** Osobní číslo: **492208**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Asynchronní komunikace pomocí WebSockets

Název bakalářské práce anglicky:

Asynchronous Communication Using WebSockets

Pokyny pro vypracování:

1. Study problematic of asynchronous client – server communication using Websocket protocol
2. Create proof of concept using Websocket protocol
3. Design and implement asynchronous client – server communication into existing product CodeNOW, value stream delivery platform.
4. Use iterative development approach.
5. Test the effectiveness of the implemented solution by monitoring the volumes of transferred data.

Seznam doporučené literatury:

FAANG, Crack. Ajax Polling vs Long-Polling vs WebSockets vs Server-Sent Events. Medium.com. 2021. Dostupné také z: https://medium.com/geekculture/ajax-polling-vs-long-polling-vs-websockets-vs-server-sent-events-e0d65033c9ba
MOZILLA FOUNDATION. The WebSocket API (WebSockets). MDN. 2021. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
DE TURCKHEIM, GRÉGOIRE a ROWENA JONES. IoT Hub: What Use Case for WebSockets?. In: Scaleway [online]. [cit. 2022-01-18]. Dostupné z: https://blog.scaleway.com/iot-hub-what-use-case-for-websockets/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek katedra informační bezpečnosti FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **07.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Declaration

"I declare that I have prepared the submitted work independently and that I have indicated all information sources used in accordance with Methodological Guideline on ethical principles in the preparation of university theses."

Prague, May 10, 2022

Acknowledgements

First of all, I would like to thank my family who have supported me all the time, I appreciate it very much.

I would also like to thank both my supervisors Ing. Martin Komárek from CTU and Jason Hibbeler, Ph.D. from the University of Vermont for their efforts to help me in writing this thesis. Special thanks also go to the CodeNOW development team for their assistance.

Abstract

The aim of this thesis is to study the problem of WebSocket communication and alternative solutions and to use the knowledge gained to improve the performance of the CodeNOW application (a web-based software platform as a service) in terms of network communication efficiency. The optimization is achieved by transforming the synchronous communication, realized by the polling technique between the frontend and backend components, into asynchronous communication by using the WebSocket protocol. This thesis deals with the analysis of the problem, the design and implementation of several solution approaches, and the evaluation of the results obtained. Although the assignment concerns CodeNOW as the environment for developing and testing the solution, the principles discussed in this work are applicable to a wide range of web applications.

Keywords

WebSocket, Polling, Server-Sent Events, Web application, Asynchronous communication, Network efficiency, React

Abstrakt

Cílem této práce je prostudovat problém WebSocket komunikace a alternativních řešení. Dále využít získané poznatky ke zlepšení výkonu softwarové továrny CodeNOW z hlediska efektivity síťové komunikace. Optimalizace je dosaženo transformací synchronní komunikace, realizované technikou pollingu mezi webovým klientem a serverem, na asynchronní komunikaci pomocí protokolu WebSocket. Tato práce se zabývá analýzou problému, návrhem a implementací několika přístupů k řešení a vyhodnocením získaných výsledků. Přestože zadání využívá aplikaci CodeNOW jako prostředí pro vývoj a testování řešení, principy diskutované v této práci jsou aplikovatelné na širokou škálu webových aplikací.

Klíčová slova

WebSocket, Polling, Server-Sent Events, Webová aplikace, Asynchronní komunikace, Efektivita síťového provozu, React

Content

1. Introduction

The software factory CodeNOW is a software platform as a service, enabling businesses to deliver customer value faster. It makes developing cloud-native applications with microservices-based architectures possible for coders of all skill levels by automating out operational complexities. [1]

CodeNOW is made up of many components – a frontend[1] component and several backend[2] components. As a result of user activity, there are often changes in the state of the application that have to be reflected in the user interface, as it is necessary to display current data to connected users. It is therefore desirable for the frontend to work with the most up-to-date data possible. This has so far been handled using the HTTP polling technique, which consists of periodically sampling the server state. [2] [3]

As CodeNOW grows in size and is used by more and more users, it is increasingly necessary to address the efficiency of communication between the individual components of the application to avoid overloading the server.

The important note is, that although this work uses CodeNOW as an environment for deploying and testing the solution, principles discussed in this work can be applied to a wide range of applications, which deal with the problem mentioned above.

This thesis deals with the analysis of the problem, the design and implementation of the solution, and the evaluation of the observed results.

1.1 The aim of the work

The aim of this work is to improve the performance of the CodeNOW application in terms of network communication efficiency using asynchronous techniques. [3]

In principle, this problem can be solved in two ways that are not mutually exclusive: by reducing the volume of data transmitted or by reducing the frequency of data transmission.

Because of the polling technique, data is transmitted at regular intervals and some data, typically the part of the data that has not changed between intervals, may be sent unnecessarily. This offers room for improvement.

The result of this work should therefore be to minimize polling in CodeNOW, i.e., to minimize periodic sampling of backend state using HTTP requests. [2]

The optimization is achieved by transforming the synchronous communication between the frontend and backend components into asynchronous communication. [4] For this purpose, communication over the WebSocket protocol is used. [5]

[1] Frontend refers to a program's or website's user interface. It is commonly known as a presentation layer.
[2] Backend, also referred to as the "server-side", is the part of a computer application that users do not see.

1.2 Work methodology

As mentioned in the assignment, an iterative approach is used for developing the solution. This means that every two weeks a part of the solution, either a part of the implementation, analysis or design, is delivered. [6] Delivered pieces are then discussed with the supervisor from either CTU or UVM, and with the CodeNOW development team. Also, during the consultations the next steps are discussed.

This regular communication is necessary for the iterative approach which consists of planning, analysis, design, implementation, and testing and review activities that are shown in Fig. 1 below.

Thanks to the possibility of using CodeNOW, the two-week outcomes can be tested regularly. This includes the build, deployment and test processes. All of that can be done with ease, because CodeNOW contains pipelines to build feature branches, and offers a separate deployment environment.

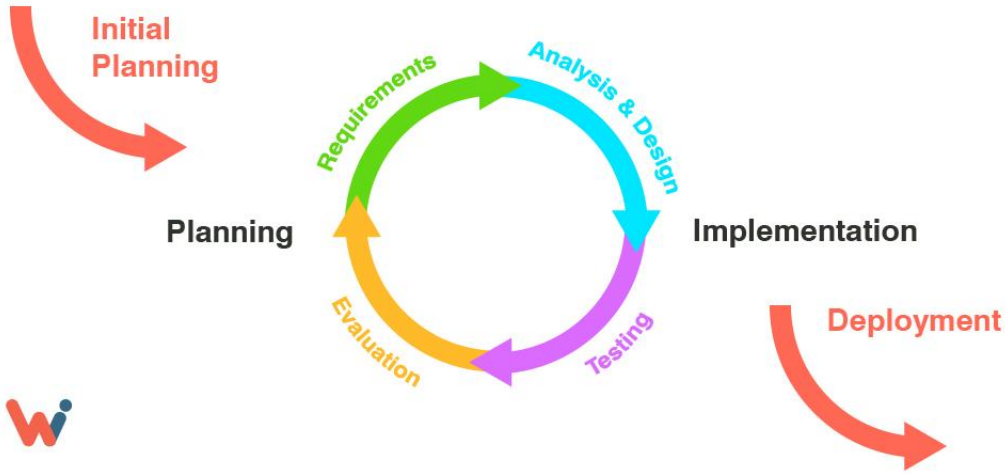

Fig. 1 – Iterative design approach [7]

2 Problem of asynchronous communication

Asynchronous communication is a method of exchanging messages between two or more parties in which each party receives and processes messages whenever it is convenient or possible to do so, rather than doing so immediately upon receipt. [8]

This means that the sender of a message does not always expect an immediate response, but can do something else and wait to be notified that a response has been received.

There are several ways and techniques which we can use to achieve this behavior, one of which is the use of the WebSocket protocol that is described in detail in the subchapters below.

2.1 WebSocket protocol

WebSocket is a computer communication protocol that provides a full-duplex (two-way) communication channel over a single TCP connection. [9] In particular, the mentioned feature of two-way communication is crucial for this work, as it is necessary for the server to be able to send messages without doing so at the client's request, which is the case of the traditional request-response HTTP communication. [3]

2.1.1 Handshake

To establish a WebSocket connection, it is necessary to perform a handshake, which uses the HTTP protocol. In order to do that, the client sends an HTTP request to the server. In this case, the request contains among others two important headers – connection and upgrade. The connection header is set to *upgrade* and the upgrade header is set to *websocket*. This lets the server know that we want to switch protocols, and what specific protocol we want to use, in this case the WebSocket protocol. The request object also defines other parameters in the header, such as Host, which is the domain name[3] of the server, Origin, which is the domain name of the client, or Sec-WebSocket-Key. [Fig. 2] [10]

[3] Domain name is the designation of an identifier of a computer or computer network that is connected to the Internet

2. Problem of asynchronous communication

Fig. 2 – Request header

If the server is implemented to be able to communicate using the WebSocket protocol, it sends an HTTP response to the client confirming the upgrade. You can notice that also the response contains the connection and upgrade headers which confirm that the server is switching from HTTP protocol to WebSocket protocol. [Fig. 3]

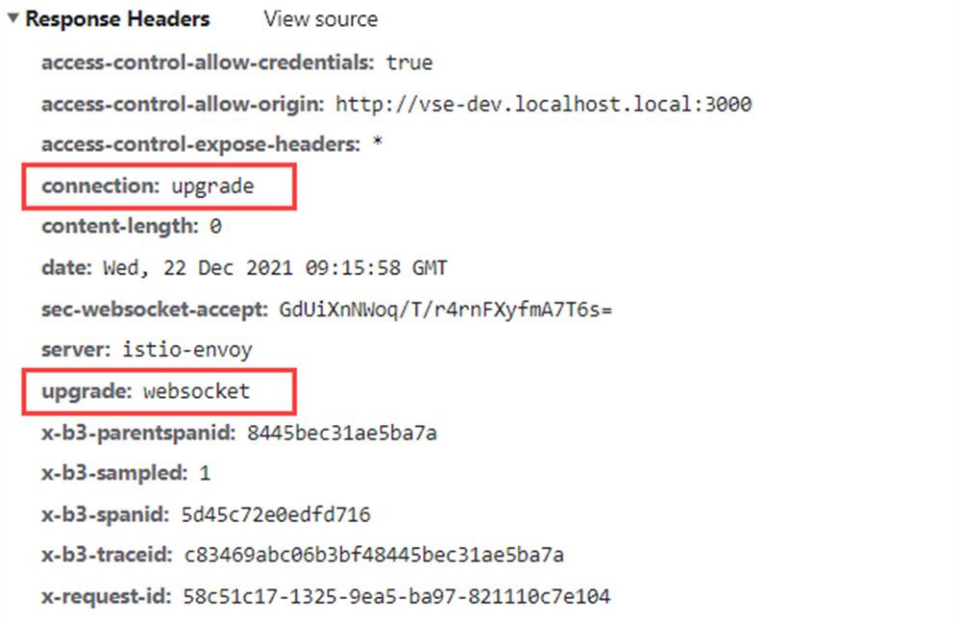

Fig. 3 – Response header

If no error occurs, a single WebSocket connection is established at the beginning of the communication, which is maintained while messages are sent from the client to the server

and vice versa. It is worth saying that the WebSocket connection bypasses the CORS[4] policy, so any validation, such as checking the client's address, must be implemented on the server. [11]

The WebSocket specification defines *ws* and *wss* as two new Uniform Resource Identifier (URI) schemes that are used for unencrypted (*ws*) or encrypted (*wss*) connections. [Fig. 4]

It is generally recommended to use solely the *wss* scheme, specifically the scheme providing encrypted connections. The unencrypted schema might be used primarily for local development.

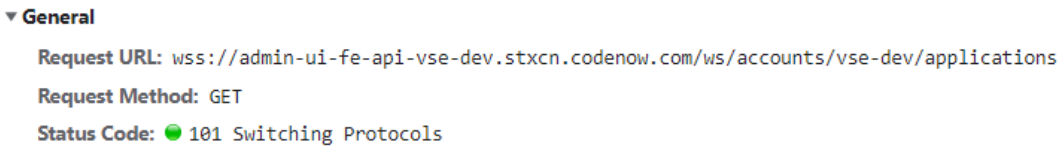

Fig. 4 – Connection mode

2.1.2 API

The WebSocket object provides an API[5] for creating and managing a WebSocket connection to a server.

The constructor contains a mandatory *url* attribute and an optional *protocols* attribute specifying one or more used sub-protocols. By default, the *protocols* attribute is an empty field and in many cases does not need to be set any value. [5]

- *WebSocket (url, [protocols])*

There are two methods available on the object mentioned above:

- *close ()* – to close connection
- *send ()* – to prepare and send data

In addition, the following events are available on the object:

- *open* – triggered if a WebSocket connection is opened
- *message* – triggered if any data is received
- *close* – triggered if the connection is terminated by one of the parties
- *error* – triggered if the connection is terminated because of an error

[4] Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.
[5] Application Programming Interface (API) is a software intermediary that allows two applications to talk to each other.

2.1.3 Comparison with HTTP protocol

Standard HTTP communication is based on the request-response principle. The connection exists during this one interaction and terminates when a response is received from the server.

However, the situation is different for the WebSocket protocol. First, a handshake is performed using the HTTP protocol. Then, when communication via the WebSocket protocol is already arranged, the connection lasts until one of the parties terminates it. Note that in case of WebSocket communication, the server is not bound to any request from the client and can send several messages in a row to the client. The differences between the HTTP and WebSocket communication are shown in Fig. 5.

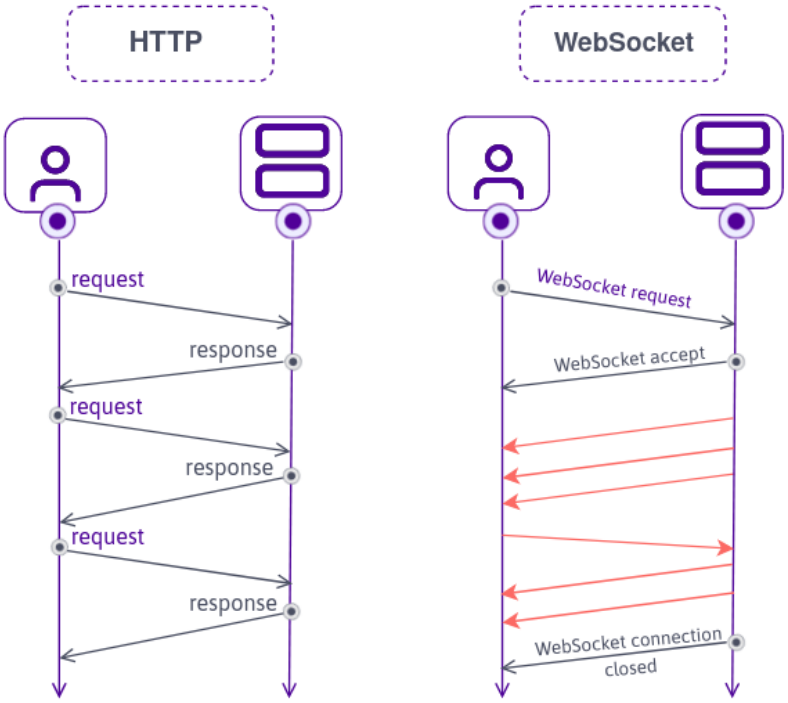

Fig. 5 – Comparison of HTTP and WebSocket communication [12]

2.1.4 Browser support

In order to use a specific functionality in our frontend application, we need to make sure that it is sufficiently supported in the browsers that we want to target.

The WebSocket protocol is currently supported in most major browsers such as Google Chrome, Microsoft Edge, Firefox, Safari or Opera, as can be seen in Fig. 6. [13]

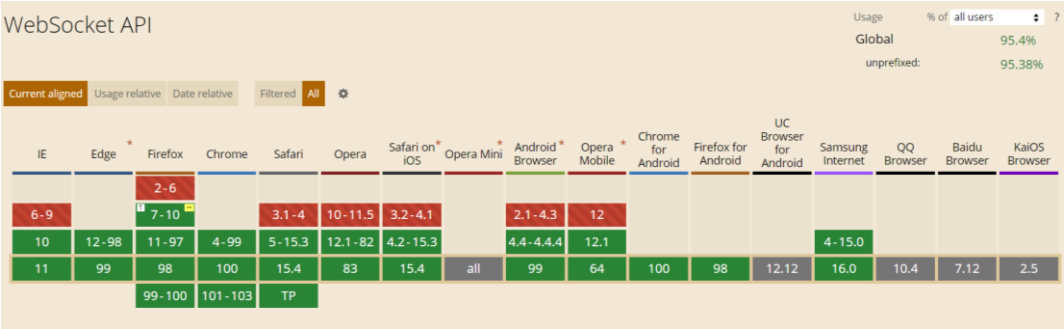

Fig. 6 – WebSocket browser support [13]

2.2 Alternative approaches

Asynchronous behavior can be achieved in a few different ways and there is no optimal solution for all cases.

This section therefore maps the commonly used alternative technical solutions that are available today to address the problem of data up-to-dateness and highlights their advantages and disadvantages.

2.2.1 Polling

The polling technique is a frequently used solution when solving problem of rendering actual data for many users. It is also an approach currently implemented in the CodeNOW application. It consists of having the client send a request to the server at periodically recurring intervals to find out whether any data has changed or not. After receiving a response from the server, the connection is terminated and a new connection is created with each request.

It follows from the above description that polling might place a heavy load on network traffic, especially if a large amount of data is transmitted. This technique might be also particularly unsuitable for applications where data changes occur less frequently, because many messages will be sent unnecessarily. [3]

It is worth mentioning that the classic polling is a synchronous technique, because the server sends a response right after receiving a request from the client.

The advantage of this approach is an easy implementation in comparison to other techniques.

The principles of the communication are shown in Fig. 7.

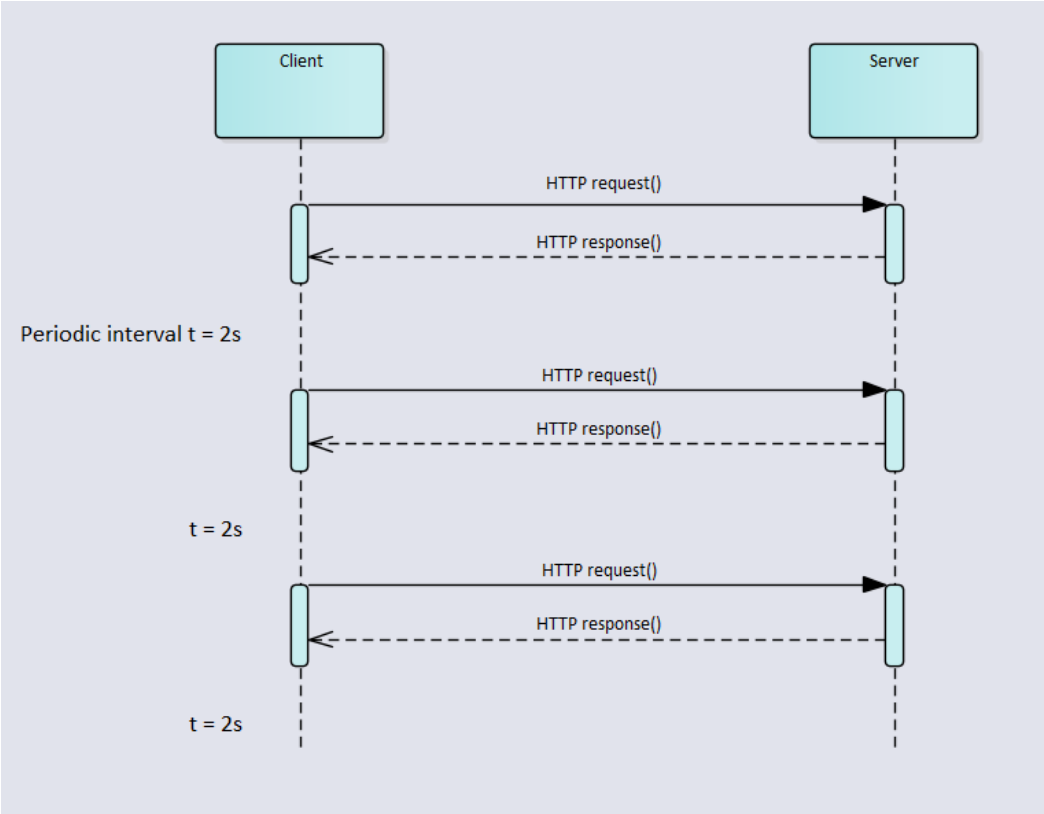

Fig. 7 – Polling communication example

2.2.2 Long Polling

In case of long polling, the server does not have to respond to the client request immediately if it does not have the required information available. In this case, the server keeps the connection open until the data is available and then sends a response to the client.

After receiving a response from the server, the client sends the same request again. As with polling, a new connection is created for each request-response pair. [14]

Problems may occur if there are many requests that the server must keep. This could potentially lead to memory and performance issues if the number of clients is high.

Compared to the polling technique discussed above, the long polling technique is asynchronous, because the server does not respond immediately to the client's request, but waits until the data needed is available. However, in terms of central processing unit (CPU) usage and memory consumption, it achieves worse results than WebSocket communication. [15]

The principles of the communication are shown in Fig. 8.

2. Problem of asynchronous communication

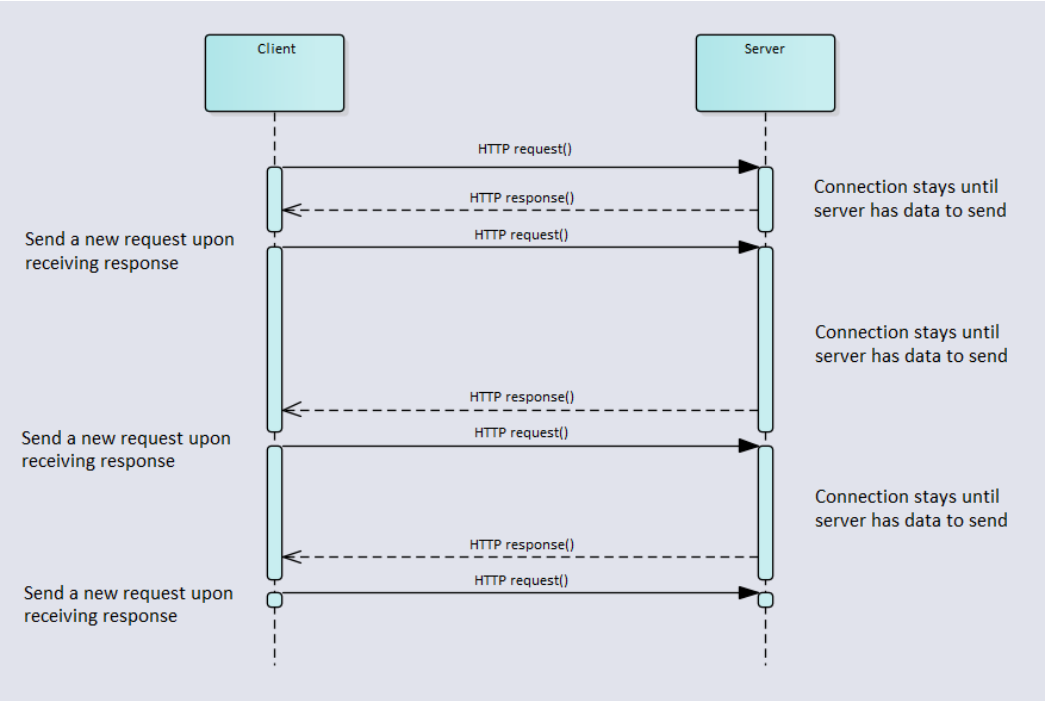

Fig. 8 – Long Polling communication example

2.2.3 Server-Sent Events

Server-Sent Events (SSE) technology allows the client to automatically receive messages from the server using an HTTP connection. As with the WebSocket protocol, the server is able to initiate a connection without being requested to do so by the client. The connection is kept open after it is established. Unlike WebSocket communication, SSE provides a one-way communication, with messages passing in the direction from server to client. This feature can be very restrictive for applications which need also the client to send messages to the server. [16]

However, in terms of performance, the SSE technique achieves similar results as the WebSocket communication. [15]

In the case that there is no need for messages to be send from the client and we are satisfied with one-way communication from the server to the client, this technique can be a very good alternative to the WebSocket communication.

Principles of the SSE communication are shown in Fig. 9.

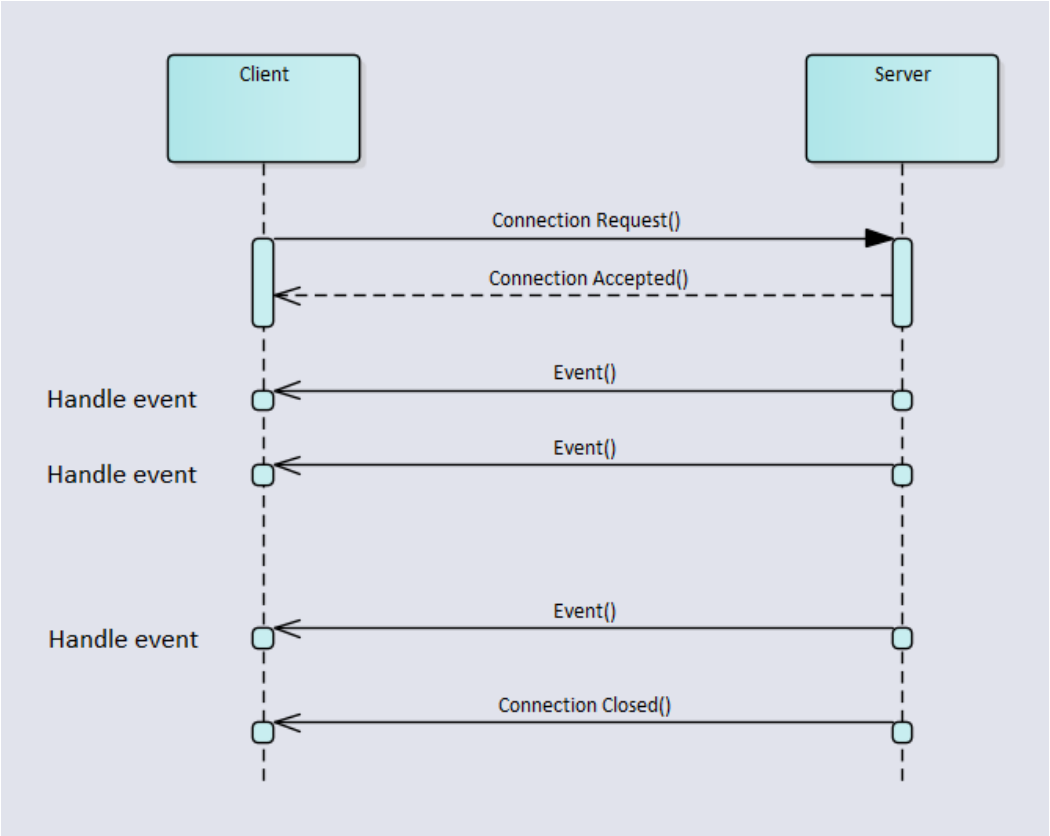

Fig. 9 – SSE communication example

2.2.3.1 Server-Sent Events vs WebSocket

The previous chapter concerning the alternative approaches showed that Server-Sent Events is a very capable competitor of communication using a WebSocket protocol. Both techniques significantly outperform polling or long-polling techniques. [15]

SSE has the advantage of being easy to implement and it expected to perform very similar to WebSocket, both in terms of CPU usage and memory consumption. [15]

The disadvantage of SSE is a less variability because it provides one-way communication. SSE practices one-way communication, and if we decided or were forced to send messages from the client to the server, we would not be able to do so with SSE.

In the case of SSE, it is important to keep in mind that there are a limited number of six connections pre browser. [16]

Due to the greater variability provided by full-duplex communication, WebSocket communication has been chosen for the purposes of this work.

3 Analysis and solution design

A web application may consist of several components. There is always a frontend component that contains the user interface, and then there might be one or more backend components and other services, such as a database. One of these backend components may act as an API component that receives requests from the client and sends them to the appropriate backend component. CodeNOW also uses a similar architecture. An example of a simple web application architecture is shown in the component diagram in Fig. 10.

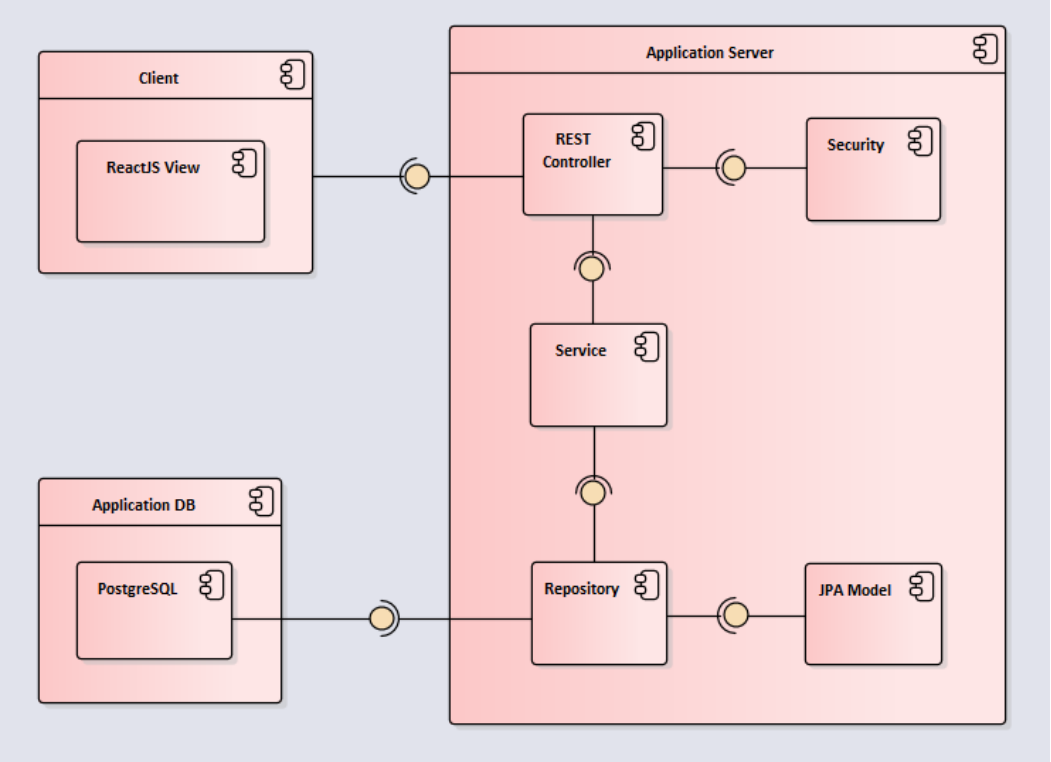

Fig. 10 – Web application architecture example

3.1 CodeNOW AS – IS communication status

In the current state, a frontend component performs many queries on various backend components using HTTP polling. As CodeNOW grows in size and gains more users, polling starts causing performance issues as there are more and more requests to which the backend must respond.

3.1.1 Influence of the customer type

CodeNOW is intended to serve many users at once. For this reason, polling may not be the best solution for communication between the frontend and the backend, as it can

place a lot of load on backend components that must send responses to these users every few seconds, in the current CodeNOW implementation it is 15s.

3.1.2 CodeNOW deployment

The CodeNOW application runs in a cloud environment, where it is deployed using Kubernetes configuration files. [17] With the help of Kubernetes, an open source system for automating deployment, scaling, and management of containerized applications, it is possible to run backend components in multiple replicas, so-called pods, on multiple machines, so-called nodes. The network traffic can be then divided between these nodes using a load balancer component. However, this brings greater demands on the purchased computation power and thus makes the operation more expensive overall. In other words, if the problem can be solved at the code change level, it is definitely a better and cheaper solution than buying more powerful hardware or more machines to run the application on.

3.2 CodeNOW TO – BE communication status

The high-level goal is to significantly reduce the frequency of sending queries to the server and also to reduce the volume of data transferred. The target state to be achieved by this work should replace periodic HTTP queries and use asynchronous events instead.

An asynchronous approach should streamline communication between the frontend and backend, improving the responsiveness of the entire application and thus the user experience.

Although the network situation is not currently critical, it is expected that in near future, as CodeNOW expands to new countries and gains more users, this problem will appear soon and it is important to be prepared for it.

3.3 Architecture

In terms of this work, the detailed CodeNOW architecture is abstracted, because it contains several components that are irrelevant to this work and also I want to point out that principles discussed in this work apply in general to a wide range of applications.

The frontend component, which is based on ReactJS, a JavaScript library for building user interfaces, used together with the TypeScript language for greater type security, is an especially important part of the solution. [18] [19]

The frontend connects to the API component, which is based on the Java Micronaut framework, which can be described as a modern, open source, JVM-based, full-stack toolkit. [20] The API component receives client requests, authenticates them using its security service, and calls the appropriate backend component to execute the request.

When calling backend components, it can use its client service, since in this case it is the API component that acts as the client.

In Fig. 11 below, there is just one component representing the backend logic, but it is not uncommon to have several backend components, each dealing with its own domain.

There might be also a message broker, for example Apache Kafka, that provides asynchronous messaging between the API component and backend logic component, so that the API component would use Kafka topics to push and pull messages and to communicate with the backend logic component instead of calling its REST[6] controllers using HTTP. [21] [22] The backend logic component contains services needed to process requests.

There is also a database, specifically the widely used open source relational database PostgreSQL, because most applications, including CodeNOW, need to store some data. [23]

[6] REST (Representational State Transfer) is an architectural style for providing standards between computer systems on the web.

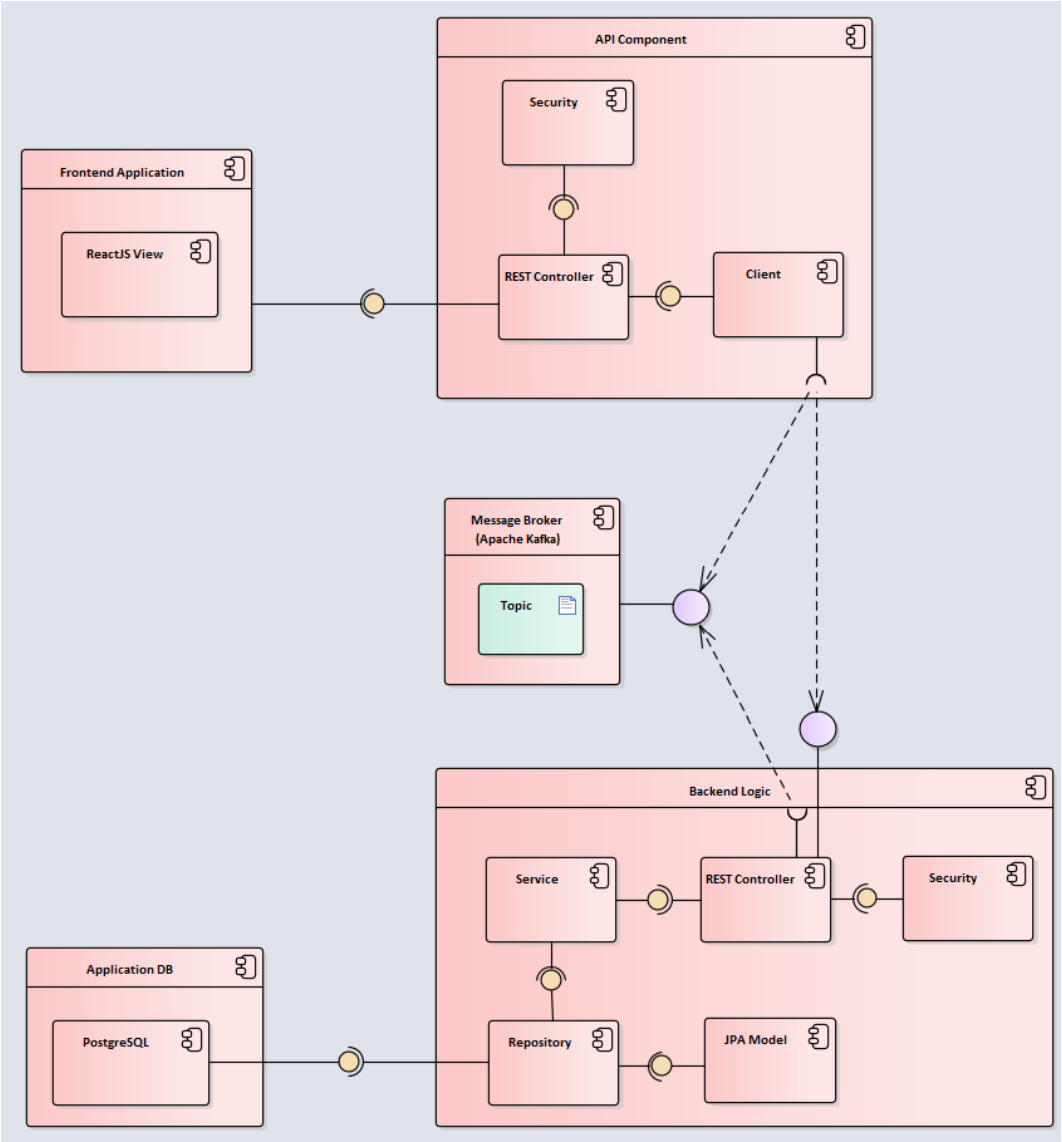

Fig. 11 – Simplified CodeNOW architecture

3.4 Deployment

Fig. 12 shows a deployment diagram of the simplified CodeNOW architecture. The whole application is deployed to a Kubernetes cluster and each component runs in a Docker container, which can be interpreted as a standalone, executable package of software that includes everything needed to run an application. [24] These containers run in so-called pods. Each component in the cluster runs on a separate machine called a node and runs the Linux operating system. The user device may run any operating system as it only needs access to a web browser. You can see that WebSocket communication is used between the frontend running on the user device and API component. Also there might be a message broker component between the API and Application Server components which provides asynchronous messaging. Otherwise HTTP(s) communication is used.

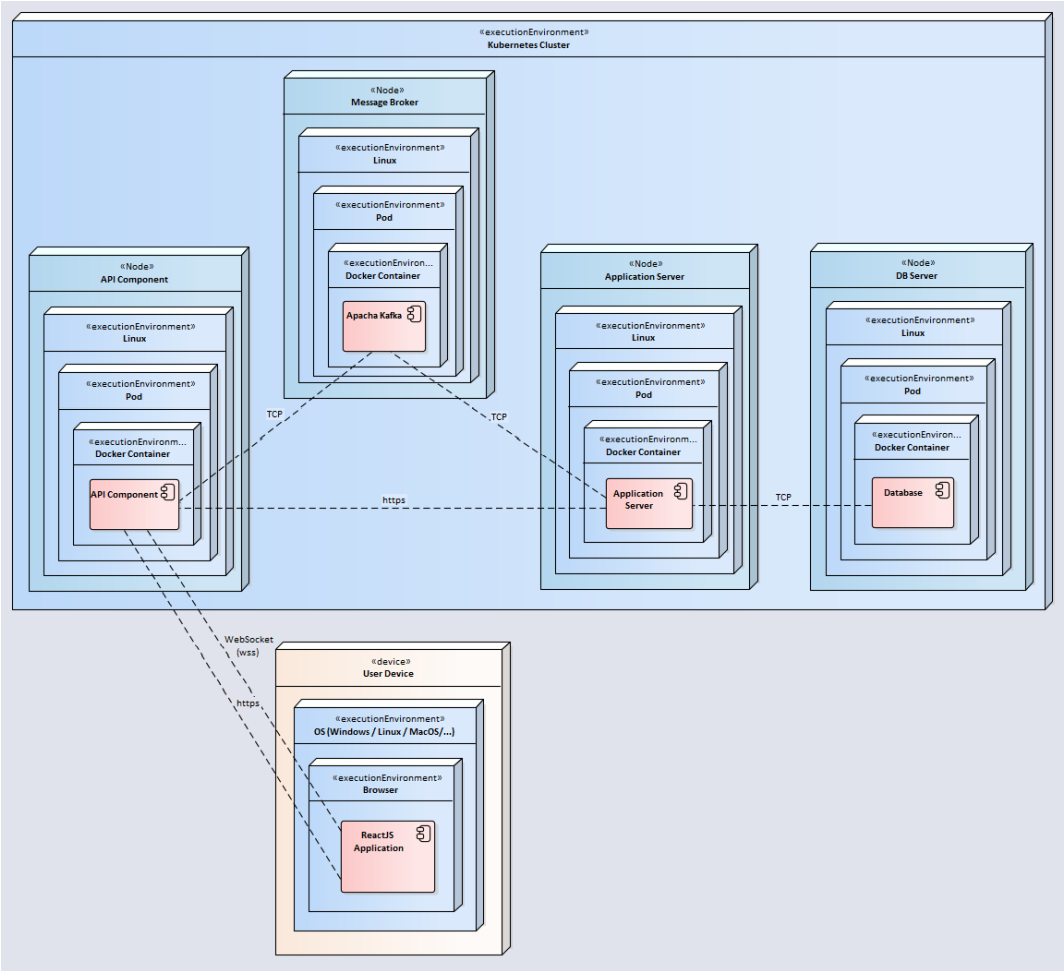

Fig. 12 – Simplified CodeNOW deployment scheme

4 Proof of Concept

To test whether the proposed solution works correctly and meets the requirements mentioned in Section 1.1, the problem of polling queries designed to obtain a list of created applications for a given account in CodeNOW was selected.

After a user performs an action on the application resource, e.g. user creates a new application, other users should be notified about this fact by a message from the server.

4.1 Application resource

The basic and most important resource in CodeNOW is the application. The size of this object depends on many factors such as the number of its components, number of commits, builds, application packages, deployment configurations, deployments and contributors. There are many other properties such as DORA[7] metrics for each application.

Fig. 13 shows what the Application card looks like in CodeNOW.

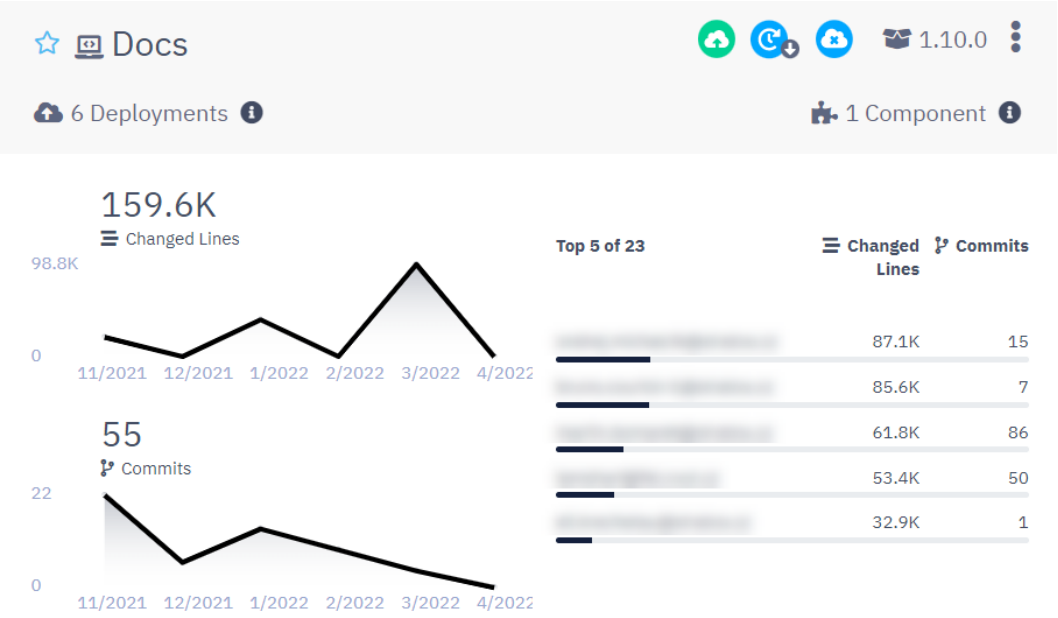

Fig. 13 – Application card in CodeNOW

[7] DORA metrics are used by DevOps teams to measure their performance. The four metrics are deployment frequency, lead time for changes, time to recovery, and change failure rate.

4.1.1 Analysis of application properties

If a user performs an action, it may or may not change the appearance of the application card. As shown in Fig. 14, the following application properties are susceptible to user actions (the numbers in the figure correspond to the ordered list):

1. Application name
2. Number of deployments
3. Latest application package version
4. Number of application components
5. Number of changed lines
6. Number of commits
7. Top contributors

For the purposes of the proof of concept, I will focus on the first four of the seven properties because they are the most important to the user in terms of working with the application.

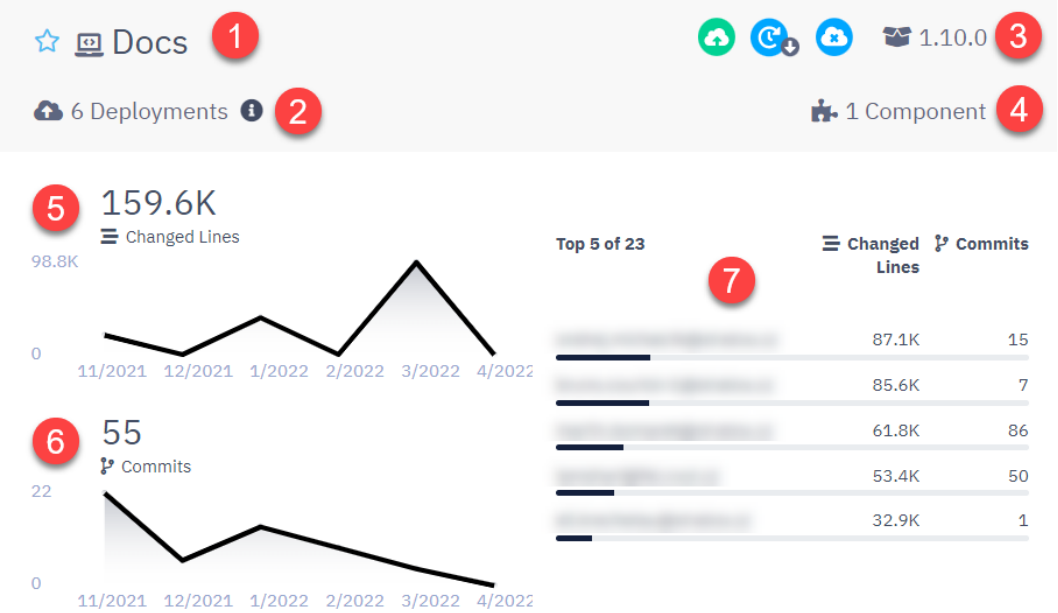

Fig. 14 – Analysis of the application card

4.1.2 Size

Currently, for an average-sized application, the size of the object is approximately 6kB. This number was determined using the JSON Size Analyzer website, which calculated the value for a real application object from CodeNOW in JSON[8] format. [25]

[8] JavaScript Object Notation (JSON) is a popular lightweight data-interchange format.

4.1.3 Network effect on CodeNOW

Let's consider 10 such applications per CodeNOW account and 25 users working with them. Given the information in the previous subsections, the size of the object containing all the applications that the client receives from the server is approximately 60kB (6kB*10 applications). If we take into account the number of users, the server must send 1500kB (60kB*25 users) so that everybody receives the data.

If the polling approach is implemented, it results in 1500kB of data transmitted every few seconds depending on the polling interval. I would like to emphasize that this number varies with the number of applications and the number of users.

Although the application is the biggest resource, there are several other resources in CodeNOW, such as an application component, a managed component, or a cluster, for which the same principle applies, but those resources will not be further discussed in this research.

4.2 Solution proposal

The first step is to establish a WebSocket connection between the frontend and the API component. This requires implementing such functionality on the server to be able to communicate via the WebSocket protocol.

I have created two versions of the solution, each has its advantages and disadvantages and can be suitable for different types of applications.

4.2.1 Version 1 – HTTP/WebSocket hybrid communication

This version represents a kind of hybrid solution between the HTTP and WebSocket communication. It uses the WebSocket protocol to notify the frontend that the backend has performed a modifying action on an application resource. The frontend may then either send the HTTP request to fetch actual data, or ignore the message in case the data is not needed. Also, whenever a user visits the page, an HTTP GET request is performed to the server to retrieve the applications.

Fig. 15 shows a sequence diagram showing the behavior of this version when a user visits the page with applications and when the frontend receives a notification message from the server.

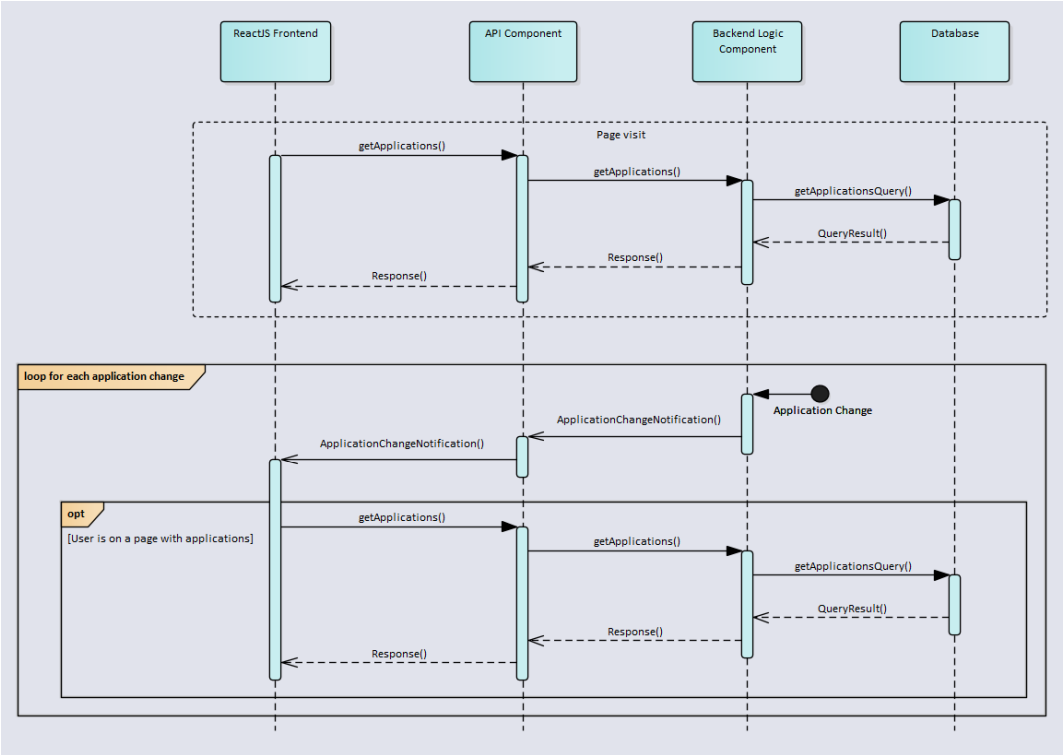

Fig. 15 – HTTP/WebSocket communication sequence diagram

4.2.1.1 Backend

The following applies to the API component. This component will contain a REST controller and a @Post endpoint in it. The aim of this endpoint is to receive messages from other backend components and broadcast[9] them to all connected WebSocket clients. In most cases, the WebSocket connections are maintained internally by the framework you use, so you do not have to worry about them, i.e. the connections do not need to be manually stored in the code.

You can now call this endpoint from other backend components in methods that modify the application object.

4.2.1.2 Frontend

After receiving a message from the server, the frontend knows that changes have been made to an application object. It has to decide whether this affects the page the user is currently on. If the user is on a page where applications are displayed, it performs an HTTP GET request to the API component to get the actual applications. If the action does not affect the page, it ignores the message.

[9] [9] Broadcasting refers to transmitting a message that will be received by every connected device on the network.

4.2.2 Version 2 – Pure WebSocket communication

This version minimizes the usage of the HTTP protocol. The key difference is that now the frontend stores the data needed to render a specific page, in our case the page with applications.

This requires that the WebSocket message sent by the server contains additional information about the object on which the action was performed. With that, the HTTP request does not need to be performed to refresh the data as in previous version. Instead, the data stored in the frontend are modified according to the content of the received message.

This approach also means that whenever user re-enters the page, the stored data is used.

Fig. 16 shows a sequence diagram showing the behavior of this version when a user visits the page with applications and when the frontend receives a notification message from the server.

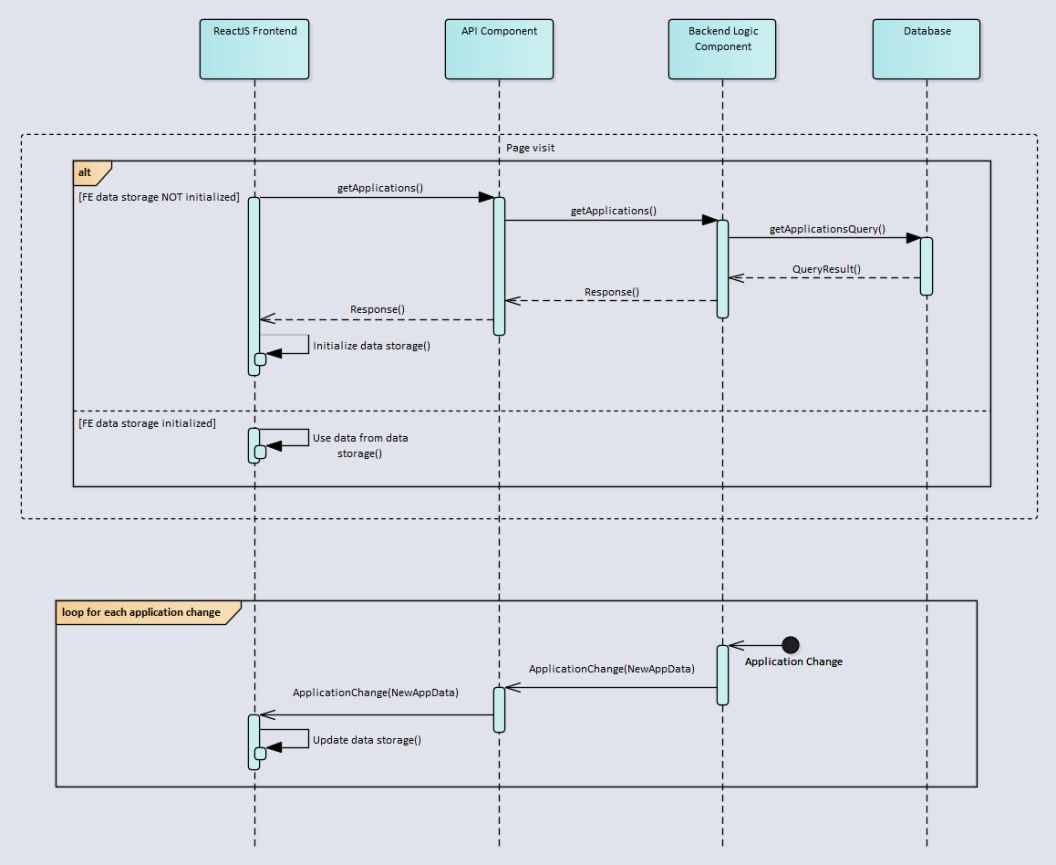

Fig. 16 – Pure WebSocket communication sequence diagram

4.2.2.1 Backend

This approach requires that the message contains all the information needed for the modifying operation in the frontend. In case of application creation or update, the

message must contain the newly created or updated application. In case of deletion, it should be enough to send an identifier of the deleted application.

As an optimization, in the case of an update action, only changed properties of the changed object can be transferred to the frontend, so that we reduce the amount of transmitted data.

4.2.2.2 Frontend

As mentioned above, the idea here is that the frontend stores the data needed to render a page showing applications. When a user enters the page for the first time, a regular HTTP GET request is performed to the server because the frontend does not have the data yet. After receiving the message, the data in the response is stored. Whenever the user accesses the page again, the stored data is used. In other words, no additional HTTP request is needed.

4.2.3 Possible modification

An option for the future is to use a component acting as a message broker between the frontend and the API component. A message broker is a component whose purpose is to receive, store, and send messages. For example, RabbitMQ or Apache Kafka could play this role as they meet the above requirements. [26] The backend component on which the WebSocket server functionality is implemented would not need to internally store all WebSocket connections, but would only know the network address of the message broker. The message broker would contain a message queue to which the backend would send messages as a producer. The frontend would then consume the messages from this queue as a consumer.

5 Implementation

The solution requires modifications on the client and server side of CodeNOW. Two versions of the solution were developed according to the proposals described in the 4.2Solution proposal chapter.

This implementation uses ReactJS for the frontend solution part, and Java Micronaut for the backend solution part.

I would like to point out that although the above frameworks are used for the purposes of this work, an implementation using other frameworks is possible, but features like React hooks will need to be replaced with appropriate code.

5.1 Basic functionality for both versions

First, let's implement the basic functionality that will be used by both implementation versions.

5.1.1 Backend

In this step, we need to implement a WebSocket server that is able to accept a connection request from a client and establish a WebSocket connection, the so-called WebSocket tunnel.

As long as we want the server to just accept the connection request, we do not need to implement any methods that process received messages or send messages.

Example implementation written in Java Micronaut framework is shown in Fig. 17.

```
...imports

@ServerWebSocket("/ws/applications")
public class WebSocketServer {
    private WebSocketBroadcaster broadcaster;

    public WebSocketServer(WebSocketBroadcaster broadcaster) {
        this.broadcaster = broadcaster;
    }

    @OnOpen
    public Publisher<String> onOpen(WebSocketSession session) {
        return Flowable.empty();
    }

    @OnMessage
    public Publisher<String> onMessage(Object message, WebSocketSession session) {
        return Flowable.empty();
    }

    @OnClose
    public Publisher<String> onClose(WebSocketSession session) {
        return Flowable.empty();
    }
}

```

Fig. 17 – WebSocket server example in Java Micronaut

5.1.2 Frontend

In order to communicate using the WebSocket protocol, we need to create a WebSocket object. This will automatically attempt to open the connection to the server.

The WebSocket constructor accepts one required and one optional parameter as discussed in the chapter 2.1.2.

After that, we can set event listeners for open and message events. You can see the example implementation in Fig. 18.

```
const websocket = new WebSocket("wss://server-url/ws/applications");

websocket.addEventListener("open", () => console.log("open"));
websocket.addEventListener("message", handleMessage);

```

Fig. 18 – WebSocket – Creation and event listeners

We can also use a React hook called *useWebSocket*. [27] To be able to use this hook, it needs to be installed using *npm*, a package manager for the Node JavaScript platform.

[28] We can do so by running the following command in the terminal: *npm i react-use-websocket*

The use of the *useWebSocket* hook may bring a few advantages because it solves several problems for us, such as automatic reconnecting in case of an error. For this reason, I will use this hook further.

See the example implementation in Fig. 19.

```
1  const websocket = useWebSocket("wss://server-url/ws/applications", {
2      onOpen: () => console.log("open"),
3      onMessage: (message) => handleMessage(message),
4      shouldReconnect: (closeEvent) => true,
5  });
6
```

Fig. 19 – useWebSocket hook

5.2 Version 1 – HTTP/WebSocket hybrid communication

As we have our basic functionality prepared, we can now start implementing the first solution version.

5.2.1 Backend

In this step, we will add a REST controller with a @Post endpoint, whose intention is to be called from other places in the backend whenever an application is created, changed or deleted. After being called, it broadcasts the message over all WebSocket connections that are held on the WebSocket server. The message may contain the type of the object, the type of action performed, the object ID, and other properties.

An example implementation is shown in Fig. 20 below. Note that in this example, messages are sent only to those clients that meet a certain requirement, i.e. we can check their *accountId* property present in the client session.

```java
...imports

@Controller("/event-receiver")
@RequiredArgsConstructor
public class EventReceiver {

    private final WebSocketBroadcaster broadcaster;

    @Post
    public void onApplicationChange(@NotNull @Valid @Body EventInformation eventInformation) {
        broadcaster.broadcastSync(eventInformation, (session) -> eventInformation
                .getAccountId()
                .equalsIgnoreCase(session.getUriVariables().get("accountId", String.class, null)))
        );
    }
}

```

Fig. 20 – Event receiver controller example

5.2.2 Frontend

The frontend should be able to decide whether we want to react on the application change message or not. To be able to do so, let's hold resource types, on whose changes we want to call a refresh callback, inside a *useState* hook as shown in Fig. 21 below. [29] If the coming event concerns the resource type that we are currently interested in, the refresh call is performed.

```javascript
const [resourceTypes, setResourceTypes] = useState([]);

useWebSocket("wss://server-url/ws/applications", {
  onMessage: ({ data }) => {
    const parsedData = JSON.parse(data);
    if (resourceTypes.includes(parsedData.resourceType)) { // resourceTypes = ["application", "cluster"]
      refreshCallback();                                   // parsedData.resourceType = "application"
    }
  },
  shouldReconnect: () => true,
});
```

Fig. 21 – useWebSocket (v1)

This implementation might have a performance drawback if we call this piece of code from multiple places in our code because multiple WebSocket connections will be established. This would happen if our web application displayed applications on multiple pages. In this case, each page would create a WebSocket connection. This problem also occurs if the user accesses such a page repeatedly. The connection would be re-established each time user re-enters it.

5.2.2.1 Optimization

In order to establish just one WebSocket connection per user, the React *useContext* hook can be used. [29] This hook allows to share one WebSocket connection through the entire application. Using this design pattern reduces connection overhead because the handshake is performed only once after the application starts.

The *useContext* hook will in this case contain an array of resource types on whose events the connection listens, and also the callbacks for performing refresh calls. We can then set these variables on the individual pages of the application.

Fig. 22 shows the example implementation.

```
const WebSocketContext = React.createContext(() => {});

export const useWebSocketContext = () => useContext(WebSocketContext);

export const WebSocketTopicProvider = ({ children }) => {
  const [resourceTypes, setResourceTypes] = useState([]);
  const [refreshCallback, setRefreshCallback] = useState();

  useWebSocket("wss://server-url/ws/applications", {
    onMessage: ({ data }) => {
      const parsedData = JSON.parse(data);
      if (resourceTypes.includes(parsedData.resourceType)) {
        refreshCallback();
      }
    },
    shouldReconnect: () => true,
  });

  return (
    <WebSocketContext.Provider value={{ setResourceTypes, setRefreshCallback }}>
      {children}
    </WebSocketContext.Provider>
  );
};

```

Fig. 22 – useWebSocket (optimized)

In components that work with resources that are interesting for us from the WebSocket perspective, we can use this context and set the resource type and refresh callback variables.

5.3 Version 2 – Pure WebSocket communication

This version uses again the basic functionality implementation described in Section 5.1 as the starting point.

5.3.1 Backend

The implementation in this version follows the same procedure as in version 1 so we can reuse the code from the previous version. The only difference is that the event that is sent from the endpoint to the WebSocket client must contain the entire application object. Only in case of application deletion, it should be enough to send only the application ID.

5.3.2 Frontend

As described in Section 4.2.2, the idea here is that the frontend should store the data needed to render a page that contains applications.

To keep things simple, we can create variables that will hold the data. We can define them using TypeScript as shown in Fig. 23.

```
export interface FeDataProps {
  appCards: { data: any[]; initialized: boolean };
}

export const FE_DATA: FeDataProps = {
  appCards: {
    data: [],
    initialized: false,
  },
};

```

Fig. 23 – Frontend data storage example

Now let's adjust our *useWebSocket* hook. After receiving a message from the server, the message should be passed to an event handler function. See Fig. 24.

```
export const WebSocketClient = () => {

  useWebSocket("wss://server-url/ws/applications", {
    onMessage: ({ data }) => {
      const parsedData = JSON.parse(data);
      const eventInformation = {
        resourceType: parsedData.resourceType,
        eventType: parsedData.eventType,
        object: parsedData.object,
      };
      handleWsEvent(eventInformation);
    },
    shouldReconnect: () => true,
  });
  return null;
};

```

Fig. 24 – useWebSocket hook (v2)

The event handler will then check the action property which is contained in the received message and according to its value, it will adjust the data storage variables.

After that, pages that use these data re-render to show actual data.

Fig. 25 shows how the handler can be implemented.

```
const handleWsEvent = ({ resourceType, eventType, object }: EventInformation): void => {
  switch (resourceType) {
    case "application":
      handleAppChange(eventType, object);
      notifyAppObservers(FE_DATA.appCards.data);
      break;
    case other resource types...
  }
};

const handleAppChange = (eventType: string, object: any) => {
  switch (eventType) {
    case "CREATE":
      FE_DATA.appCards.data.push(object);
      break;
    case "UPDATE":
      FE_DATA.appCards.data.forEach((card) => {
        if (card.application.id !== object.id) card.application = object;
        return;
      });
      break;
    case "DELETE":
      FE_DATA.appCards.data = FE_DATA.appCards.data.filter((card) => card.application.id !== object);
  }
  FE_DATA.appCards.initialized = true;
};

```

Fig. 25 – event handler example

5.3.2.1 Notes

In components that use WebSocket communication, there is a *useEffect* React hook that is called when a user enters the page. [29] If the application data that the page uses is not initialized in the frontend, then it performs an HTTP GET request to the server and stores the data, as seen above. If the data has already been initialized, the page reuses it.

An implementation of a component which renders applications and reuses the stored data is shown Fig. 26 in below.

```
const ComponentWithApplications = () => {

  const [applications, setApplications] = useState(FE_DATA.appCards.data || []);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await dataProvider.get("/applications");
        return await response.data.applications;
      } catch (err) {}
    }
    if (!FE_DATA.appCards.initialized) {
      subscribeToAppChanges("appComponent", setApplications);
      fetchData().then((data) => {
        FE_DATA.appCards.data = data;
        FE_DATA.appCards.initialized = true;
        setApplications(data);
      });
    }
    return () => {
      unsubscribeToAppChanges("appComponent");
    };
  }, []);

  return (
    <Page>
      ...
    </Page>
  );
};
```

Fig. 26 – Component rendering applications example

In order to re-render the page automatically after the frontend has received the message, pages use a *useState* hook which contains the data needed. If this data changes, the page re-renders automatically because that is the feature of the *useState* hook. In order to change the data in all interested pages, pages subscribe to the event handler. Technically, this means that event handler stores functions which set the data in the *useState* hook for those pages. If the data stored is changed, subscribed components are notified, as seen in Fig. 25.

6 Achieved objectives

Both versions of the solution were developed as proposed in Section 4.2. Design and implementation of the solution were made in the iterative way according to the assignment. First, the proof of concept of the *useWebSocket* hook was created and it was verified to be useful for the purpose of this work. After that, the first version of the solution was designed and in consultation with the CodeNOW development team. After it was approved, the first version of the solution was successfully implemented and deployed on CodeNOW. Then, the results of this solution were observed in CodeNOW, as discussed further in this document. After successfully completing the first version and measuring the impact on the network efficiency, the second version was designed with the aim of making the network communication even more efficient. After this version had been approved, it was again successfully implemented and deployed on CodeNOW. Also in this case, the results of this solution were observed in CodeNOW and compared to the results of the first solution afterwards. The result of the comparison is discussed further in this document in Section 7.

6. Achieved objectives

7 Test

This section presents the results of observations and measurements of each version, including the original implementation in CodeNOW.

7.1 Test scenarios

The considered situation assumes 25 users and 10 applications in CodeNOW. As mentioned in the section 4.1.2, I will assume the average application size of 6kB.

According to CodeNOW's development team, a user spends approximately 10 minutes (600 seconds for calculations) per working day on the page with applications, visiting the page 20 times. These figures were determined as medians by six CodeNOW developers who estimated their time spent on the page and the number of visits, as shown in the table below. [Fig. 27]

Developer	Time spent on the page with applications per day [s]	Page visits per day
Developer 1	900	40
Developer 2	600	20 - 25
Developer 3	600	20
Developer 4	600	25
Developer 5	300	10 - 15
Developer 6	600	25

Fig. 27 – Estimates of the CodeNOW development team

Also, the number of application actions, such as creating an application package or committing, must be taken into account. This number, determined as an average value, was set to 5 actions per application per day.

Last but not least, not only the actual payload but also the request and response header should be taken into account. According to Firefox Developer Tools, the size of the request for applications is about 1.1kB and the size of the response header is about 0.4kB which makes it approximately 1.5kB together.

Finally, the sizes of the event messages for both versions need to be determined. For version 1, only the *resourceType* property with its value can be a payload, and thus the total message size is approximately 0.5kB. For version 2, the payload is the application object (6kB), and therefore the total amount of data transferred is approximately 6.5kB.

For the test purposes a few constants need to be introduced:

- Number of users ..... $U = 25$
- Visits to the page per user per day..... $V = 20$
- Time spent on the page per user per day ..... $T = 600\ [\mathrm{s}]$

- Total size of applications ..... $S_A = 6 * 10 = 60\ [kB]$
- Total size of one request – response pair ..... $S = S_A + 1.5 = 61.5$ [kB]
- Size of an event message (version 1) ..... $S_{M1} = 0.5$ [kB]
- Size of an event message (version 2) ..... $S_{M2} = 6.5$ [kB]
- Number of application actions per day ..... $A_D = 10 * 5 = 50$
- Day (8h) / *T* ratio ..... $R = (8 * 3600) \div 600 = 48$
- Number of application actions during *T* ..... $A_T = A_D \div R \approx 1$

In each implementation version, we will monitor two variables:

- Number of performed requests per day..... *R*
- Amount of data transferred per day..... *D [kB]*

With that defined, let's take a look at each solution in turn and see how *R*'s and *D*'s differ.

7.1.1 Current status

Currently, CodeNOW uses polling to retrieve applications with a 15s interval between repeated requests. Whenever user enters the page with applications, a new request is made.

7.1.1.1 Results

We can determine *R* as the time spent on the page per user divided by the polling interval, multiplied by the number of users.

Then, *D* can be determined as the product of *R* and the size of applications.

Note: We can omit the number of page visits as it would not have a noticeable impact in this case.

$$R = (T \div 15) * U$$

$$R = (600 \div 15) * 25$$

$$R = 1000$$

(Eq. 1)

$$D = R * S$$

$$D = 1000 * 61.5$$

$$D = 61\,500\ [kB] \approx 61.5\ [MB]$$

(Eq. 2)

The volume of the data transferred in this implementation is observed to be **61.5MB**.

I would like to point out that the following situation can easily happen. Consider a user who opens the page with applications, but then leaves for lunch and returns back in an hour. In this case, we have *R* equal to 1240, because we need to add the requests performed during this hour to the current number of requests. Therefore, *D* is equal to approximately 76.3MB as shown in the following equations.

$$R = 1000 + (3600 \div 15)$$

$$R = 1\,240$$

(Eq. 3)

$$D = R * S$$

$$D = 1240 * 61.5$$

$$D = 76\,260\ [kB] \approx 76.3\ [MB]$$

(Eq. 4)

We can see that in this implementation, the results are very susceptible to similar scenarios like the one above. This is not the case for version 1 or version 2 since similar scenarios may change the resulting *D* only slightly (version 1) or not at all (version 2).

7.1.2 Version 1

This version requests data whenever user enters the page, but also in case the user is on the page with applications and receives a message concerning a change of an application.

7.1.2.1 Results

We can determine *R* as the number of page visits per user multiplied by the number of users, plus the expected number of requests preformed for all users in response to messages received while on the application page.

Then, *D* can be determined as the product of *R* and the size of applications, plus the volume of event messages received from the server for all users.

$$R = (V * U) + (A_T * U)$$

$$R = (20 * 25) + (1 * 25)$$

$$R = 525$$

(Eq. 5)

$$D = (R * S) + (A_D * U * S_{M1})$$

$$D = (525 * 61.5) + (50 * 25 * 0.5)$$

$$D = 32\,912.5\ [kB] \approx 32.9\ [MB]$$

(Eq. 6)

The volume of the data transferred in this implementation is observed to be **32.9MB**.

Let's assume again the situation described earlier, where one user leaves for lunch for an hour while staying on the page with applications. Let A_I be the number of actions additionally performed in response to server notifications during the hour when the user left. Then A_I is calculated as

$$A_I = A_D \div 8$$
$$A_I = 50 \div 8$$
$$A_I \approx 6$$

(Eq. 7)

Then the total amount of data transferred is equal to the previously computed D plus the additionally transferred data.

$$D_{totat} = D + 6 * S$$
$$D_{total} = 32\,912.5 + 6 * 61.5$$
$$D_{total} = 33\,281.5\ [kB] \approx 33.3\ [MB]$$

(Eq. 8)

We can see that the difference between the originally assumed scenario and the modified scenario is approximately 0.4MB which is a much smaller increase compared to the current implementation of CodeNOW where the increase is 14.8MB.

7.1.3 Version 2

This version requests data only when the user first enters the page. Any time after that, when the user enters the page, the stored data is used. However, the data received as event messages from the server must also be taken into account.

7.1.3.1 Results

We can determine *R* as the number of users.

Then, *D* can be determined as the product of *R* and the size of applications plus the size of event messages received from the server due to an application change. This message contains the modified application (6kB) and is therefore approximately 6.5kB in size.

$$R = U$$

$$R = 25$$

(Eq. 9)

$$D = (R * S) + (A_D * U * S_{M2})$$

$$D = (25 * 61.5) + (50 * 25 * 6.5)$$

$$D = 9\,662.5\ [kB] \approx 9.7\ [MB]$$

(Eq. 10)

The volume of the data transferred in this implementation is observed to be **9.7MB**.

In the case of this version, the previously discussed scenario of one user leaving for lunch does not change the volume of data transferred because it does not matter whether the user is on the page with applications or not.

7.2 Results Comparison

As can be seen from the above results, there are significant differences in the resulting amount of data transferred in each version.

With the specified data, the current version requires approximately 61.5MB of data transferred, version 1 reduces this figure to approximately 50% with its 32.9MB of data transferred, and if version 2 is implemented, the amount of data transferred can be reduced to approximately 9.7MB, which is around 16% of the data required by the current implementation.

A graphical representation of the differences between the different implementations under the conditions defined above is given in the following graph. [Fig. 28]

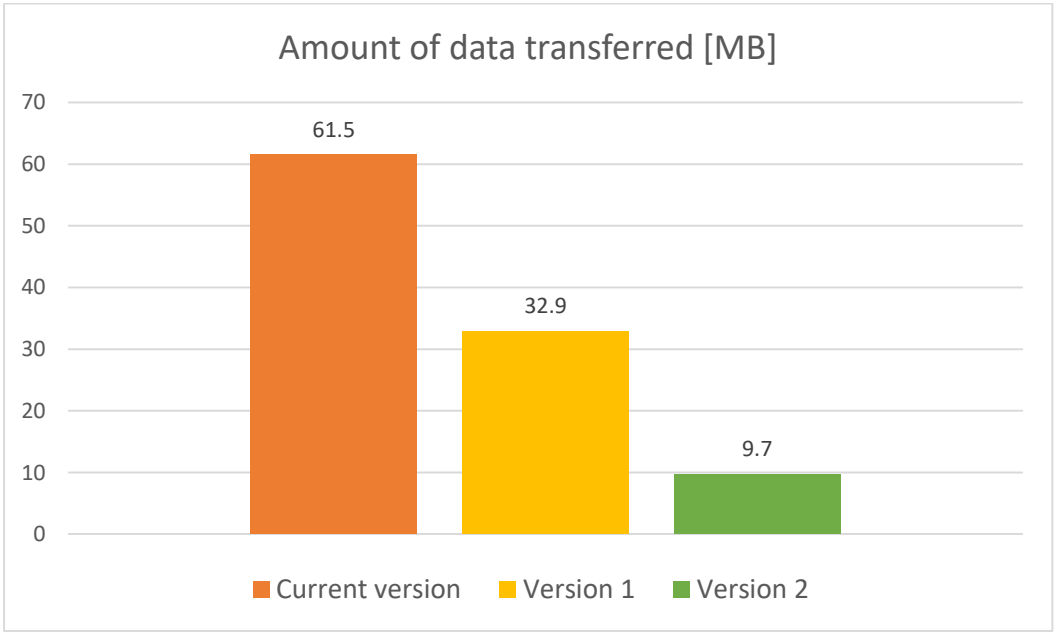

Fig. 28 – Data volume measurement results

A scenario where one user leaves for an hour for lunch but stays on the page with applications was also discussed, and network effects were measured. The following graph shows a comparison of the measured values. [Fig. 29]

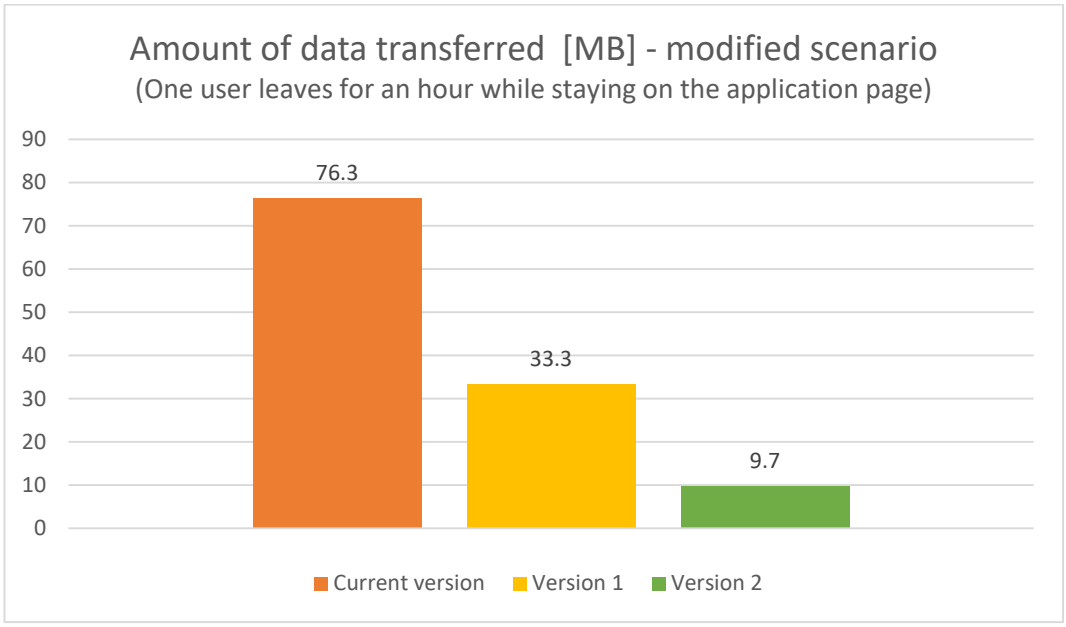

Fig. 29 - Data volume measurement results of the modified scenario

8 Conclusion

The motivation for this work was to streamline the performance of the CodeNOW software factory through asynchronous communication.

Although the assignment concerned CodeNOW as the environment for developing and testing the solution, replacing synchronous communication with asynchronous communication is a general problem that is relevant to a wide range of applications, and therefore the principles discussed in this work apply also to them.

As a part of the problem analysis, the possibility of communication via the WebSocket protocol was discussed in detail, as well as other alternative approaches such as polling, long polling, or Server-Sent Events. In the end, the WebSocket communication was chosen for its greater flexibility and efficiency.

After analyzing the problem, the Proof of Concept solution was developed for specific areas in the CodeNOW product, concretely for the page which renders the list of applications. Within this area it was verified that the proposed solution can be implemented and meets requirements and defined goals.

Two versions of the solution were successfully developed – a hybrid solution that uses both HTTP and WebSocket protocols (version 1), and a pure WebSocket solution (version 2).

Version 1 uses both HTTP and WebSocket communication using the WebSocket protocol to notify the frontend that a backend has performed a modification action on an application resource. The frontend can then either send an HTTP request to retrieve the actual data, or ignore the message if the data is not needed.

Version 2 minimizes the usage of the HTTP protocol. The key difference is that now the frontend stores the data needed to render a specific page, in our case the page with applications.

After deploying each version, observations and measurements were made in order to determine how data efficient each version was.

In a model situation with the specified data, the current version requires approximately 61.5MB of data transferred. Version 1 reduces this figure to approximately 50% with its 32.9MB of data transferred. If version 2 is implemented, the amount of data transferred can be reduced to approximately 9.7MB which is about 16% of the data required by the current implementation. It follows that by far the most effective option is version 2. In addition, the more time a user spends on the page with applications, the greater the difference in efficiency between versions.

In addition to the benefits for network traffic, the solution also contributes to an improved user experience as users are shown up-to-date data. In CodeNOW, the fact that users don't have to wait for a polling interval to see the actual data is not very important, but there are applications for which this would be very beneficial.

What was successful was that I managed to meet all the points of the assignment.

What was not optimal was the difficulty of communication. I spent this semester at the University of Vermont in the United States under a bilateral agreement with CTU, and therefore there was a significant time lag. Because of this, it was sometimes not possible to address current issues with the supervisor and/or the CodeNOW development team in Prague.

8.1 Future steps

The next step is to consider implementing the new functionality in other parts of CodeNOW to apply WebSocket communication wherever polling is used and where it makes sense. This will require modifications on both the client and server side of CodeNOW.

In addition, the objects transferred between the client and the server in event messages can be further optimized so that a minimal amount of data travels over the network.

Currently, there is also a discussion about using the React Query library to work with data on the frontend. [30] This library makes fetching, caching, synchronizing and updating server state easier and can also work with the WebSocket protocol. If the decision is made to use this library, it will be necessary to research whether or not the solution proposed in this thesis needs to be modified.

9 Picture list

9. Picture list

10 Bibliography

[1] CodeNOW. *CodeNOW DevOps Value Stream Delivery Platform* [online]. Praha [cit. 2022-03-31]. Dostupné z: https://www.codenow.com/

[2] MOZILLA FOUNDATION. HTTP. In: *MDN* [online]. 2021 [cit. 2022-03-31]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[3] FAANG, Crack. Ajax Polling vs Long-Polling vs WebSockets vs Server-Sent Events. *Medium.com*. 2021. Dostupné také z: https://medium.com/geekculture/ajax-polling-vs-long-polling-vs-websockets-vs-server-sent-events-e0d65033c9ba

[4] Synchronous vs Asynchronous Communication. In: *Guru: Organize company information* [online]. Guru Technologies [cit. 2022-03-31]. Dostupné z: https://www.getguru.com/reference/synchronous-vs-asynchronous-communication

[5] MOZILLA FOUNDATION. WebSocket. *MDN*. 2021. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket/WebSocket

[6] What is Iterative Design Approach. In: *Wishdesk* [online]. [cit. 2022-04-01]. Dostupné z: https://wishdesk.com/blog/what-is-iterative-design-approach

[7] Iterative Design Approach. In: *Wishdesk* [online]. [cit. 2022-04-01]. Dostupné z: https://wishdesk.com/blog/what-is-iterative-design-approach

[8] Asynchronous. In: *Mozilla Foundation* [online]. https://developer.mozilla.org/ [cit. 2022-01-27]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous

[9] MOZILLA FOUNDATION. TCP. In: *MDN* [online]. [cit. 2022-03-31]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/TCP

[10] MOZILLA FOUNDATION. Writing WebSocket Servers. *MDN*. 2021. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

[11] MOZILLA FOUNDATION. Cross-Origin Resource Sharing (CORS). In: *MDN* [online]. [cit. 2022-03-31]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[12] DE TURCKHEIM, GRÉGOIRE a ROWENA JONES. IoT Hub: What Use Case for WebSockets?. In: *Scaleway* [online]. [cit. 2022-01-18]. Dostupné z: https://blog.scaleway.com/iot-hub-what-use-case-for-websockets/

[13] *Can I use* [online]. [cit. 2022-04-03]. Dostupné z: https://caniuse.com/?search=websocket

[14] SINGH, Siddharth. What is HTTP Long Polling ?. *Educative.io*. Dostupné také z: https://www.educative.io/edpresso/what-is-http-long-polling

[15] APPELQVIST, Rasmus a Oliver ÖRNMYR. *Performance comparison of XHR polling, Long polling, Server sent events and Websockets*. Karlskrona. Bachelor thesis. Blekinge Institute of Technology.

[16] NWAMBA, Christian. *WebSockets vs Server-Sent Events*. 2019. Dostupné také z: https://www.telerik.com/blogs/websockets-vs-server-sent-events

[17] *Kubernetes* [online]. Production-Grade Container Orchestration [cit. 2022-03-31]. Dostupné z: https://kubernetes.io/

[18] *React: A Javascript library for building user interfaces* [online]. [cit. 2022-03-31]. Dostupné z: https://reactjs.org/

[19] *Typescript: JavaScript With Syntax For Types* [online]. [cit. 2022-03-31]. Dostupné z: https://www.typescriptlang.org

[20] *Micronaut: A modern, JVM-based, full-stack framework for building modular, easily testable microservice for serverless applications* [online]. [cit. 2022-04-01]. Dostupné z: https://micronaut.io/

[21] *Apache Kafka* [online]. [cit. 2022-03-31]. Dostupné z: https://kafka.apache.org

[22] What is REST?. In: *Codecademy* [online]. [cit. 2022-04-01]. Dostupné z: https://www.codecademy.com/article/what-is-rest

[23] *PostgreSQL: The World's Most Advanced Open Source Relational Database* [online]. [cit. 2022-04-04]. Dostupné z: https://www.postgresql.org

[24] *Home - Docker* [online]. [cit. 2022-04-01]. Dostupné z: https://www.docker.com/

[25] JSON Size Analyzer. *DebugBear* [online]. [cit. 2022-04-04]. Dostupné z: https://www.debugbear.com/json-size-analyzer

[26] *RabbitMQ: Messaging that just works* [online]. [cit. 2022-03-31]. Dostupné z: https://www.rabbitmq.com/

[27] *Npm* [online]. [cit. 2022-04-01]. Dostupné z: https://www.npmjs.com

[28] React-use-websocket. *Npm* [online]. [cit. 2022-04-01]. Dostupné z: https://www.npmjs.com/package/react-use-websocket

[29] Introducing Hooks - React. In: *React: A JavaScript library for building user interfaces* [online]. [cit. 2022-04-01]. Dostupné z: https://reactjs.org/docs/hooks-intro.html

[30] *React Query: Performant and powerful data synchronization for React* [online]. [cit. 2022-04-11]. Dostupné z: https://react-query.tanstack.com/overview