Research into capabilities of modern web-development tools

# Table of contents

## Introduction

### History of web-development

### Description of the problem

### Application architecture

### Technology stack

## General Development Tools

### Version Control

### Virtualization

### Database

## Back-end

### Language and Framework

### Packet manager and build tool

### JWT Tokens

### Connection with the database

### Accepting connections from the frontend

### Authentication and Authorization

## Front-end

### Language and Framework

### Packet manager and build tool

### Creating styles

### Getting data from the backend

### Routing between pages

### Managing global variables

# Sources

[1] – “**A Brief History of the Internet**”, *University of Georgia*<https://www.usg.edu/galileo/skills/unit07/internet07_02.phtml>

[2] – “**The WorldWideWeb browser**”, *Tim Berners-Lee*<https://www.w3.org/People/Berners-Lee/WorldWideWeb.html>

[3] – “**RFC 883**”, “**RFC 882**”, *Paul Mockapetris (1983)*[*https://datatracker.ietf.org/doc/html/rfc883*](https://datatracker.ietf.org/doc/html/rfc883)[*https://www.rfc-editor.org/rfc/rfc882*](https://www.rfc-editor.org/rfc/rfc882)

[4] – “Hypertext Markup Language (HTML)”, Tim Berners-Lee (1993)  
<https://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt>

[5] – “**History of PHP**”  
<https://www.php.net/manual/en/history.php.php>

[6] – “**Cascading HTML style sheets -- a proposal**”, *Håkon Wium Lie (1994)*<https://www.w3.org/People/howcome/p/cascade.html>

[7] – “**Cascading Style Sheets, level 1**”, *Håkon Wium Lie, Bert Bos (1996)*<https://www.w3.org/TR/REC-CSS1-961217>

[8] – “**Introduction, What is JavaScript?**”  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#what_is_javascript>

[9] – “**ECMAScript: A general purpose, cross-platform programming language**” (1997)  
<https://ecma-international.org/wp-content/uploads/ECMA-262_1st_edition_june_1997.pdf>

[10] – “**Initial** **Release of Node JS on GitHub**”  
<https://github.com/nodejs/node-v0.x-archive/releases/tag/v0.0.1>

[11] – “**Standard ECMA-2626th Edition**”, *(2015)*<https://262.ecma-international.org/6.0/index.html>

[12] - ”**jail, section 9**”  
<https://docs-archive.freebsd.org/44doc/papers/jail/jail-9.html>

[13] – “**Chapter 17. Jails and Containers**”  
<https://docs.freebsd.org/en/books/handbook/jails>

[14] – “**Design, implementation, and evaluation of a Revision Control System**”, *Tichy, Walter (1982)*  
https://dl.acm.org/doi/10.5555/800254.807748

[15] – “**The Source Code Control System**”, *Rochkind, Marc J. (1975)*https://web.archive.org/web/20110525193926/http://basepath.com/aup/talks/SCCS-Slideshow.pdf

[16] – “**Initial revision of "git", the information manager from hell**”, *Linus Torvalds (2005)*<https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290>

[17] – “**1.1 Getting Started - About Version Control**”  
<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

[18] – “**Commits are snapshots, not diffs**”, *Derrick Stolee (2020)*<https://github.blog/open-source/git/commits-are-snapshots-not-diffs>

[19] – “**Server Operating System Market Volume**”  
<https://www.fortunebusinessinsights.com/server-operating-system-market-106601>

[20] – “**What is containerization?**”  
<https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization>

[21] – “**Dockerfile reference**”<https://docs.docker.com/reference/dockerfile>

[22] – “**Welcome to Apache Maven**”  
<https://maven.apache.org>

[23] – “**Networking overview**”  
<https://docs.docker.com/engine/network>

[24] – “**Bind** **mounts**”  
https://docs.docker.com/engine/storage/bind-mounts

[25] – “**Volumes**”  
https://docs.docker.com/engine/storage/volumes

[26] – “**Docker Compose**”  
<https://docs.docker.com/compose>

[27] – **“DB-Engines Ranking”**<https://db-engines.com/en/ranking>

[28] – “**License**”  
<https://www.postgresql.org/about/licence>

[29] – “**Chapter 9. Multi-Version Concurrency Control**”  
<https://www.postgresql.org/docs/7.1/mvcc.html>

[30] – “**PostgreSQL Concurrency with MVCC**”  
<https://devcenter.heroku.com/articles/postgresql-concurrency>

[31] – “**How Parallel Query Works** ”  
<https://www.postgresql.org/docs/current/how-parallel-query-works.htm>

[32] – “**Authentication Methods**”  
<https://www.postgresql.org/docs/current/auth-methods.html>

[32] – “**SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms**“  
https://datatracker.ietf.org/doc/html/rfc7677

# Introduction

## History of web-development

The Internet, as we know it today, was not created all at once. It was a long process of small inventions that fueled a big machine. It is said that date of birth of the Internet is 1983[2]. However, it was not the global network, only several hundred hosts across the United States. This date is chosen due to the switch to TCP/IP protocol. It was a standardized way of establishing communication between networks. Before that, NCP (or Network Control Protocol) was used. It was a good way of connecting two hosts and exchanging files using FTP (File Transfer Protocol), but it lacked scalability. And scalability was needed, since people realized that it was a perfect way of communicating between different institutions.

The first browser was introduced in 1990[2]. Now the familiar shape of the Internet is forming. URLs became available, users got the possibility of accessing websites just by typing them in a convenient program, without having to write commands in a terminal. DNS system, which was Introduced in 1983[3], also allowed using names instead of IP-addresses.

After the standardized software for accessing web pages, the young network needed a standard for creating them. HTML was introduced in 1993. At first it contained only the simplest of all features. Such as basic formatting, tables, images, hyperlinks and forms[4].

Back then, the only tools available were text editors and knowledge of developers. Web pages were basically text files optimized for sending them via network. When the user made a request, the server machine sent them, then it was shown by the browser on the local machine. Everything was plain and simple. No complexity nor extra details. The rendering was happening on the side of the server. After the request was received, the server would look up the needed page and send it to the user. Nothing like browser-side rendering was not present and essentially was not needed at that time.

The server side of the application was also raw and simple. The programmer had to manually establish connections, create requests and add content to them. No frameworks or automating tools were available. The web was static, pages were sent to the clients. The first truly backend, scripting language was introduced in 1994, it was PHP[5].

CSS was introduced a little later. In 1994 it was proposed by Håkon Wium Lie, Norwegian technologist[6]. In 1996 it was initially released[7]. The invention of CSS was a milestone in web development, since it allowed us to personalize pages. It allowed us to create a soul for our sites. Of course, the possibility of styling was proposed before Style sheets. However, the proposition lacked one crucial feature, the separation between writing html code and styling.

ECMAScript, or more known as JavaScript, is a scripting language. It was initially created for web pages. Its main task was to make them more “alive”. It allowed us to manipulate pages and dynamically change them [8]. It was the first step to the browser rendering and frontend applications.

In the next chapters, every part of the web application and the process of its creation will be described. The application will be divided into three parts. Backend, computed on the server and Frontend, computed in the browser of the client. The third part will be dedicated to the tools not directly involved in creating code, but simplifying the life of the developer, such as version control system or virtualization.

## Description of the problem

Nowadays, the role of websites has become more crucial. The demand was rising gradually throughout these years, and now more and more companies are willing to have their own internet representation. Even the smallest ones often have their webpage which functions as a small portfolio and contains contact information. Sometimes, even solo private entrepreneurs have a small web page. And since new web applications have been developed continuously, instruments for creating them have evolved.

In my opinion, it is important to clarify now that tools for development are not new languages nor specifications. All programming languages and protocols have surely acquired new features, but essentially, they are still the same. Nothing completely new was introduced. Development tools only help us to manipulate what we have now and do it effectively. Usually by reducing the amount of code we need to write or by allowing us to divide our product into snapshots, to prevent progress loss in case of bad updates.

There are a lot of such methods. For backend there are several languages to choose from and each one has at least one big framework. Even though, server part of the application can be written on the raw instance of the language, here frameworks will be discussed. It is because most people will choose them, since it is the easiest way to develop an application and do it as quickly as possible. For frontend we still have only one language, which is JavaScript, but there are a certain number of popular frameworks. And each framework, whether it is for frontend or for backend, has its own libraries which help the developers solve more specific tasks. Also, there are instruments not related to writing code.

However, there is a lot of confusion and misbeliefs around web development technologies. It may not be clear why a programmer should learn new approaches to the development. Also, due to the big number of things to choose from may confuse a starting developer.

The aim of this material is to check how effective they really are. One case will be studied. With one stack of tools a full-scale application will be developed. The process will be described, and it will be emphasized on features of each technology and how it helps in the development.

## Technology stack

Choosing the right technology can be a demanding task. As it was said above, there are numerous tools and all of them can achieve the goal with pretty much equal efficiency. Usually, the pick of the program depends on previous experience, recommendation or popularity of it. Also, a developer can decide on what is more comfortable for them. The end user wants to see a ready product, what is used during the development process is often hidden behind the scenes. In this material the selection of tools will be done from the perspective of a developer who has familiarity with Java and JavaScript.

## Application architecture

It is worth noting that the backend is usually referred to as server-side of the application, which is not visible to a user, and frontend is the part that user sees and can interact with. And previously it was so. There was one program which was listening on a specific port of the server machine. When it got a request from a client it decided which html page to send. The HTML page contained content, styles and some scripts, written in JavaScript. And it was the frontend part, the small part of the application, basically the interface for user. This technique is called server-side rendering. And it is still used nowadays.

In the modern world of web development there is a new way of developing applications. Backend and frontend are pretty much standalone applications. They run separately; there are two distinct processes. They can be even run on different machines. Their only need is to be able to communicate with each other, since the backend can retrieve data from the database and the frontend needs this data to show it to the user. If previously the backend generated HTML pages and sent them as a response to the request, now the frontend app decides independently what to show to the client. Also, clients are not able to access the backend. It is built specifically for exchanging data with the frontend part. For this purpose, backend applications have built-in API. API stands for application programming interface; it is a set of rules describing how two programs should interact with each other. On the other hand, frontend can be created without creating backend, it can use API developed by someone else.

To understand how this transition happened, we need to look at JavaScript and its history. JavaScript was created 1997 as a standard for the language, it was called ECMAScript[8]. And it is a formal name of JavaScript till today. It is a scripting language. It is not complied but interpreted instead. That means that code is ran line by line, instead of compiling the whole program and running it afterwards. The interpreter is called engine. Each browser has one. The most popular ones are V8, developed by Google, SpiderMonkey by Mozilla, JavaScriptCore by Apple and Chakra by Microsoft. All of those were first created to be used in browsers and it was the main aim of the young language. To run small scripts in a browser and make html pages more interactable and “alive”. It was a part of browser environment such as HTML or CSS.

Almost from the beginning of JS there were people who wanted it to be run not only on browsers. The first server-side attempts were made since 1990s. This was an excellent possiblity because it allowed us to generate pages more carefully with a language that was designed specifically for this purpose. However, this approach did not gain much attraction among developers, at that time

However, the idea of creating web applications in one standardized language has been slowly gaining popularity. On 27th of May 2009[10], Node JS was released. Up to the day of writing this material, it remains the most widely used server-side runtime environment for ECMAScript, allowing to build backend with it. It is crucial for us, because it also allows bundlers and packet managers to be created. Such instruments help to build JavaScript application by bundling code into fewer files, optimizing it, removing redundant characters like spaces or comments. Packet managers allowed to create databases of libraries with easy access for all developers, which pushed JS further to standardization.

The language itself was evolving too. It gained more features and transformed from being a pack of variables and functions to a fully functional scripting language which can be run in its own environment, not being tied up to browsers. Till that time, it has gotten new capabilities, such as:

* Manipulating the webpages, changing their content using only code
* Reacting to browser events, such as mouse clicks or text selections
* Executing code asynchronously in the background, without blocking the application
* Accessing URL line and manipulating it [11]

The slow separation of backend and frontend was now complete. With the assistance of frameworks or without complex logic of the websites can be written. Files can be prepared and optimized for production. Libraries and frameworks can be downloaded and utilized quickly. And everything was standardized and only one language was used.

Frontend applications remain a pack of HTML and CSS code, but now they are glued by JS. Nonetheless, they still need to be delivered to a user by some server application. It can be any, now even the ones written in JS itself. After that, it becomes fully independent. It can show the client data, change content on pages and request data from the backend app. Such applications can simulate the transition between pages. They can read URL and intercept the page-changing event and instead show content themselves. Those are called single-page applications. The creation of it will be presented in this material

Another mechanism that is now widely used not only on the web, but just in software development is containerization. Containers are old technology, emerging from the 1970’s. It is an idea of running an application in an isolated environment, so nothing can impact it. The first attempts have been made in 1979, with a command called *chroot* [12]*.* It was a UNIX command which created an alteration of the root folder for a specific process to run inside of it. It then was adopted and developed further into *jails*, in FreeBSD[13]. Those are completely isolated environments, which can be fully configured and customized. They behave pretty much like small virtual machines. They can engage in communication with each other, have their environmental variables and start several processes. Only what was left is to make them easily portable, so anyone can run a container on their local machine. At that time, they were often to their original OS, because they shared the same kernel as their host OS. So, even though most of the containers were created for Linux, they still had to be configured manually, their dependencies had to be injected by hand, there was no standardization. In 2013 a tool named Docker was created. Its primary goal was to make containers portable and runnable on any machine. A standard for creating containers was being created. Docker introduced the concept of images, a blueprint for creating containers. By having an image, a container can be created. Firstly, it was available only for Linux, because of containers’ file system and structures inherited from Linux. However, they have been developed throughout the years and now they can be also used both on Windows and MacOS. The final app

So, the general structure of the application will feature backend, frontend and database. It will also be containerized. A version control system will be used. Frontend will be a single-page application, built using a framework. Backend will be built as an API for frontend, also using a framework. Packet managers and build software will be used for both frontend and backend.

## Technology stack

For frontend modern ECMAScript 6 will be used. As a framework, there will be React, since it is the most popular one for today. It will be most likely used by a developer. Node Packet Manager will be used for managing libraries and dependencies. For Building the whole application together, a Vite bundler is to be used. As a server, NGINX tool will be used as an old and reliable choice, since a frontend server does not really matter in this study, since most of them are featuring standard security and its only purpose is to send the application to the client only once.

For backend modern Java with JDK 21 will be used. Also, there is a framework, Spring and especially its streamlined version Springboot. Springboot is more modern and introduces several new features into creating a server-side web application. Apache Maven will be used as a bundler and packet manager. It also includes a compiler, so there will be no need in compiling Java code before bundling.

As non-coding tools, there will be Docker and Git. As was said above, Docker allows us to create portable containers. And it is also used in most modern web applications, so its presence in the stack will be a good addition to the simulation of professional development. Git is a famous version control system. It allows a developer to make snapshots of the program at any given time and roll back to them in case of need. It is widely used in software development.

# General Development Tools

## Version Control

Even though the development of the app is often a pre-defined set of steps, everything still can go wrong. Each feature can be implemented in various ways, and it is a difficult task to predict how it will be impacting the rest of the application. Bugs of different scales can occur. Some may crash the application straightaway; others can be quiet and difficult to spot and still create difficulties for users. In some cases, it is crucial to be able to roll back to the previous, working version of the product. Another use case for version control is when an earlier version is needed. For example, it can have smaller system requirements, or not all users might want to update their software and therefore an older instance should be available.

Multiple methods can be used to control versions. The simplest one is to create a new directory for each version, store them and access whenever they are needed. Or another variant can be to log every change in a separate file with date and time. Needless to say, both strategies are time-consuming and error prone. Developer can easily confuse files, dates or just forget about some line of code which was changed in a small, non-critical file. More advanced engineers have come with an idea of creating their own, in which each snapshot was to be stored. All those early approaches were local.

Hopefully, software dedicated to managing versions of other software will be created soon enough. One of them was RCS - Revision Control System, created in 1982[14]. Its environment was UNIX system, and it was suitable for controlling both text as well as source code files. Even though it was not the first one, it included notable changes, such as not merging the differences between files directly, but keeping them in the form of scripts, manuals which told where and how insert each change, if it was ever needed [1]. Its predecessor SCCS – Source Code Control System, announced in 1975 [15]. It was saving changes into the files but marking them. If the previous version was needed, differences were removed by a special program [14].

Today, a lot of Version Control Systems are available. Probably the most widely used is GIT. It is open-source software, released in 2005 by Linus Torvalds [16]. Till that day it is developed, updated and supported, as can be seen on its GitHub page. It is so popular and widely used that sometimes developers can learn how to use GIT, before understanding what a version control system actually is. Another feature is that it is open-source and completely free to use. All of the source code can be found in its repository, on GitHub for example. An important remark here is that it should not be confused with GitHub. GIT is a version control system. While GitHub is an online platform for storing and sharing code in repositories, created with GIT.

GIT is a distributed version control system. It means that there is no centralized system, orchestrating snapshots and changes of the app. Each developer receives a separate copy of the repository, mirroring the original one [17]. Changes they make are essentially independent and then are synchronized on a remote server, which is holding the main repository. The main advantage of that system is immunity to the main repository loss, since every one of the team can share their version, to become a new, main one.

The way GIT stores version is something in between SCCS and RCS. It stores snapshots of the application. Each snapshot consists of several objects: Blobs, Commits, Trees. Those are objects stored in GIT’s own database and are reachable by their unique ID. That snapshot is a tree, representing a file structure and blobs, each representing content of a separate file. A commit is a pointer to that structure alongside unique hash and metadata about date of creation and author. Each commit also contains a pointer to its parent (or parents). That allows us to reach all files and gives us the ability not to copy files that were not changed, since they can be accessed easily [18]. So, the snapshot contains a commit, metadata and a tree of changes, with a pointer to older commits with older changes, which remained the same in this one. That metadata can be retrieved at any time, and makes it easier to coordinate the development team.

That pointer structure makes the GIT very flexible in its ways of manipulating software. Basically, it makes commits “movable”. New branches can be created, commits can be transferred between them. Commitments can also be deleted. Different branches may be merged into one another. However, during merging conflicts can arise. For example, let us say that there is file named *Main.java* in both branches. Two branches can have different information about what is written on the lines 10-20 of that file. In such case merging will be stopped, and GIT will raise an error, indicating what have gone wrong and where. After all of the conflicts are resolved, merging can be continued from the point of the error.

## Virtualization

Most, if not all, web applications are developed for commercial use. They are meant to be an interface for the user to communicate with the business. That means they need to be deployed somewhere. Usually, it is a dedicated server, a powerful machine meant to be working 24 hours a day, so each user can access the service without any interruptions. So, the application is developed and run in two different environments. Sometimes those machines can differ a lot. They can have different hardware, different operating systems and different software. And such details must be taken into account. For instance, Linux-based operating systems use forward slash “/” as a path separator while in Windows it is backwards slash “\”. Another distinction is that filenames in Linux are case-sensitive, while in Windows they are not. So, for example, text.txt, Text.txt and TEXT.txt will be considered as the same file in Windows, while not in Linux. That can lead to a problem when a programmer makes a mistake in their code writing the name of a file in a different case. In Windows it would work just fine, but after the transition to Linux the app would not work. And such a case is possible, because Linux is considered to be the most popular OS for servers [19]. So, as everything said above suggests, we need a way to run an application similarly on varied hosts. And the most popular solution nowadays is Docker.

Docker is software which allows us to pack applications into so-called containers, but what is a container?   
Container is a virtual environment, a simulation of file system of a chosen OS. They are pretty like virtual machines. However, containers are much more lightweight, they do not simulate a whole computer with hardware, and share the kernel with the host OS[20]. The program running in such an artificial file system can apply all the commands available in the simulated OS. It may use shortcuts and file naming conventions belonging to this OS. Also, everything withing the container is isolated from the rest of the machine. The application executed inside of it cannot access any resource that is not allocated to the container. Therefore, providing additional security, since containers can be used to run potentially harmful software.

Another great thing about containers is that they are fully isolated. As it was said previously, they have no access to any system files, if not specifically set to. That makes them great for testing purposes. Since the environment of the program can be made completely sterile, or it can be adjusted to imitate the production environment.

Another notion, accessible in Docker are images. As it was said in the introduction, images are like blueprints, from which the actual containers are created. They can also be compared to the classes in OOP, while containers will be objects here. Images contribute to the portability of Docker containers. Thankfully to them, there is no need to send the actual containers between developers. Everything needed is an image and the Docker engine itself. In addition to that, there is a web platform for storing docker images, both for public and personals use, in the cloud.

Images are created using Dockerfiles. Those are text documents containing an array of commands. Docker engine parses them and generates an image accordingly [21]. They do not have an extension by default, but it can be created. Often it serves as an indicator of the purpose of a specific application’s snapshot. For example, *Dockerfile.dev* or *Dockerfile.prod* of Their usual location is the root folder of the application. One big, multi-service application can have several such file. In our, there are two of them for frontend and backend respectfully, Below is an image of the dockerfile used in the backend part of our application.

A screenshot of a computer

AI-generated content may be incorrect.

Commands here are written in upper-case. Each of them specifies a separate action that should be taken while creating an Image.  
FROM – describes parent image. That means, our image will be based on **maven:3.9.9-sapmachine-21**. It is an official image, marked by Docker and accessible via dockerhub. Maven is a building tool for Java projects [22]. So, it is needed to run the backend which is created with Java.  
WORKDIR – specifies the current working directory, where all of the next commands will be run. In our case it is /backend, which stands for directory “backend” in the root directory of the container.  
COPY – Copies files and directories from the host machine to the container. In this example path “ . “ is copied to the path “ . “, which means that all files from the root backend directory will be copied to the /backend in the container.  
EXPOSE – lets host machine to access the specified port of the container. In this case it is 8080, since backend is configured to listed on this port.  
RUN – executes build commands in the container. In this case, it is the default command for maven to build the application. It will compile source code, run tests package everything and prepare for executing.  
CMD – default commands which are run when a container is started. Here it will be “mvn spring-boot:run”, which runs the application. Springboot here is the tool for spring framework which speeds up the configuration process of the application.  
For the full list of commands, official Docker documentation can be accessed [21].

Similar file is in the root directory of the front end. So, technically, frontend and backend are two separate applications which should be able to communicate between each other in the process.   
While remaining isolated, it may be a complex task to exchange any data. There are two ways here. The first one is sending requests to each other via virtual network created and managed by docker engine. Each container is a host there and is assigned its own IP address. It can be either IPv4 or IPv6 [23]. Alternatively, they can communicate like default applications. This is why exposing port is needed. This port will be bound to a port of a host machine and two containers will be “talking” to each other like two separate pieces of software. This is the method used in out application. Also, containers can make external requests. Those rely on host machine internet connection.   
In the example application there is a third container. It carries the database. Just the database application installed in the container and a persistent storage mounted to it.   
And here, with three services already, we come closer to one of the problems for modern web developers: “multi-container applications”.

However, before talking about the efficient way of dealing with multiple containers, another notion should be introduced. As was said previously, containers are fully isolated from the host system. If it gets deleted, all the files that were inside are lost as well. While it can be okay for some applications, such as static website hosting or one-time tasks, usually we need to store the results of the execution in a persistent place. Docker gives two main solutions to that matter.   
The first one is bind-mounts [24]. They allow a developer to bind a directory of the host machine to the directory in the container. Every change in the container will be reflected on the host and vice versa. The most significant advantage of this approach is that each piece of data can be accessed by the developer without any interruptions. However, if a container is meant to be run on different devices, it can become a problem, since each host should have the exact same path with same files and permissions. Also, if a program needs to use the same files, those must be transferred too.   
The second one is volumes [25]. They are file storages completely managed by the Docker itself. They are created by a user, or by the engine while starting up a container. Volumes are bound to the containers, much like the bind mounts are, except for the fact that their contents cannot be accessed by the host machine. Thus, they are more secure, because data inside cannot be corrupted, and cannot harm the host machine in case of any malware getting inside the container. Also, volumes can be easily transferred and accessed by other containers on other hosts, without worrying about permission or correct path. Like that, a volume can be run on both Windows and Linux without any obstacles. Backups are much easier too, only the volume file itself needs to be copied.

The docker engine offers an extensive interface for managing each individual container. However, configuring and staring up each one with a separate, long and complex command can be time consuming. Here is where docker-compose steps in. It is a tool which helps with defining and running multiple containers [26]. For it to work, it also needs a configuration file. It is a YAML file, while dockerfile has no extension. Below is a part of our compose file, defining backend service.

A screen shot of a computer

AI-generated content may be incorrect.

Here we can see several properties, that backend object has, let us walk through:  
**Image** – specifies the image used to create the container  
**Build** – specifies the location of the docker file, to create a new image if it was not found  
**Working\_dir** - much like in the usual dockerfile, it sets the working directory where commands will be executed and where the application will be run. To avoid confusion and errors, the setting from compose file has a higher priority and overrides its peer from the dockerfile.  
**Environment** – here is a list of all variables that will be embedded into the container as environment ones. Their values, e.g. ones in curly brackets, are set in a separate file. It is done for better management, since the same variable can be used by different services. For instance, the ones related to the database are used both here and in the respective container.  
**Volumes** – here volumes are specified. The name of the volume is followed by a path that it will be mounted on in the container.  
**Ports** – define which host port will be bound to each container port. Here we need only one, and it is attached to the same port on the computer.  
**Depends\_on –** sets a list of containers on which the current one is depended on. The Docker system will wait till all the dependencies are started, before starting up the current one. In this case, backend service will be brought up only after the successful launch of the database service.

Another way of using several containers is parallel computing. The container swarm could be used to set different tasks for each one. One of the use cases is matrix multiplication. Since this operation can be easily split up into several smaller operations. Let us say, we need to multiply matrices A and B. Here is a formula for it.  
 Each container can run a script that receives a portion of rows of A and the whole B. T het it, we can divide *n* by the number of containers we wish to run. Then it computes its own respectful portion of result C and returns it. Then, a special script can be created. For example, bash or python script. It would receive a matrix, split it into parts, run them into different containers and then assemble the whole thing together.

Concluding about the Docker, we can say that it is an optimized and quick approach to a lot of problems during the development process. The application can be shared, transferred and run smoothly on different platforms with different hardware and operating systems. It manages data and provides secure access to it. Volumes can be easily backed up and transferred. The containers are isolated, which provides an additional level of security. Any harmful software inside of them cannot hurt the host machine.

## Database

A good and reliable database is one of the most essential parts of most applications. Everything related to users and their data can be stored there. Starting from their accounts themselves, up to the likes and relations between posts. However, the database is only a file stored at the machine. Like a text document, it contains something, and similarly to them it needs a program to operate it. For text documents such pieces of software are text editors, for databases: database management systems. Most of them use traditional relational models. They are OracleSQL, Microsoft SQL Server, MySQL, and PostgreSQL. The first two are being developed by private contractors, while the last two are open-sourced alternatives. None the less, those four are the most popular DBMSs at the moment [27].  
Relational databases are still very popular, but with time alternatives have arisen. The most popular one is MongoDB with its focus being to storing JSON documents, instead of tables with SQL queries to access them. And it was a very wise move from the behemoths of the industry to spread abilities of their developed DBMS’s. As for now, all of the four major players are multi-model. That means there are several possibilities to store the data, not only relational SQL one. The most multi-modeled is Oracle’s database, the second after it comes PostgreSQL [27]. However, the advantage of the latter is being open-sourced and released under license similar to BSD or MIT [28]. Therefore, PostgreSQL can be used completely free, while offering similar functionality to the paid, enterprise variants.   
That is why PostgreSQL was chosen for this demonstration. Below it will be discussed, what this database engine can offer for a web application, such as social platform.

Firstly, it is necessary to understand what data, structure and scale will be needed from the database engine. Since the demonstration app is a social media, its demands can be different for each case. However, they are usually extensive. There are a lot of different types of data. It can be structured in different ways. Already saved information might need to be rewritten into a completely different form. As the application grows, storage will be extended and scaled both vertically and horizontally. Additionally, real-time performance becomes critical, especially for timelines, notifications, and messaging systems. Data consistency and integrity must be carefully balanced with availability and responsiveness. Features such as full-text search, geolocation and recommendation engines may also introduce complex query patterns. This means the chose database should not only handle transactional workloads but also support analytical capabilities and flexible indexing. Ultimately, selecting the right database model will directly affect scalability, maintainability and user experience of the entire application  
  
We have to store various entities. They are users, posts, comments. Also, their relations should be present. Such as likes or subscriptions. All of those are pretty simple and can be handled by PostgreSQL by using a standard relational model, along with strong constraints and foreign keys. However, the possibility of using NoSQL extension can be utilized by storing something more complex. For instance, logging of user-related events. They can be various, and it would waste a lot of space to create separate tables for each type of event. Instead, they can be written into JSON documents. They are famous for being flexible regarding their contents. Also, if existing table needs to be drastically changed. For example, implementation of custom reactions to a comment may demand storing different images and each comment can have a different number of different reactions. It would spike up the complexity of the table. Instead, it can be rewritten into JSON format.   
  
In our application there will be several relational tables, representing users, comments, posts and their relations between them. Users will be able to subscribe to each other. That means, another many-to-many table. A similar table will be for likes, which users are allowed to leave under each post. It is a possibility of a very heavy read-write workload. Since at the same time, concurrently, database will receive multiple requests on reading and writing data and they should not block or interfere with each other. To solve this problem, PostgreSQL uses Multi-version Concurrency control. It is a complex technique, making it easier for the database to handle such situations. At its core, it means that at any given time, each transaction does not see the actual database, but its snapshot, made some time ago [29]. This works as a barrier preventing unstable data from being spotted and used by different queries.   
  
To understand how MVCC works in more detail, it should be broken down. Firstly, each transaction gets its own ID, called XID. It is a global variable that gets incremented with each new transaction. Also, each row in the table has hidden metadata. Two columns, which are called *xmin* and *xmax*. *Xmin* is used for insertion transactions and *Xmax* for deletion and updates [30]. For transaction to be able to see a row it must satisfy the following condition:

Where *Rj* is a row and *Ti* is transaction with ID = *i*. And the transaction **must be committed**, before the transaction ends and finally commits data, no other transaction can see the current row. So, basically, any transaction can see only the rows that were committed previously () and their deletion or change has not been committed yet (). Thankfully to that feature, readers and writers do not block each other. Since each transaction sees a snapshot of available rows based on its ID, readers can check the information that was available during the request. There is not need in waiting till the last insertion commits or the next row gets deleted.  
Also, even though *xmin* and *xmax* are hidden and considered to be metadata, they can be easily obtained using SELECT query.  
   
Same as the index of current transaction can be accessed

In addition to that, as the amount of data stored will be growing with each new user, in perspective it can slow down the execution time for CRUD operations. The more users and consequentially data will be registered the slower it will be getting. PostgreSQL delivers a nice and efficient solution to that. Special indexes can be created for each table. It uses B-tree indexing. It is a default indexing method in PostgreSQL. Balanced Trees are widely used data structures for storing data that needs to be accessed frequently. They offer a very fast search operation complexity, which equals O(logX n), where x is a number of children of each node. A table gets indexed automatically, whenever a primary key or a unique constraint is created. Alternatively, an index can be created using a special command  
***CREATE INDEX idx\_name ON table\_name (column);***  
This method of indexing performs especially great during queries with a condition or range.   
For instance, WHERE id = 54 or WHERE *age* BETWEEN 20 AND 30.  
When engine sees something similar, engine traverses the tree from the root to the leaf with the needed value, by performing a binary search.

A database system is usually considered to be something heavy and bulky. And it can be true, especially for big and grown services. The database can experience a heavy workload and not only take up a lot of space in RAM but also puts significant pressure on CPU.   
PostgreSQL handles such scenarios quite well. Without any drop in functionality or inventing smaller, lighter versions of the program. The solution it provides is called Parallelism. In essence it means breaking up one large query into several, smaller sub-tasks. Those sub-parts then run in parallel on different CPU cores.   
When a query gets received, the engine decides which way of execution will be the fastest. Then, if a parallel strategy is considered the most optimal, a special query plan will be created. Alongside that, several worker processes are created. Each of them will handle a chunk of data and return partial computation results. The leader process coordinates their work. If there are not enough workers, leader can perform their tasks, essentially becoming a new worker, to speed up the execution of the query. When all of the workers have finished their execution, the leader revises their output and combines them into one large piece of data, which is then sent as a result of an query execution.   
The maximum number of workers per one query is defined by a global setting *max\_parallel\_workers\_per\_gather*. The general number of parallel workers for the whole app is set in both *max\_worker\_processes* and *max\_parallel\_workers* [31].   
Those are general settings, they can be set by the command SET and viewed with SHOW.   
Not all operations can be executed in parallel. However, the most essential are parallelized. Those are sequential scans and aggregation queries. Therefore, the process of looking up a row can be significantly speed up. Even if no index is available at the given table and the engine must parse it whole. Merge joins and sorts can be executed in parallel too.  
In the case of social media applications, this feature can be really helpful. For example, such entities as likes or subscriptions may not have their own index and even one user can have millions of them. That would make it very slow to check if someone is subscribed to someone else.   
In addition to that, CPUs specialized for server work and usually do have a lot of cores. Several new models are passing a threshold of 128 cores.

Another crucial thing in storing user data is security. PostgreSQL offers several layers of protecting data. The first line of defense here is authentication mechanisms. PosgreSQL supports up to 11 authentication methods. They range from a known password method to specific only to FreeBSD method based on its authentication framework [32]. The recommended way for password-based authentication is scram-sha-256. It uses the SCRAM-SHA-256 Simple Authentication and Security Level procedure. The key security feature here is that no password is ever sent in plain text. Instead, the client sends a username, and server replies with salt and number of iterations for PBKDF2. Then the client performs following actions:  
After those, *ClientProof* is sent to the server where it is verified. Also, each message includes a random string, called nonce. It is used for prevention of replay attacks, since for each session a new nonce is generated randomly.   
In addition to that, all of the communication between the client and the server can be conducted via SSL/TLS. Furthermore, it is strongly advices to use TSL for the abovementioned authentication method [33].

Also, another tier of security is protection against wide-spread SQL injection attacks. For that purpose, prepared statements are used. Those are essentially SQL queries which act more like functions. They accept parameters and inject them into a pre-defined SQL code. To create one, PREPARE command should be used, to execute one – EXECUTE. The command below, shows an example of creation of such entity  
***PREPARE get\_user(text) AS   
SELECT u FROM User u WHERE u.username = $1;***  
That command will create a prepared statement that can be run using *get\_user(username)* command.  
Now let us talk about, how does it protect our database. For instance, consider the following Java code  
***String qry= "SELECT u FROM User u WHERE u.username= '"+ usernameInput+ "'";***It defines a query using some value from *usernameInput* variable. This variable is usually obtained from the frontend part of the application. Below is an example of what it can contain inside of it  
***String usernameInput = "' OR 1=1 --";***  
Usually such strings are passed so low due to the improper validation and sanitization procedures in the frontend and backend. Now the database will receive the following query  
***SELECT u FROM User u WHERE u.username = ’’ OR 1=1; --'***  
As a result it will return all of the users, regardless of the username. However, if a prepared statement is used, the database will receive a query like   
***SELECT u FROM User u WHERE u.username = "' OR 1=1 --"***And it will result in an empty list, since users are certainly not allowed to have usernames like *' OR 1=1 --.*

To conclude, about the database. We can see here that our example, PostgreSQL is an excellent solution for web applications. It is open-sourced, and thus completely free. However, it still provides an enterprise level of functionality. It is very flexible and allows storing information in a vast number of data structures, making it a multi-model database. The implementation of MVCC provides a great throughput during extensive number of CRUD operations. While B-tree indexing lets accessing the data fast even in large datasets. Security is deeply embedded, and modern authentication procedures are implemented, such as SCRAM. It ensures that the personal user data and credentials are safe. On the other hand, prepared statements provide protection against SQL injections. These points are making this DBMS a fitting choice for a big and secure social media web application.

# BACKEND

## Language and Framework

In the context of a web application, the backend is the part responsible for handling user requests. It mainly manipulates user data, decides which user to sign in and register, and what data will be saved to the database. Java was performing quite well for this task. It can be called battle-tested in this case. The robust functionality it provides covers all of the needs a developer might have.

For this study, it was chosen due to several reasons.  
First, Java is a well-established and rich history as a reliable choice for constructing enterprise-ready applications. Ever since it was developed in the mid-1990s, it has emerged as one of the widely used programming languages around the world, particularly within large enterprises that require scalable, secure, and maintainable software programs. One of the greatest advantages of Java is its huge and mature ecosystem, where there is a wide range of good frameworks such as Spring, Hibernate, and Java EE and many libraries that provide support from web programming to data processing and distributed computing.  
In addition to its technical strength, Java enjoys a vast, active, and seasoned global developer base. What this means is that organizations have better access to seasoned professionals who already know Java, which reduces the cost of acquiring talent as well as the development cycle. Furthermore, the sheer availability of learning materials translates into everything from extensive documentation and web-based training to books and tutorials, making it easy for new developers to learn the language and be productive in no time.  
These factors give Java a significant edge compared to relatively newer programming languages like Golang. While Go offers performance benefits and simplicity, the maturity, ecosystem, and widespread industry adoption of Java often make it the more pragmatic choice for complex enterprise-grade systems.  
Java is statically typed and compiled, and it plays a very significant part in backend system stability and reliability. Static typing forces type correctness at compile-time, reducing the likelihood of runtime errors and making codebases easier to comprehend and refactor after a period of time. As opposed to dynamically typed programming languages such as Python or JavaScript, whose type-related flaws can only be revealed at runtime, Java provides early feedback upon development so that programmers have a chance to catch issues beforehand. Also, being a compiled language running on the Java Virtual Machine (JVM), Java is faster and more memory-efficient at execution time than interpreted languages. It is less abstract than Python or JavaScript, offering more direct control over the performance and behavior of the system. That makes Java a particularly good fit for developing high-throughput, long-lived backend services where consistency, scalability, and long-term maintainability matter most.

Then, there is about framework. Spring was established in Java ecosystem as one of the most flexible and robust ones. In particular, it is useful for building scalable and maintainable web applications. Originally introduced in early 2000s, its main purpose was to simplify the development process of Java. Specifically, by making it easier to write code and automating several tasks, such as dependency injection or managing routing and https requests.   
Its flexibility is mainly expressed through not being a single monolithic entity. Instead, it can be understood as a collection of modules, each of them representing a crucial feature. There is Spring Core module which orchestrates all of the others. For example, if there is an application which involves heavy communication with a relational database, Spring JPA is a good tool of use. However, if an application does not require such possibilities, this package can be omitted. So, there is no need to install one large and bulky package. Such modularity allows developers to combine a set of tools fitting perfectly for a particular project.   
In addition to that, Spring ecosystem offers another extension, further streamlining the process of the development. It is called Spring Boot. Its main purpose is to create the base version of the project. The default and recommended configuration is used. Tomcat is embedded as a default server for holding the application.   
Considering all of the above, the combination of Java and Spring will be perfect fit for a social media application as an example and most of the cloud applications at all.  
  
Now, let us talk about, how exactly Spring framework will be helping in creating a cloud software. The overall architecture of the whole social network falls under the definition of MVC, which stands for Model View Controller.  
**Model** - here represents the core data and logic of the application. It is a piece of software which handles retrieval, storage and processing. In our case, Model includes the whole database system and a part of the backend. This part will be curated by Spring JPA module. It will assist in creating Java objects which then will be mapped on to the database tables. For instance, User object, which will have properties like username or password.  
**View** – is the part user interacts with. It is the interface of the application, all of the data that needs to be shown is shown here. In our application Frontend is taking this responsibility. It will be in the form of an interactive webpage.   
**Controller** – is a middleman between two other parts. It handles user requests, makes sure inputs are correct. The model is updated on behalf of the controller. Similarly, the view shows only that information that the controller told it to show. In our case, the Controller takes part in the backend, responsible for requests. In frontend, controller can be seen as the code that verifies and sanitizes user input and sends requests to the backend.   
The backend will be built in the form of REST API. REST stands for Representational State Transfer. It is a set of principles for building applications on the network. The core principle of the REST is its statelessness. It does not store any session data, everything that server might need should be carried inside of the request. As a response, the server returns the representation of the resource, not the resource itself. REST offers a great opportunity for creating API’s that will be used by several of the clients. In the context of social media application, those several clients may be users in different locations. Also, servers for frontend may be placed in various places, and they can all reach the same backend server. That feature comes in hand in case of unexpected circumstances, when one or several of backend serves go down. In Spring, the module responsible for implementation of the RESTful API is Spring Web and especially its sub-module – Spring Web MVC. They simplify the creation of REST endpoints, and handle request directing.

Dependencies used in this project are related to both Controller and Model parts of the application. They all belong to the spring-boot-starter list. This is a list of dependencies which use default Spring modules, alongside with some additional software. It is already pre-configured. Below is a list of spring-related dependencies and what do they implement.  
**Spring-boot-starter** – Is a core for Spring Boot dependencies, it manages them all and contains Spring Core inside. It also brings in the fundamental Spring Boot capabilities of auto-configuration, application context management, and logging. It enables the overall infrastructure and bootstrapping of the application, impacting every layer—Model, View, and Controller—by ensuring the app starts with sensible defaults and a bare minimum of setup.  
**Spring-boot-starter-web** - This module is a must for the Controller layer. It has Spring MVC, by which you declare REST controllers using annotations like @RestController, @RequestMapping, and @GetMapping. It also includes an embedded Tomcat (or Jetty/Undertow) server. Therefore, the application can be run alone without any external servlet containers. It does HTTP routing, request/response mapping, and JSON serialization with Jackson, basically the basis of the REST API.  
**Spring-boot-starter-jpa** - This starter facilitates the Model layer through ease of working with a relational database using the Java Persistence API (JPA) and Hibernate as the default implementation. It provides robust abstractions like CrudRepository or JpaRepository to interact with entities with minimal boilerplate. You can define your domain model as Java classes annotated with @Entity, and Spring Data will take care of CRUD operations, query generation, and transaction management. It is crucial for our application, since there will be several database entities. For example, for users, comments, likes, posts.   
**Spring-boot-starter-validation** - Controller & Model Layers  
This module enhances input validation at the boundary of the Controller and Model layers. It adds Hibernate Validator, which is used to annotate your data transfer objects (DTOs) or entities with constraints like @NotNull, @Size, or @Email. It integrates well with Spring MVC to automatically validate incoming request payloads and return helpful error responses when validation fails. It will be mainly used by User and Post models, since they have required fields, which may come empty. It is worth noting that this application is built by one person. Thus, it might not be clear why there should be validation on all of the layers. Solo developers can assume that the data coming from the frontend is correct, since they know that they have implemented a validation mechanism there. However, an additional layer of security is never excessive. Moreover, the development of the application may be continued in the future by another person or team. In the case of insufficient validation, it may cause serious problems and security threats.   
**Spring-starter-security** - Cross-cutting (Controller & Infrastructure):  
This starter includes robust security features that typically operate at the Controller level and above. It defaults to endpoint securing, authenticates and authorizes, provides CSRF protection, and integrates well with OAuth2, JWT, and custom security filters. You can easily secure REST endpoints and define role-based access using minimal configuration with annotations like @PreAuthorize. And here again, it is who is sending requests, and therefore, it might seem like there is no need in setting CSRF boundaries. However, it is not known what will happen with the application in the future. Also, such weakness can be found and successfully exploited by hackers.

Concluding, short, Java and the Spring framework make an excellent combination for creating a contemporary, scalable, and secure social networking application. Java maturity, performance, and standard usage in enterprise environments provide long-term stability and exposure to an enormous number of experienced developers. When it is, the modularity of Spring—specifically with frameworks like Spring Boot, Spring Data JPA, Spring Security, and Spring MVC—facilitates high-speed API development of RESTful APIs, efficient data management, and robust user authentication, all with minimal boilerplate. On top of that, intrinsic validation, testing, and microservices support enables developers to more easily build maintainable and feature-rich applications that scale as the user base grows. That's a huge advantage. With these strengths combined, Java and Spring create the ideal package of flexibility, potency, and reliability, as a smart, forward-thinking choice for a social networking site.

## JWT Tokens

One of the basic principles of a REST backend is that it must be stateless, i.e., there must be included in every client request to the server all the information necessary to know and process the request. The server should not retain any session state or client context from one request to the next. But this statelessness is contrary to one of the most elementary needs of most social networks: that there be a login and authentication mechanism, where users wish to stay "logged in" as they use the site.  
Traditionally, web applications have managed login through server-side sessions. Here, once a user logs in, the server generates a session ID and stores corresponding session data (e.g., user ID, role, or preferences) in memory or a disk cache. A session cookie with the session ID is sent to the client, which is included with each subsequent request. While this pattern is handy and still dominant, it violates the stateless nature of RESTful design because it requires per-user session state stored on the server.  
This model introduces a set of problems, particularly in scalability. Since the server has to retain session information for all active users, it consumes memory and processing resources in direct proportion to the number of active users. Therefore, any one server instance will only be able to handle a finite number of users before the performance is adversely affected. In order to accommodate more and more users, developers must scale up (deploying on more powerful and costly hardware) or scale out (dispersing sessions over large numbers of servers using session replication or sticky sessions), each of which adds complexity to the infrastructure.  
Also, session-based systems are less fault-tolerant when distributed as in cloud or container-based environments. When the session data is not provided or propagated between the nodes, users will experience inconsistent behavior as requests are distributed across various servers. This ruins the very elasticity and resiliency expectations of the cloud-native application.

All these reasons cause most modern RESTful APIs—especially those that operate large-scale systems like social networks—to implement token-based authentication schemes, such as JSON Web Tokens (JWTs). These tokens encapsulate user identity and permissions in a signed, encapsulated bundle, sent with each request. The server doesn't need to store session state in any form, thereby keeping REST's statelessness while still enabling secure user authentication. It is a more scalable method because it enables the requests to be processed independently by any instance of the server without using shared memory or synchronizing sessions.  
In summary, while session-based auth remains common, especially in legacy web applications, it is a scalability and maintenance issue that is at odds with RESTful design. Stateless, token-based auth is becoming increasingly popular in designing new social networking backends because it is more RESTful and permits greater, more distributed collections of users to be handled more easily.

The JWT auth will be implemented in the example application. Now let me explain how exactly those tokens work. A JSON Web Token is a compact, secure string that represents a set of claims transferred between two parties. It consists of three parts: header, payload, and signature. Since JWT is a string, here is how it looks like ***<Header>.<Payload>.<Signature>***.  
The header contains metadata about the token, notably the signing algorithm and typically the token type, usually specified as "JWT". The header is encoded in Base64URL to form the first part of the token. The payload comes next and contains the actual claims, for example, statements about an entity and other information. There are three kinds of claims: registered claims like ***iss*** - issuer, ***sub*** - subject, ***exp*** - expiry date, and ***iat*** - issued at. Public claims that are designed to be collision-resistant; and private claims custom fields agreed upon by the parties.   
Like the header, the payload is Base64URL-encoded but not signed, meaning its contents can be read by anyone who possesses the token and therefore sensitive information cannot be stored here.   
The final element is the signature, which protects the integrity and authenticity of the token. It is created by encoding the header and the payload, concatenating them together with a dot, and signing this material with the given algorithm and a secret key (in symmetric algorithms like HS256) or a private key (in asymmetric algorithms like RS256 or ES256). As an example, in HS256, the signature is an HMAC of the message with SHA-256 and a secret key shared. Mathematically, HMAC is nested uses of the hash function with constant padding and XOR combined with them. In RSA-based signatures, signing is modular exponentiation with a private key, the SHA-256 hash of the message, to the secret exponent modulo an extremely large integer, RSA modulus. When the signature is created, it is Base64URL-encoded and included in the header and payload to form the complete JWT.  
When the server verifies a JWT, it may validate the signature against the secret or public key so that it can determine whether the token has been tampered with or not and was indeed issued by a trusted source. JWTs are widely used in stateless authentication since they allow the server to authenticate requests without maintaining a session state, and so all the server needs are contained and encrypted in the token itself.  
Although very useful in distributed and scalable systems, JWTs must be used with caution. They must be transmitted over HTTPS, safely stored on the client, and managed in a proper manner with expiration and refreshing mechanisms to prevent abuse.

To conclude JWT’s, the shift from session-based to token based authentication, particularly with JWT’s, addresses the intrinsic limitation of stateful server designs in RESTful systems. JWT’s make authentication and authorization stateless, scalable, and cloud-native application-friendly. Through their bundling of identity and claims into self-contained, verifiable tokens, they eschew centralized session storage and facilitate frictionless request processing between distributed services. However, while JWTs achieve efficiency and scalability, they also necessitate careful implementation routines for security purposes, such as the requirement to use HTTPS transmission, storing in secure clients, proper expiration, and token refresh strategy. Employed properly, JWTs achieve balanced RESTful design considerations with secure, user-centric authentication.