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

### JWT Tokens

### Packet manager and build tool

### Connection with the database

### Accepting connections from the frontend

### Authentication and Authorization

## Front-end

### Language and Library

### Packet manager and build tool

### Creating styles

### Getting data from the backend

### Routing between pages

### Managing global variables

## Conclusion

# 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>

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

[34] – **“Spring Framework Documentation”**https://docs.spring.io/spring-framework/reference

[35] – **“JSON Web Token (JWT)”**<https://datatracker.ietf.org/doc/html/rfc7519>

[36] – “**Top 20 Java Applications in Real World [2025]**”  
<https://www.geeksforgeeks.org/blogs/top-applications-of-java-in-real-world>

[37] – “**Introduction to the Build Lifecycle**”  
<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

[38] – “**Spring Data JPA**”  
<https://docs.spring.io/spring-data/jpa/reference/index.html>

[39] – “**JPA**”  
<https://docs.spring.io/spring-data/jpa/reference/jpa.html>

[40] – “**Getting Started**”  
<https://vite.dev/guide>

[41] – “**Poll Results: Popularity of CSS Preprocessors**”  
<https://css-tricks.com/poll-results-popularity-of-css-preprocessors>

[42] – “**Introduction**”  
https://zustand.docs.pmnd.rs/getting-started/introduction

# 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 robust and stable database is an underlying piece of nearly every modern application. It's the locus for storing and managing all manners of structured data. In social sites, for example, databases store user profiles, authentication credentials, posts, likes, comments, and even user-to-user or item-to-item relationships. Nearly any data that needs to outlast more than one session or request exists in a database.  
However, a database itself is just an organized file on a machine's storage system. Just as a text file requires a text editor or word processor to be read, edited, or accessed, a database requires an application program front end to query and administer it. This is the work of a Database Management System. A DBMS is the intermediary that lies between raw data and applications using it, taking care of everything from storage and retrieval of data to security, indexing, and concurrency control.  
Many contemporary DBMSs are relational, and they organize data into tables with rows and columns. The most popular relational DBMSs are Oracle Database, Microsoft SQL Server, MySQL, and PostgreSQL [27]. The first two, Oracle and SQL Server, are commercial and were developed by private firms. MySQL and PostgreSQL are open-source and are supported by active developer communities. Despite their different base architectures as well as licensing models. These four systems dominate the database space and are industry standards for building scalable and reliable applications.  
Each of these DBMSs has both strengths and weaknesses. Oracle and SQL Server are typically used within corporations for their extensive tool sets and broad support, while MySQL is famous for its speed and elegance, especially for web development purposes. PostgreSQL is renowned for conforming strictly to SQL standards, having strong extensibility, and supporting features such as full-text search and user-defined data types.  
Generally speaking, the choice of a suitable DBMS is a significant design decision. For the majority of Java-based backend applications PostgreSQL is often a leading contender due to its rich feature set and open-source nature. Especially those that prioritize reliability, regulatory compliance, and open standards.

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 abominaly 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. This means 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 help 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 such as 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.   
For further reading about Spring framework and its capabilities, official documentation can be consulted [34].

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 is 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.  
For further reading about JWT’s the official page on datatracker can be consulted [35]

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.

## Packet Manager and Build Tool

Since Java grew over time, project complexity and size have increased by leaps and bounds. While single applications in the early days of Java can be created and serviced using the bare minimum tools present within the JDK (Java Development Kit), today's software development calls for much higher-level infrastructure. Java is a popular language, used in an impressive range of applications, from enterprise-class systems and scientific computing to games development and embedded systems. Classic examples are Minecraft, an internationally famous sandbox video game that was first developed in Java, and Apache Tomcat, a flagship web server and servlet container used extensively around the globe.  
Further, Java's influence extends to other modern programming technologies. Kotlin, started as a less verbose but more expressive language to execute on the Java Virtual Machine (JVM), now has widespread use in developing Android mobile apps. Java and JVM-based languages are also used to develop IoT solutions, web servers, desktop GUI apps, and cloud-native microservices [36].  
As such projects grow and complexity, they become increasingly reliant on large numbers of third-party libraries, frameworks, and APIs to deliver functionality, performance, and integration features. To manually seek out, download, install, and maintain all these dependencies would be appallingly inconvenient, buggy, and time-consuming, especially when dealing with transitive dependencies.  
In response to this problem, strong build automation and dependency management tools are used by the Java community, one of which is the Apache Maven, the most popular and most widely utilized. Maven enables developers to automate building, compiling, testing, and packaging Java applications and controlling dependencies with less human interaction. Through bundling project configuration into a solitary pom.xml file, developers can produce reproducible builds in different environments and teams.  
While Maven isn't the only tool of its type Gradle being a very popular alternative with more flexibility and performance tuning. On the other hand, Maven is a solid, stable choice with a strong community of support, especially for projects that can leverage its convention-over-configuration approach. For the project at hand, Maven will be used as the go-to tool to make the build process easier and use external dependencies to their best advantage.  
  
Now let us discuss which advantages does maven offers exactly for a developer.   
First of all, maven is a build tool. A build tool automates and optimizes the process of compiling source code, running tests and packing the whole application. Maven is one of the most developed tools for JVM-run programs, that means it can be used for Kotlin-written applications too. While building the application, Apache Maven has its own lifecycle [37], described below:  
**Validation** – the project configuration and file checking  
**Compilation** – compilation of the project’s source code  
**Testing** – testing of the code using unit test framework that is used in the project  
**Packaging** – packaging the compiled code into a easily distributable format such as .jar  
**Verifying** – the second pack of tests is run  
**Installing** – the package is installed into the local maven repository, so it can be used as a dependency for other projects.   
**Deploying** – the package is copied to the remote repository  
So, basically, maven runs all of the tasks that are usually run by a developer, during building process. To speed up the building process, Maven supports multi-threaded builds. It can be triggered by using *-T* flag.   
*mvn -T 4 clean install*  
This means the build operation will be carried out using 4 threads, if such resources exist on the system. Using parallelism in this way can have significant build time reductions, especially when operating on multi-core systems. Besides parallel execution, Maven also offers a helpful feature in the form of incremental builds. In the above case, we have added the clean command, which will instruct Maven to delete all the previous compiled files and redo the project from the start. This is useful if you would like to have a fresh new build with no faulty or previously compiled artifacts.  
However in most day-to-day development scenarios, no such complete rebuild is needed. Maven will scan the source files of the project and their dates if the clean phase is bypassed and figure out which parts have been modified since the last successful build. It compiles and processes only the updated modules or classes without even touching anything that hasn't changed. Incremental build significantly increases efficiency, particularly for complex and large codebases, in which rebuilding each file from scratch would be abominably time-consuming. Since most changes in a development cycle will typically affect a small number of files, incremental builds allow developers to work faster, test faster, and remain productive without wasting the time.  
Also, it is needed to say that this feature has to be used with caution. Especially if any of the modern IDE’s are used alongside. Maven skipping several files can do it without any synchronization with the IDE. That will lead to IDE showing errors and warnings which may confuse the developer and result in spending time looking for a non-existent flaw in the code.

Another crucial feature Maven provides is dependency management. As it was said previously, manual installation and management of dependencies for a large project is meticulous work. Human factors can leave a lot of room for various bugs to arise. Therefore, such a feature is a must-have in any sufficiently large project.   
For developers, dependency management is introduced in the form of a single file *pom.xml*. POM here stands for Project Object Model. Briefly speaking, it is a configuration file, which contains various metadata and settings that are applied to the project. Firstly there is information about the project itself, it is presented in such way  
*<modelVersion>4.0.0</modelVersion>  
<groupId>org.socialapp</groupId>  
<artifactId>backend</artifactId>  
<version>1.0-SNAPSHOT</version>  
<name>Archetype - webdev-tools-research-backend</name>*The beginning is *<modelVersion>4.0.0</modelVersion>* tag, which indicates the version of the Project Object Model in use and right now, Maven only supports version 4.0.0, and this element is essential for validating the file's structure.   
The *<groupId>org.socialapp</groupId>* points to the project’s group or organization, following a reverse domain naming convention much like Java packages. This approach gives the project a unique namespace and helps avoid naming clashes in larger ecosystems.   
The *<artifactId>backend</artifactId>* specifies the exact name of the build artifact being created, which in this case refers to the backend component of the application. When combined with the group ID and version, this artifact ID allows Maven to generate a unique coordinate to identify this project. *The <version>1.0-SNAPSHOT</version>* signifies that the current build is a work-in-progress or development version that will eventually evolve into version 1.0. The SNAPSHOT label informs Maven that this build is subject to change and may be updated frequently, so it should always check for the latest dependencies.   
Lastly, the *<name>Archetype - webdev-tools-research-backend</name>* element offers a human-friendly name for the project. While it doesn’t affect the build process, it’s handy for documentation, user interfaces, and project descriptions. Altogether, these components create the core identity of a Maven project and guide Maven in managing its build lifecycle and dependencies.  
In similar form the whole document is built. There are several important elements, which are almost always present in each and every pom file.  
They are *<dependencies>, <build>* and *<properties>.*The *<build>* elementcontains useful instructions which are applied during the compiling process. In our project it contains plugins *org.springframework.boot* and *org.apache.maven.plugin*. The first one lets springboot to run, while the second one defines the version of maven compiler.   
The tag *<properties>* contain some additional information about the project*.* Inour example, there is only the version of Java that is being used.  
And the last, but not least, one is *<dependencies>*. It is the element which tells Maven what to include into the project build. Each *<dependency>* tag has important metadata about the package, such as name and version and looks like this  
*<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter</artifactId>  
<version>3.4.3</version>*It can be easily seen, that these three tags are the same as those in the head of the document. The explanation is simple – each and every package can be a dependency. During the build process, packages are uploaded into the local repository, and then they can be included into a different project. However, if the dependency is not present in the local repository, Maven will download it from the official cloud one. It behaves almost identically to the process of fetching data from GitHub or DockerHub. All of those are official websites working as a cloud repositories, where any user can store their package, and download packages of others.

To conclude, Apache Maven is a very efficient tool that is suitable for developing any Java-based application. Its vast functionality allows a developer to manage dependencies, create new packages, and include them seamlessly in their work. These packages can then be shared among other people via cloud-based repositories, promoting code reuse and collaboration. Maven also offers excellent speed during the build process. It efficiently utilizes multiple threads, and CPU cores and recompiles only the parts of the codebase that have changed, saving valuable time during development. Thus, Maven remains one of the most reliable build solutions in the Java ecosystem. Whether for small personal projects or large-scale enterprise systems, Maven significantly simplifies the software development lifecycle.

## Connection with the database

As it was said previously, the backend is a kind of a middleman between a user and the database. Frontend must not be allowed to access any sensitive information on its own, if there is a backend part. Such an action would create security risks, since backend usually protects itself by enforcing CSRF policies. Log in systems and diverse endpoints with different access levels. All of these serve the purpose of restricting access for unwanted users and software. Frontend, on the other hand, does not have so many restrictions, since its job is to accept any user and provide them with an interface of the application. So, the backend will protect the data, even if the Frontend is compromised.   
Naturally, JavaScript also provides means to handle databases, primarily through some libraries and frameworks designed for the same. These libraries, such as Sequelize, Mongoose, or Knex.js. They are primarily designed to be used on the server-side, especially in environments like Node.js, where JavaScript can be used as a full-fledged server-side language. In this sort of setup, JavaScript is not only required to get to the database but also is responsible for business logic implementation, API endpoint management, and user authentication. It is extremely essential, however, to recognize that JavaScript run on the frontend cannot ever connect to a database directly. Doing so would expose sensitive information such as database passwords and compromise the system with attacks of SQL injection or undue data tampering. Therefore, best practices necessitate that all database operations must be routed through secure backend services, frontend simply forwarding requests and obtaining responses, both being more secure as well as an appropriate application structure.

Now, we should discuss, what does Spring offer for simplifying the connection to the database.   
Spring Data JPA integrates JPA into the Spring ecosystem [38ї. JPA stands for Jakarta Persistence API. It is a standard for mapping Java objects to relational database ones, using ORM. ORM – Object Relational Mapping is an algorithm that allows converting objects in the memory, to objects in the database. It was developed specifically for OOP styled objects. That makes it a solid choice for Java, since it is an OOP-oriented language. Every entity in Java is a class.   
Behind the scenes, ORM translates dedicated Java classes into the objects in the database table. It also provides a set of commands, which are then translated into SQL queries. For instance,  
*int countAllByPost(PostEntity post);*This command is mapped to the CommentRepository. It will return the number of comment for the given post. ORM will translate it to the following query:  
*SELECT COUNT(\*)*

*FROM comment*

*WHERE post\_id = ?;*

To create an object for mapping, special annotations should be used.   
A screenshot of a computer

AI-generated content may be incorrect.  
Here it is an example of code, which defines a class for comments. @Entity says that it is a mapping object, and @Table tells which table in the database should it be mapped on. In this case, CommentEntity is mapped on “comments” table in the database.

A screen shot of a computer program

AI-generated content may be incorrect.  
And here is how the rest of the class is defined. Each of the annotations carries a valuable information about the field.   
@Id means that this field is a primary key of the entity.  
@GeneratedValue specifies that value of this field is created automatically, and the *strategy = GenerationType.IDENTITY* says that the database will handle the generation of this field. It is an equivalent to *id SERIAL PRIMARY KEY.*@ManyToOne indicates many to one relationship. In field *post* and *user* that means that each user can leave a lot of comments and each post can have a lot of comments as well.   
@JoinColumn sets the parameters of foreign key. Here it sets its name as well as the possibility of it to be null. It equals to  
*FOREIGN KEY (user\_id) REFERENCES user(id) NOT NULL*  
@Collumns, sets the properties of a single column. In the app, it is mostly used for forbidding null entries.   
@NotBlank forces JPA to validate whether it is empty or not. Also, it allows to set the error message in a case when the check is broken.   
The last @collumn annotation sets the value not to be included into INSERT and UPDATE queries. That means, that this value is set by the database itself and should not be changed.

The second aspect of data persistence configuration is the setup of a repository for an entity. A repository is an annotated interface with the @Repository annotation that provides pre-defined methods for database operations, such as the said int countAllByPost(PostEntity post);. These methods are automatically translated by Spring Data JPA to matching SQL queries. Repositories are reserved for data access and do not contain business logic.  
To handle that, we have a @Service class. This class acts as an interface between the repository and the external layers of the application, i.e., controllers or APIs. While the repository specifies data operations only, the service layer offers a functionality to the developer to invoke those operations, chain them together, and handle their output according to the needs of the application. It is in the service layer that business logic is implemented—such as eliminating results, checking rules applied, or calling other processes. Performing this separation of concerns prevents complicated, unmanageable code by clearly separating data access from business behavior.  
For further information, an official JPA reference on Spring website can be found [39]

Also, Spring JPA offers an additional layer of security to the application. While, it is not its main purpose, it still does it well. It is done through denying direct access to the database itself for the developer. Thus, any sensitive data will not be exposed, since the interaction happens only between Java objects. Another way of protecting the data is that by taking away the need to write queries manually, JPA eliminates the possibility of concatenation of queries. Tt may happen when a query is defined as a string which is concatenated with other string values to construct a desired request. For example,  
*String query = “SELECT \* FROM users WHERE name = ‘“ + userInput + “’”*Given scenario is a classical vulnerability to SQL injection attacks. While  
  
To conclude, JPA offers a reliable way for connecting to the database. It lets the developer focus on developing the application internal logic, instead of manually managing connections and queries. It is very flexible and offers a vast range of commands and configurations when it comes to setting up the entities of the database in the code. Also, using it will serve as a protection layer for the data stored in the database, by not showing any sensitive data in the code base.

## Accepting connections from the frontend

The other critical component of the MVC model handled by the backend is the controller. As discussed before, the controller serves as an interface between the model, which holds the application's data and business logic, and the view, which is the client side's user interface.   
In a RESTful backend, the controller's function is quite clear. It is the main point of entry for client-side requests, frequently called from the frontend, and directs them to related backend services. Each controller method is associated with an HTTP verb such as GET, POST, PUT, or DELETE and linked to a distinct endpoint. These methods should perform clearly defined operations, such as reading data, sending new data, modifying resources, or deleting records.  
Validation of incoming requests is one of the key tasks of the controller. It entails verifying the format and structure of the data, mandatory fields, and enforcing security controls such as authentication and authorization. Malicious or incorrect data is prohibited at this level to prevent it from being passed to further layers of the application.  
While the controller layer can appear to be a single block of code, its logic is actually shared across various parts of the backend. For instance, it usually delegates business activities involved to service classes, which hide the reusable business logic. These services tend to talk to the model layer, those are repository classes, to read or write data from the database.  
As a result of this delegation and interdependence, the controller layer is not isolated to a single file or module. Rather, it is an interconnected web of components that work together to orchestrate request routing, data validation, and service coordination. In this way, the controller layer assumes a fundamental, dynamic role of coordinating communication between frontend and backend, controlling the overall behavior of the system.

Now, let us discuss, how does Springboot helps a developer with accepting and responding to requests. As was said above, the essential work is done by the Spring-starter-web package. To create an endpoint, a developer must create a specifically annotated class. Then, methods will be created. Each method can be marked as an endpoint, which will result in its invocation if a request is trying to access that particular path.  
A screen shot of a computer program

AI-generated content may be incorrect.

Above is a screenshot of such a declaration. It creates a class called post-controller, while. Then, after a few dependencies, a method is declared. It is public, so anyone can access it. As the name suggests, it gets comments to a specific post and then adds them to the response array. In this case response Array is a Map, a hash table, but each hash is defined by the developer. The method returns *ResponseEntity. ResponseEntity* is an object provided by the Spring framework. It simulates an https response, in a similar manner how DTO object simulates a database table. It allows the developer to set custom headers, set different https statuses or return errors in a consistent format. In a similar manner all the endpoints are created. They are basically methods grouped into classes by their purpose. Annotations, like @*RestController* define how Spring will see and treat them. They are basically a way of communicating with the framework and telling it what a certain class will do, and when to address it.

Below are descriptions of the most important ones, which are required for a RESTful backend to work  
**@RestController** – Marks the class as a controller for an individual or a collection of REST endpoints. It tells Spring to automatically convert the return types of the methods (normally Java objects) to a response type like JSON or XML, and send them to the client.  
This annotation combines @Controller and @ResponseBody to ease the creation of REST APIs.  
**@RequestMapping(“/name-of-the-mapping)** –   
Declares the base URL path which this method or controller will serve.  
When a client sends a request to the given path, Spring will pass it to the method (or class) that is annotated with this. Can be used at class level (to identify a default route) and method level (to handle certain requests inside that path).  
**@GetMapping** - Abbreviated form of *@RequestMapping(method = RequestMethod.GET)*.  
Maps HTTP GET requests to a specific method. It is used to read or fetch data.  
Usually applied to load user profiles, lists of products, etc. Similarly, there are shortcut mappings for all request types, *@PostMapping, @DeleteMapping.***@PathVariable** - Binds a URL path parameter to a method argument.  
Allows the method to read portions of the URL, like an ID. It is used for non-sensitive data, which will not be dangerous if compromised.  
**@RequestParam** - Binds a query parameter to a method argument.  
Usually applied to optional filtering, sorting, or paginating.  
**@RequestBody** - Takes the body of the HTTP request and converts it into a Java object. Useful when the client sends structured data like a form or an object.  
**@ResponseBody** - Instructs that the method return value should be written directly to the HTTP response body and not processed as a view name.  
Comes with *@RestController* by default but can be used on its own with *@Controller*.

To conclude, Spring framework offers an excellent solution for connecting backend and frontend together. There are possibilities for a declarative creation of endpoints. They are easily mapped to certain paths in the domain. Numerous annotations make the code easier to write and read and thus making the whole codebase more maintainable.

## Authentication and Authorization

Another critical component of creating a secure web application is secure client-server communication. When the web client communicates with the backend, it is critical that the system be protected from illegal access and malicious requests.  
By default, most of the endpoints will have to be restricted, and only specific users or roles can be allowed to access them. In a secure system, sources that are unknown must be refused requests, and only trusted sources must be allowed to perform backend interactions.  
Web applications are generally hosted from the same origin frontend and backend under the same domain in normal cases. The browser requests are given by default in such cases. But nowadays, frontend and backend generally run on different domains or ports In such cases, Cross-Origin Resource Sharing (CORS) policies must be configured correctly to give explicit permission for communication between two services.  
In a safe setup, there must be a single trusted domain (the frontend's domain) allowed to request the backend. It is done so that no evil domains can request the backend and potentially exploit the API.  
CRSF – Cross Site Request Forgery is a type of attack when a malicious request is sent to the backend by a different website, on behalf of our frontend. Web applications with session-based authentication are especially vulnerable to such an attack, since backend stores the information about a specific host. When a request is received, it will assume that it is that host who sent it.   
CORS – Cross Origin Resource Sharing is a policy, that restricts any requests between different hosts. Even if those are different applications on the same machine. So, only the same website can send any requests to itself. In our case, an exception is made, since the frontend is in a separate container, that means that it is treated as a completely different entity.   
In addition, CORS protection can be configured through Spring Security or Spring Web configuration, and you can then define which domains are permitted to send requests, what methods are supported, and what headers can be used. With correct configuration, CORS enables secure communication between frontend and backend even when they are on different domains.

The second layer of request security is the decision whether a client is authorized to access a specific resource. And especially if they are allowed to change or delete data of any kind. For that reason, since the dawn of internet, developers have been implementing authorization mechanisms. The most common approach was session-based authentication.   
It works by backend storing information about all of the current sessions. When a user logs into their account, a record is created. This record holds data about who is logged in and how they can be recognized.   
Now, the modern way of authentication is via JWT tokens. As was described above, JWT token is a string that carries information about the user and is signed by the backend. This signature is what makes them safe, since it can only be created using a private key that is stored in the backend. The signature is a string composed of header and payload and hashed with a hashing function. The strength of the algorithm determines how reliable the signature will be. However, in most cases it is reliable enough. Since almost for each string there can be a unique hash.  
When a user is authenticated, the backend will generate and send the token back to the client. The token will contain user-related information (claims) and will be cryptographically signed. When the client subsequently sends out a request that will be subject to authentication, the token will be sent back to the backend typically in an HTTP header or cookie. To confirm the identity of the user, the backend does two principal verifications:  
**Expiration** - The token's exp (expiration) claim is verified. If the current time exceeds the expiration time, the token is invalid and is rejected immediately.  
**Signature Verification** - If the token is still valid according to time, the backend generates a new signature by combining the header and payload, and then employing the same algorithm and secret key used while creating the token. The generated signature is compared with the one in the token.  
If the token data was tampered with. For example, if someone was trying to modify the payload or create a fake token, the signature would no longer match. Because the attacker wouldn't have the secret key, they wouldn't be able to generate a valid signature. The backend would then detect the discrepancy and reject the token.  
Now, each user is assigned with a role “USER”. Roles help separate contents with different access levels. In the sample application only one role will be presented. All other endpoints will be public. A screenshot of a computer program

AI-generated content may be incorrect.

Here is a part of backend code that defines which endpoints are accessible for public use are which are not. Most restricted paths are the ones that manipulate the data. They create new posts and comments, as well as let users follow each other. On the other hand, public resources provide clients with data. They allow anyone to see posts and comments as well as log in and create new accounts.

To conclude, Spring security adds a layer of protection against most common attacks and integrates easily with widespread authentication mechanisms. Still, it is highly customizable and can be changed drastically in accordance with the application’s needs. It also allows developers to divide users into roles and restrict access to specific endpoints for specific roles.  
And in general, Spring Framework offers a vast number of solutions suitable for a web application. It allows developers to write code in a more declarative manner, just specifying what should be done, not how. It is built on top of Java language, which has extensive and robust functionality. Java has been developed and supported for years, as well as used for backend development purposes. It combines neatly with maven build tool, which simplifies dependency and package management.

# Frontend

## Language and Library

In MVC design, the frontend would typically be working with the View component. It is the main interface through which users interact with the application, and it handles the presentation layer. On the web, it typically shows up in the form of a website.  
The frontend displays information sent by the backend and allows users to perform actions such as filling out a form, navigating pages, or handling graphical objects such as sliders and buttons. These actions trigger events that are forwarded to the backend, which gets the input, communicates with the data layer, and sends responses or new data.  
An effectively built frontend improves user experience along with ensuring that information is being communicated in a clear, accessible, and responsive manner on any platform. It becomes critical in determining the usability and perceived performance of the application.

Nowadays, there are two main ways of building a frontend. They are both popular and widespread. One is focused on the server creating a webpage which is then sent to the client. Everything is being made in the cloud. Client only gets a result. Such pages are usually static. When a significant update is needed, the backend processes such request and returns a new, refined page. Such significant change can occur when a user tries to access a document that is in another location in the domain. For example, go from *https://social-media.com/home* to [*https://social-media.com/profile*](https://social-media.com/profile). In this situation a whole new page will be rendered and sent to the client. It may take some time, especially if the client’s internet connection is slow. In this case, the role of JavaScript is to make changes on the page itself. It can create animations, drop-down lists or carousel views of the images. Also, it handles the whole communication process with the backend. Such architecture is called *backend-render*.  
Another way, which is more modern, is to render pages on the client’s local machine. Such pages rely on the backend only for sending them fresh data. Of course, the machine of a user should be more high-performance. However, modern day personal computers and even mobile devices are usually capable of doing such tasks.   
The gradual development of JavaScript throughout the years made the language suitable for rendering the pages itself. It can manipulate the DOM structure. DOM – Document Object Model is a virtual structure of the document. It is a composition of all the elements present there. Tags like *<body>*, *<div>*, or *<image>* are all included inside of it. It is a tree with each node being a tag. Almost each of them can have children and siblings.  
Moreover, JavaScript allows a developer to implement routing. It is possible due to the feature called event-listening. Language has a database of events that can occur when a user interacts with the page. They include clicks, selects, submits, selections etc. So, for example, when a button is clicked, the language engine can detect it. Then code tied to that will be executed. Similarly, clicking on the link that leads to another sub-page of the website is an event too. In addition to that, JS can access and manipulate the URL of the page. Thanks to all of that, it can fully intercept a request and render the page by itself, while only fetching data from the backend. Such applications are called SPA, which stands from Single Page Application. The name suggests that there is almost no reloading of the page. It is the type of the frontend that will be shown in the sample application. It fits perfectly into the combination with RESTful backend, since both of these are made partially independent of each other.

This separation of concerns not only promotes cleaner architecture, but JavaScript is also a mature and robust programming language, utilized heavily for the control of the frontend of web applications. As mentioned before, it is fully capable of handling rendering, user interaction, and communication with backend services. JavaScript is a dynamically typed, interpreted scripting language, i.e., it requires no explicit declarations of type and does not have to be compiled prior to running time. Such aspects, combined with their flexible syntax, make it relatively easy to learn for new programmers as well as experienced developers.

Although JavaScript is not formally object-oriented, it does provide support for a number of programming paradigms, including object-oriented, functional, and event-driven style programming. It is much like Python in most ways, in simplicity and flexibility, although the two languages are very dissimilar in both syntax and environment.

JavaScript has many unique aspects that distinguish it from other languages, and these are what contribute to the uniqueness of the world of frontend development. These range from its asynchronous behavior in how it uses callbacks, promises, and *async*/*await*, to its event loop model, and its tight coupling with the Document Object Model.  
**Callback** is a function that is passed as an argument into another function to be executed after a certain task. Callbacks are frequently utilized in asynchronous operations, such as waiting for a user event or data from a server. While useful, excessive use of callbacks, and especially nested callbacks, leads to what is known as "callback hell," making code not as readable or maintainable. Callbacks are actually known in other languages. They are often called ”lambda” or “anonymous” functions. Those are functions that are defined where they are called. They do not have a name and are not stored in the memory after their execution. Of course, if nothing else is not referencing them. JavaScript is known for its extensive usage of those functions.  
**Promises** offer a neater approach to asynchronous operations. A promise is a value that might be available now, in the future, or never. It can be in one of three states: pending, fulfilled, and rejected. Promises enable chaining operations with *.then()* and error handling using *.catch()*, which makes asynchronous code easier to read compared to highly nested callbacks. While JavaScript being a single-threaded language, promises are an attempt to mimic multi-threading and parallel execution.   
JavaScript also functions in tight collaboration with the DOM. Through its inherent methods like *document.querySelector()* or *getElementById()*, JavaScript is able to access, modify, add, or remove HTML elements on the fly. This allows developers to refresh content dynamically, respond to user events, or animate UI elements without reloading the page.

It's interesting to note that JavaScript was originally designed to run within the browser, adding dynamic capabilities onto web pages. Only in more recent times, due to the arrival of environments like Node.js, did JavaScript find itself able to run outside the browser as well and thus became entirely well-suited to server-side coding as well. That's a step forward that turned JavaScript into a full-stack development tool, able to push anything from UI rendering to backend logic.

React, as noted above, is a JavaScript library for building user interfaces. It is often mistakenly referred to as a framework due to its immense popularity and extensive functionalities, but in reality, React only deals with the view layer of an application. It provides a set of functions and utilities for producing and rendering UI components in a seamless way. One of its prominent features is the virtual DOM, which enables React to keep track of changes and update only the sections of the page that require re-rendering, thus enhancing performance and responsiveness.  
The arrival of React made building Single Page Applications a whole lot easier with its capability to enable component-based architecture and declarative UI programming. React's ecosystem has evolved a lot over the years, and today there is a wealth of supporting libraries and tools such as React Router for routing, Redux or Recoil for state management, and frameworks such as Next.js for server-side rendering and full-stack development.  
As a result, while React is still technically a library, its ecosystem enables developers to build much more than just responsive UIs by themselves. It can now support anything from dynamic frontends to full integrated applications.

Among other things, React introduces a new semi-syntax. It is called JSX. JSX allows developers to write html tags directly blended in the JavaScript code. It fulfills the main purpose of React, which is to help developers in creating UI components.  
Component is a piece of a page. Typically, it consists of several html tags. For example, it can be a button and its wrapper, or it can be the whole page. Below is an example of a component. It describes the log in page.  
A screen shot of a computer

AI-generated content may be incorrect.  
Here a declaration of a function can be seen. It is a function that returnts a JSX syntax. Basically, the role of such piece of code is to describe what component will be rendered into the page. As we can see here, most of the tags are usual HTML elements. However, there are some pieces which should be explained furhter.  
Firstly, ***className*** property of two elements. React introduces its own names for properties, since pure HTML can conflict with JavaScript syntax. In this situation, class property will come in direct conflict with the JS keyword *class.* Also React writes all of the properites in camel case. For instance *tabindex -> tabIndex*.  
Secondly, there is an elements called *<LoginForm />* . There are crealy no such tags in pure HTML. This is an instance of pre-made element, being called inside of another one. Inside the project there is another function declared, it is called *LoginForm*. Similarly to the presented one, it returns HTML elements. So, what is happening here is simly a call of that function. In the same manner, *LogIn* is called somewhere else in the project codebase. What React does, is takes all of such functions on a page and renders their returned value. Thus, in the final document there will be only pure HTML tags. Instead of *<LoginForm />,* there will be its returned contets*.*

To conclude, JavaScript has also traveled a long distance in its development. Today, with all its rich functionality, it is able to manage entire pages, generate content dynamically, and even route. That means, it has turned into a full-fledged self-sufficient language specialized in the development of interactive web applications. Along with JavaScript, React is also a robust tool that provides developers with a simple and efficient way of developing reusable UI components. These elements are then translated into HTML elements and painted onto the page, which makes developing contemporary web interfaces simpler.

## Packet Manager and Build Tool

As mentioned earlier, frontend applications increasingly can coexist with minimal reliance on the backend. This freedom, however, also has a tendency to be highly architecture intensive. Today's frontend applications are no longer just UI components and styling, they also possess an abundance of utility modules to handle essential features such as state management, routing, handling forms, responsive layout, etc. Over time, these utilities end up being a large, complicated codebase that captures the kind of complexity traditionally seen with backend systems.  
In addition to modules consumed internally, frontend applications are also dependent upon numerous external frameworks and libraries. Versioning, integrating, and maintaining these dependencies is an additional level of complexity that contributes to the challenge. Hence, it becomes more challenging to maintain a frontend application at scale.  
Just as the backend employs frameworks and tools to structure code, manage dependencies, and enhance development, so too must the frontend employ similar tooling. Webpack, Vite, or Next.js and Angular CLI are some of the tools that are utilized for this sort of thing. They provide standard project structures, automate builds, manage dependencies, and keep maintainability when the app grows. In essence, good tooling is required to aid frontend developers in coping with and controlling the increasingly high complexity of modern web applications.

In the sample project, Vite is selected as the build tool because of its speed, simplicity, and newer architecture [40]. Those features make it extra suitable for frontend development in the modern era. Unlike legacy bundlers that precompile the entire app prior to serving, Vite uses a different strategy. When in development mode, it utilizes the full power of native ES modules in the browser to serve source files on demand. This results in much faster launch times and almost instant hot module replacement (HMR), so that developers can see changes in the browser almost real-time without reloading altogether.  
Vite is able to handle a great variety of file types. It can automatically process JavaScript, TypeScript, JSX/TSX for React components, HTML, CSS, and style preprocessors such as SCSS, which is used in this project. It can also import static assets such as images, fonts, and SVGs directly into JavaScript modules. Whenever a file changes, Vite intelligently recompiles only the involved section of the codebase and refreshes it in the browser, providing a seamless and efficient development experience. So basically, a code is almost instantly translated into changes in the browser. Needed to say though, that this dev server cannot be used for production environment. For such purposes, Vite offers a possibility of production build. After that, the whole project is translated to an index.html file with complex JS-code and CSS built around it. Then the website should be served using a server engine, so an end user can access it. In the sample project, to simulate production environment, Nginx engine will be used as a server for frontend.  
Under the hood, Vite uses the esbuild bundler and transpiler, implemented in Go, for pre-bundling of dependencies. This reduces the startup time for the development server greatly and optimizes module resolution performance. When building for production, Vite switches to Rollup, an older but extremely popular bundler that outputs highly optimized and tree-shaken code. This combination ensures that development and production builds are as fast and efficient as possible.  
In addition, Vite features a powerful plugin system that allows developers to introduce custom behavior into its operations, such as auto-importing components, handling GraphQL files, or even linking up with test libraries. It also comes with sane defaults to start with but is completely configurable through its configuration file. These are vite.config.js or vite.config.ts. Such aspects make Vite more than a build tool, but a development environment tailored for scalable, maintainable, and high-performance frontend application creation.

In modern frontend projects, the number of dependencies, foreign libraries, frameworks, and utilities, tends to grow quickly with more sophisticated projects. They are needed to introduce features such as routing, form validation, UI components, or state management. However, they need to be downloaded, managed, and maintained, which can become a tedious and bug-prone process if done manually. This is similar to how packages outside of the package manager are handled in languages like Java but, in the JavaScript realm, this process is greatly simplified because there is a built-in package manager.  
JavaScript was first built to run inside browsers, allowing web page manipulation and user interaction response for programmers. However, with the implementation of Node.js, a runtime environment based on Google's V8 JavaScript engine, JavaScript was a general-purpose language that could run on servers or desktop applications.

One of the pillars of the Node.js ecosystem is NPM. NPM stands for Node Package Manager. It is a powerful tool that can automatically install, upgrade, version, and administer third-party packages. When a developer initiates a project, NPM typically makes a *package.json* file, which is a manifest that includes metadata about the project along with its dependencies. Through running commands like *npm install*, NPM reads this file and installs the packages listed here automatically from the NPM registry, a huge virtual repository of over a million public JavaScript packages. The structure of that file is pretty similar to *pom.xml.* However, *package.json* has another extension. In all of the other things they are alike. *Package.json* starts with a declaration and metadata about the project, which is followed by dependencies for production and development. The difference between those two is that devDependencies are not compiled into the production environment. They are typically some plugins which make the development process easier. For example, EsLint. It is a plugin that can be customized by a config file, then it acts as a syntax checker or even auto-correction tool. It checks if all of the indentations consist of the same number of whitespaces or if numerous import statements in the file are structured correctly. Those features are clearly not needed in production, but they can greatly simplify teamwork between several developers. They will be forced to write code in a one, specific, style.

Below is an example of a *package.json* file, that is used in the sample project.  
*"dependencies": {  
 "@hookform/resolvers": "^5.1.1",  
 "lucide-react": "^0.511.0",  
 "prop-types": "^15.8.1",  
 "react": "^18.3.1",  
 "react-dom": "^18.3.1",  
 "react-hook-form": "^7.58.1",  
 "react-router": "^7.6.1",  
 "react-router-dom": "^7.6.1",  
 "sass": "^1.89.0",  
 "zod": "^3.25.67",  
 "zustand": "^5.0.6"  
}*  
It is a list of dependencies of the project. It is a Json object. Each entry is composed of a name and a version of a dependency. These packages are installed to a local *node\_modules* folder within the project root, and then the application can be used with simple import or require statements. NPM ensures versioning is correct through semantic versioning so that there are no compatibility issues when updating dependencies. Versions of dependencies can also be locked down by developers through package-lock.json to achieve identical installations across different environments.

NPM is not only used for package installation. Custom scripts, such as build processes, test suites, or development servers, can be run with NPM, all specified in *package.json*. For example, npm run build might call a tool like Vite or Webpack to bundle the code for production. This scripting capability turns NPM into a task runner, automating essential parts of the development process. Such scripts are also described in the *package.json*.  
*"scripts": {  
 "dev": "vite",  
 "build": "vite build",  
 "lint": "eslint .",  
 "preview": "vite preview"  
}*  
In a form of a json object, npm aliases are provided. They are followed by Vite lifecycle commands that will be invoked upon their usage. For example, *npm run dev* will invoke the command *vite*. In result a dev server will be started up.

To conclude, in the modern world JS is equipped with a robust package manager. It offers productive dependency management, automates boring chores, allows for version control, and connects developers with a gigantic universe of reusable code, making it a necessity for frontend and backend development using JavaScript. It works great with Vite, a quick and reliable building tool. Vite provides developers with a possibility of development without reloading and offers a quick packaging process both for dev and production environments.

## Creating styles

The frontend of a web application is what directly interacts with users—essentially, the user interface (UI). For it to play its role effectively, it not only has to be functional but also appealing to the eye and comfortable to use, especially in business contexts where design directly affects user engagement and conversions.  
In the early days of the web, most websites were merely collections of static HTML pages. They featured basic functionality such as plain text, links, and images. While functional, they were not exactly designed in any sense for interactivity or impressing visitors. The primary concept at that time was to present information, but not necessarily to impress individuals or get them involved to any great extent.  
Yet as the web began gaining popularity, it began attracting the attention of companies that wanted to go online. Companies saw the web as a new medium to market goods, build brands, and reach audiences globally. At the same time, individuals began creating personal websites, blogs, portfolios, hobby sites, many of which also wished to stand out with distinctive designs  
This heightened demand for more visually engaging and interactive sites heralded the introduction of CSS. CSS, Cascading Style Sheets, was a major shift in web development in that it allowed developers to separate content from presentation. With CSS, web pages could now be styled with colors, fonts, spacing, and layouts, causing them to appear more like polished, professional-looking designs and less like plain documents. It allowed designers to take control of the visual identity of a website, in much the same way that graphic designers work with print media.  
As CSS evolved, it introduced support for animations, transitions, and responsive design, enabling dynamic visual effects and layouts that could fluidly adjust to different screen sizes. Developers could now make pages move, respond, and flow, by nudging web interfaces to life and improving the user experience (UX). The frontend, once static and limited, became a creative playground for crafting immersive, interactive, and engaging web applications.  
Today, frontend development continues to evolve with styling tools like CSS Grid and Flexbox and preprocessors like SASS that enable even greater design flexibility. Paired with JavaScript and newer frameworks, the frontend has become a core and sophisticated layer of web development that is not only responsible for aesthetics, but also how users perceive and interact with a digital product.

Styles in web development define the visualappearance of HTML elements. They control everything from basic properties like width**,** height**,** color**,** and **font** size, to more advanced features such as transitions**,** animations**,** shadows**,** andgradients. Basically, styles determine how an element looks and behaves visually on the page.  
There are multiple ways to apply styles, the most direct being inline styles. They are written directly within an HTML tag using the style attribute. For example:  
<p style="color: red; font-size: 16px;">   
Inline styles have the highest specificity. That means that they override other styles defined elsewhere. However, they are generally discouraged in large projects because they clutter the HTML and make it harder to maintain or reuse styles.

A more scalable and preferred approach is to write styles in separate CSS files or in a <style> block within the HTML document’s <head>. In this case, you need to indicate which elementsthe styles should apply to, and that’s where selectors come in. Selectors can target elements in several ways.  
**Tag name**, for example p or div, to apply styles to all elements of that type  
**Class** to target elements with a specific class attribute  
**ID** to target a single, unique element  
**Combinators and pseudo-selectors** to apply more advanced targeting logic. For example, div .title will target only <div> elements with .title class.

When a style is defined for a selector, it gets applied to all elements matching that selector. Among these, using class-based selectors is the most common and recommended practice. Class names are often semantic, meaning they reflect the purpose or role of the, and they allow multiple elements to share the same styling rules.  
The term “cascading” in Cascading Style Sheetsrefers to how styles inherit and override one another. Styles defined on a parent element often apply to its child elements, unless a more specific rule overrides them. This inheritance allows for efficient styling, as you can define broad styles at higher levels of the DOM and have them cascade downward, creating consistent design patterns across an entire page or application.

CSS is a styling force for web pages with a vast number of properties to deal with layout, appearance, and visual appearance. But it had one essential weakness: it was designed for tiny static HTML documents, not the big-scale, component-based web applications we build today.  
With frontend applications getting increasingly complicated, developers are left with the common problem of writing CSS. For instance, styles get repeated across components, especially when needing a uniform color scheme, spacing system, or typography. Without variables, it is boring and error-ridden to deal with frequent values that have to get shared. Also, very complicated components, composed of highly nested HTML structures, are at risk of having long, repetitive selectors in the CSS, which make the stylesheet cumbersome to read and maintain.  
Fortunately, there are new solutions to overcome these deficiencies nowadays. CSS preprocessors are sophisticated tools which allow developers to write styles in more expressive and powerful forms. Those powerful styles get converted, or transpiled, into plain CSS that can be understood by the browser. Preprocessors help to introduce programming-like ideas to CSS, so that stylesheets are more modular, scalable, and maintainable.

Here in this demonstration application, we shall use SCSS, Syntactically Awesome Stylesheets. Is it one of the most popular CSS preprocessors at the moment of writing [41]. which is most likely the most popular and feature-rich CSS preprocessor. SCSS introduces to CSS a variety of useful features, including:  
**Variables** – define and reuse frequently recurring values like colors, font sizes, and spacing units. They work by declaring a variable *$var:* *#000*, which then can be used in the same file.

**Nesting** – makes selectors look like HTML structure so that they are easier to read  
Instead of something like this  
*div p {  
 color: red;  
}  
div button {  
 width: 20px;  
}*The code will look like this  
*div {  
 p{  
 color: red;  
 }  
 button {  
 width: 20px;  
 }  
}*

**Mixins** – reusable style sets that can be imported wherever needed. They act similar to functions, or methods, in regular programming languages. Below is an example  
*@mixin page\_container {  
 background-color: var(--background);  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 height: 100vh;  
 padding: 1rem;  
}*It is a block of an html code. When used in another element, it will paste all of those styles into that element.

**Functions** – perform mathematical operations and return dynamic style values. For example,  
*width: calc(100% / 3);*That will set the width to the result of the maximum width divided by three.

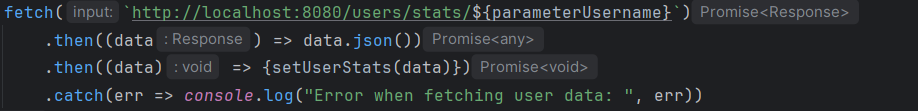
**File Importing** – split large stylesheets into small, manageable modules. For example,  
*@use '/src/styles/auth\_mixins';*  
That will import the contents of a file *auth\_mixins* into a current file. So the mixins can be used there too.

**Conditionals and Loops** – enabling logic and iteration in styles for creating patterns programmatically. For example  
*$theme: 'light';*  
*.button {  
 padding: 10px 20px;  
 border: none;  
 border-radius: 4px;  
 @if $theme == 'light' {  
 background-color: #ffffff;  
 color: #333333;  
 }* *@else {  
 background-color: gray;  
 color: black;  
 }  
}*That will set the colors of a button taking into account if a light theme is enabled.

To conclude, CSS offers a wide and robust functionality for styling web pages. It can make it more personalized and dynamic and thus fit for a specific user. With the use of SASS, we have a more organized and scalable way of styling, particularly with bigger UI systems and component libraries. It avoids inconsistency in the application and accelerates development and cuts down on duplication by making CSS feel like an average programming language with imports, functions and variables.

## Getting data from backend

As has been said, the frontend of a web app is largely disconnected from the backend. It can render views, process user input, and even page-to-page navigation independently. Nevertheless, it still relies on the backend to supply it with critical data, specifically when such data needs to be secure, dynamic, or persistent.  
There are several reasons why this separation of concerns is beneficial. Security is one of the strongest. The backend will typically run on a secure server, guarded by firewalls, authentication layers, and encrypted communication protocols. The frontend, however, is merely an aggregation of static files, HTML, CSS, and JavaScript, that are downloaded into the user's browser. What this means is that any data or logic within the frontend is completely exposed to the user and can be seen or modified. Due to this, sensitive operations, user authentication, business logic, and database access should all remain on the backend.   
The frontend can be thought of as a visual interface, but not the whole application. It shows data, handles user input, and communicates with the backend to get things done. This requires a way for the frontend to fetch data from the backend. It is typically done through APIs. In our case, such API is our backend.  
In traditional server-rendered apps, everything is simple: the server constructs the entire HTML page with the needed data already preloaded and serves it to the user in a single response. But for Single Page Applications, this is not the case. SPAs are created as completely standalone frontend applications that run inside the browser and load data asynchronously.  
To retrieve data, SPAs use HTTP requests to talk to the backend using APIs. Usually, they are RESTful, as in our case, or built by GraphQL architecture. The requests are sent to specific endpoints defined by the backend, and responses are usually given back in JSON format. This way, the frontend can dynamically load content, reload views, and respond to user input without the need to reload the entire page.  
We will copy this pattern into our sample app: the frontend will make API calls to communicate with the backend, sending requests to particular backend endpoints and displaying the data that is returned appropriately. This kind of architecture has the tasks tidily segregated, is more secure, and allows for flexible and scalable development.

JavaScript offers a vast functionality for fetching data. Main part of it is Promises. Promises are asynchronous, meaning that the execution of promises does not block the main application flow. When a promise is created, it can be treated like a contract that something will happen in the future. For example, a promise is created to fetch data, then when data is received that promise will resolve itself. That means that it will return its data for further processing. This data cannot be directly assigned to any of the variables, since technically it is not known when a promise will be resolved. Though, data can be processed using special promise methods. Those are *then()* and *catch()*. The latter one is used for handling errors that may arise during promise execution. While if everything goes smoothly, *then()* is used. It received data that was returned by a promise and can process it in any way. Usually, it even returns a new promise that is then resolved with its own *then()* method called. Below is an example,  
  
Here a *fetch()* function can be seen. It sends a get request to the endpoint. In this case, it attempts to get stats of a user that is written in *parameterUsername* variable. It returns a promise. Then it is handled by *then()* method. Here a work of JS anonymous functions can be seen. They are also called ”arrow functions”. In the case of them being passed as parameters, they are usually referred to as “callbacks”. A lot of functions in JS accept other functions which define how they would act. In this case a function is passed to *then()* method, it basically defines how the data will be processed. The first callback transfers received data into *json* format. That function returns another promise which is then handled by the next callback. The second callback takes the data returned by the previous one and passes it to a function *serUserStats()*. The purpose of this function is to send a signal to React to update the DOM and re-render the page. In a case if pure JS is used, a developer can update the DOM directly.  
  
Promises actually are only mimicking parallel work. As JavaScript is a mono-threaded environment, it is unable to execute more than one piece of code in real parallel mode like some other languages that support multi-threading. This means that it is forced to execute one operation at a time on its master execution thread. However, JavaScript handles tasks such as user interaction, timers, and asynchronous operations effectively with the assistance of a process called Event Loop.  
The Event Loop is the backbone of JavaScript's non-blocking, asynchronous behavior. Instead of blocking all resources waiting for something to finish slowly JavaScript hands such work off to the browser or Node.js runtime environment, and continues executing other code. When the task is done, a callback or promise is put into the task queue, and the Event Loop will pick it up at some time when the main thread is free.  
This design allows JavaScript to remain responsive and quick even while it's waiting for slow operations. It's especially important while building the frontend, where blocking the main thread would freeze the user interface. An understanding of the Event Loop is central to writing beautiful, non-blocking code in JavaScript, especially while using asynchronous functions such as setTimeout, fetch, async/await, and event listeners.

Fetching function *fetch()* offers a robust interface for creating requests. It is not semantically created, since it cannot not only fetch but send data too. It supports all of the standard request types like GET, POST, PUT, DELETE and allows a developer to customize headers and body of a request. Data can be sent in various formats like JSON or FormData. As well as response data can be received in different formats. It can also be JSON, text or binary.   
Here is how a POST request can be sent A screen shot of a computer

AI-generated content may be incorrect.  
Here fetch will make a POST request in order to create, or cancel, a subscription for between two users. Firstly, JS will check if *isFollowing* is true or false, and depending on that it will send an appropriate request. This endpoint is considered to be private in the backend. Only a logged in user can access it. To ensure a proper workflow, two headers are attached to the request. One tells the server that the data is formatted as JSON, the second one attaches the access JWT token. Now the server can check this token and if it is valid, fulfil this request. In similar manner all of the communication between frontend and backend is implemented. Frontend sends requests and receives new, updated data from the backend.

JavaScript provides a dynamic and fluid front-to-backend communication platform in the guise of its intrinsic support for asynchronous operations and network requests. Through the fetch() function, programmers can send and fetch data from and to backend APIs without blocking the user interface. Through promises, JavaScript provides smooth, non-blocking processing of data so that frontend applications are responsive while waiting for the server response. This robust alliance forms the core of modern dynamic web applications, where frontend and backend building blocks effectively work together to deliver data-driven real-time experiences.

## Routing Between Pages

Traditionally, sites are made up of many discrete pages, and each page corresponds to some specific route or endpoint on the server. If a user is requesting one of these endpoints, the server responds by returning a complete HTML document for the page. This is the classic way of serving multi-page sites, where many interactive stuffs, such as buttons, are really hyperlinks that take the user off to a different route on the same site. Single Page Applications, however, function differently. Instead of reloading the entire page with each navigation, SPAs dynamically update the content on a single HTML document. This approach allows the frontend to handle its own routing logic without relying too much on the backend and only interacting with it for retrieving the data or doing other required operations.

The strength of JavaScript predisposes it to be applicable in an effort to accomplish client-side routing. In simple terms, every page switch or redirect that occurs in a web application can be referred to as an event. This means that the JavaScript interpreter is capable of intercepting these navigation events and handling them according to specified logic rather than allowing the browser to reload the whole page. In addition, JavaScript is able to access the URL of the browser directly through the *window.location* object and the History API in order to read, change, and update the address bar without making a request to the server. Together, these two features all the tools needed for JavaScript to handle routing on the client side independently, lowering the backend dependency for navigational purposes.

However, managing routing manually, through pure Javascript can also create some security vulnerabilities. For example, routes which are usually forbidden for anyone to access can be exposed in frontend code. It is a general security rule to treat frontend as the most vulnerable part of the application. It is done due to the fact that the whole page alongside JS code and CSS is shipped to a user’s browser and can be easily accessed. Therefore, a developer must be very careful when exposing any data on the frontend. And in addition to that, backend still must implement security precautions, even if the same application’s frontend is the only one to access data. In security question no trust should be given without double-checking. And also, writing routing in pure JS can be a time-consuming task.   
However, in conjunction with React, there is another library that can handle routing pretty well. Its name is React router dom. The routing structure of our frontend will include five pages. Home, Profile, Login, Signup and Post. Each of them will be on a separate path. Also, paths to those pages are different from API endpoints. For example, for one page there can be two independent requests to different endpoints to fetch different resources. It is done not only for security reasons, such structure can also reduce the number of api endpoints, since they can be used by different pages. Below is a image that depicts routing structure in our frontend application.A screenshot of a computer code

AI-generated content may be incorrect.  
Once the package is installed, routing logic is defined inside an instance of *<Routes />*. As noted earlier, in React a component is essentially a function, *<Routes />* is essentially a function call that defines the routing configuration.  
Inside *<Routes />*, you define one or more *<Route />* components. Each *<Route />* takes two main props:  
**Path** — a string of the URL part to match. It may be a static path like "*/home*" or a dynamic one like "*/users/:id*", where :id is a route parameter.  
**Element** — the React element to render when the current URL matches the path. It may be any valid element, from a basic page to a complete layout.  
When the user navigates onward, either by clicking a link or programmatically through calls to functions like *navigate()*, the router is listening for modifications to the URL of the browser. Instead of inducing a full-page refresh, it catches the navigation event, modifies the browser's history, and renders the relevant component in the existing DOM tree. This is why Single Page Applications seem so smooth and fast: the browser does not have to re-download the whole HTML from the server, only the content that must be re-rendered.

To conclude, routing tasks can be done with pure JS. As with everything in programming, a developer can implement each and every task by hand. Libraries are just solutions already made by someone else. In case with routing, react-router-dom library is very useful, since implementing routing by hand can expose vulnerabilities of the application. It can be considered a sensitive matter; therefore it is recommended to do it with a secure and reliable library. Also, as with everything else, React here reduces the boilerplate code and makes the whole coding more declarative.

## Managing global variables

There are any number of variables in a given application with data to render and interact with. Manipulating these variables on the frontend is generally not a very heavy task in typical server-side rendering scenarios. Since each page is constructed on the server and sent as a complete HTML document, most important data — especially global or sensitive data — is on the server. A tiny percentage of variables, typically user behavior on the current page, reside on the client side. This provides SSR with a degree of security due to the fact that the client has less control over significant information.  
But in a Single Page Application, things are different. With the frontend taking the responsibility of rendering and updating the UI dynamically, it must deal with a larger portion of the application's data directly within the browser. This data management needs to be secure and efficient. It should not , not to attract unauthorized access or interference and make the application responsive and easy to maintain  
Libraries and frameworks such as React bring an additional layer to variable handling by the introduction of state. In React, state variables are bound to a particular component, and changes to state variables are monitored by the rendering engine. When a state variable is modified, React re-renders the impacted component, and possibly its children, with the new information. The system keeps the UI consistent with the data model. The state cannot be altered through direct assignment but is required to be implemented through special setter methods provided by React to ensure predictable rendering behavior.  
State management is now one of the most critical parts of React-based development, impacting how data moves around the application and how the components talk to each other. It's interesting to mention that state isn't specific to React. There are other frameworks such as Angular, Vue, Svelte, and Solid that also implement reactive data tracking, though with their own syntax and lifecycle rules.authentication before rendering sensitive pages.

Each page in an application tends to have its own state, a set of data values that are required to display the UI correctly. States are not generic variables; they are data that, if changed, must be updated in the interface right away. For example, the count of likes on a post in a social network. This value must be stored in state, as when a user clicks the "like" button, they expect immediate response. The number should be updated without waiting for a full page reload.  
Another common example is a theme for a website. Most modern applications include the option to toggle between light and dark themes. The color scheme needs to be changed as soon as the user toggles it, without reloading the page. This makes theme preference an excellent candidate for state management.  
The theme example introduces an important notion, however: scope of state. In an SPA, the site is more of a collection of components, rather than separate, unrelated pages. Whereas some states are specific to one component, some must be global, meaning that they should be accessible to multiple components. A theme setting, for instance, must be applied to the entire app and be accessible anywhere in the component tree.

In React, there are several ways of transferring states between components. The simplest one is passing each state to another components in props. In this case, child will be re-rendered along with its parents in case of state changes. However, this approach is not optimal for global states. Since it would be a terrible infrastructure decision, to pass each global state to each component starting from the very first one. For this purpose, in React ecosystem there are several state managers. In the sample application Zustand will be used [42].

Zustand is a lightweight and effective state management library tailored for use in React applications. In contrast to more heavyweight solutions like Redux, Zustand has a minimalist API that minimizes boilerplate code, allowing users to quickly and easily initialize global state without losing flexibility. It's actually using React hooks internally, so developers can develop a centralized store in which state variables and updater functions exist together in a straightforward, intuitive way. Components subscribe to the common state and update it for only the specific components they care about, which maximizes rendering performance by preventing unnecessary re-renders. This kind of selective subscription with small size makes Zustand suitable for small and medium-sized projects or for any application where simplicity and speed matter. Developers define the store in a single function where state values and functions to update them are declared, then access and update that state directly within React components by using a custom hook. Zustand also supports middleware for adding functionality like logging, persisting state, or handling asynchronous data, giving the power needed for more complex scenarios while keeping it simple at its core. But as Zustand is an abstraction on top of React's hooks architecture, it is closely coupled to React and can be used with nothing else like Angular or Vue.

In the sample application Zustand manages two main global states. There are two stores, *AuthStore* and *PostStore*.  
*AuthStore* is responsible for managing a user account. It has several fields for storing data. A black background with text

AI-generated content may be incorrect.  
**IsLoggedIn**is a semantic field that is used to check whether a user is logged in or not. It is mainly used by components to display appropriate data. For example, if a user is not logged in, they cannot access a profile page. Instead, they will be redirected to Login page.   
**PublicKey** stores public key that is received from the store. Using this key application can verify integrity of the JWT token.  
**Username** contains username of a user currently logged in. When a user-specific request is sent to backend, this field is included in it. For example, it is used in requests which determine whether a user has liked a specific post or not.  
**AccessToken** contains JWT token. The authorization system uses two tokens. One is called access token the second one is a refresh token. The role of the first one is to offer user access to the data in the backend. It expires in several minutes. The second one is stored in a secure, http-only cookie in a browser. It is not accessible by JS scripts. It is used to obtain a new access token. It lasts 24 hours.   
When a user wants to log out of the account, all of those fields are reset. Also, an appropriate request is sent to the server. In response there is an empty cookie, which nullifies the cookie with the refresh token inside.

The second store is *PostStore*. It contains data about posts which are currently available on the main page. It has one field and one method that fetches ten new posts from the backend. Those are then displayed on the main page. Also, those posts are accessed by post pages. When a user clicks on a post, they are redirected to a page which contains information about that post. Then, program will check if this post was already fetched from the backend. And only if it is not present, a new one will be requested.

To conclude, global variable management can be a demanding and time-consuming task. In all of moder frameworks there is such a notion as a state. Which is a variable needed to store important information that should be especially responsive to user interactions. Some states are global and are accessible to all of the applications. Managing those can be done through special state managers. Libraries which offer convenient interfaces for developers to store and update states.

# Conclusion

It was never a simple job to create web applications. They are usually large, intricate systems that need to be carefully designed infrastructure and the joint effort of an entire group of programmers. In the background there is a stable server that serves as the backbone for the system, and it handles communication between the frontend or the end user. This server must also be the most secure part of the architecture since the backend has traditionally been thought of as the center of the application. It holds the main codebase, enforces access controls, and manages interaction with the database. This central position, being both the system's engine and its main security guard, is unchanged to this day.  
The frontend, in contrast, is the application's public face. It possesses an interactive, responsive, and visually appealing interface that engages users, and their experience is smooth and enjoyable. Its prime job is to engage users, keep them, and make them happy so they continue using the application in the long term. Today, the frontend may be even more decoupled from the backend and can even be an independent application by itself. It can even be remotely installed from another site, positioned strategically closer to the user to eradicate latency and ensure faultless performance.  
Aside from these basic components, many specialized tools now support developers. Version control systems facilitate cooperation and change tracking, secure database tools guard precious information, and containerization platforms simplify deployment and scaling. These technologies in concert have turned the process of how teams develop web applications on their head, making them more efficient, secure, and versatile than ever before.

In today's setting, web applications are still highly complex systems that take a broad skill set and body of knowledge to develop. But with today's tools being available, the process has come down significantly. Foremost among these are libraries and frameworks, which provide pre-built solutions to common problems and accelerate development.  
Frontend example is React, which is a library to make building user interfaces easier. It addresses one of the most critical tasks in frontend development: building dynamic, high-performance, and unkeepable UIs. React leverages a virtual DOM, which is a simulation of the actual DOM model. This enables React to make selective choices regarding which components should be re-rendered, minimizing performance wastage and rendering the app more performance efficient. It also offers a concept of state, a powerful instrument for tracking changes and controlling the timing at which elements of the interface are re-rendered. Furthermore, React encourages a component-based design, where the UI is divided into modular, reusable functions expressed in clean, declarative style. Apart from its core function, React is positioned at the center of a thick network of supporting libraries, allowing developers to apply only the tools they need for the specific requirements of a project.  
On the server-side, the Spring framework does the same job to facilitate and accelerate server-side application development. Spring provides a flexible and mature environment, including huge sets of modules that can be used to fit any project requirement. Its greatest advantage is Spring Boot, which makes it easy to configure applications as it provides most of this automatically. Spring's web application module has an in-built server to handle client connections and its security module to safeguard files, databases, and other sensitive items. The platform is also highly customizable, with developers able to optimize it to the unique requirements of a range of applications. Its declarative programming model also increases productivity, since developers specify what must be done, and Spring takes care of how it's done behind the scenes.  
Together, technologies like React and Spring allow developers to focus on the problem-solving and creative aspects of application development, allowing much of the boilerplate, repetitive programming to go to proven, battle-tested, and vetted frameworks.

Apart from programming languages and their environments, a number of supporting utilities play a major role in modern-day development. Of most significance is the version control system Git, which is invaluable to the maintenance and preservation of work. Git places repositories along with their whole histories in a space-efficient way so that developers can track changes and backtrack when they need to, without jeopardizing the integrity of the project. Its remote repository locations—such as GitHub or Bitbucket—also allow collaboration, and it is therefore an excellent choice for collaborative work, especially in big projects where coordination among numerous contributors is needed.  
Docker is also a very powerful solution that lets applications run smoothly in different platforms. It is based on the Java idea of "write once, run anywhere," but extends it to whole environments and not just programs. By encapsulating applications in portable containers, Docker ensures development is not distracted by the underlying server infrastructure. Since these containers can be run on all major operating systems, projects can be shifted between cloud providers and environments with simplicity, again allowing teams to be more productive and consistent.  
Equally important are databases, such as PostgreSQL, which can be the basis for almost any application. They offer secure and reliable data storage, complemented by intrinsic security features and backup facilities to ensure data integrity. Modern databases are able to handle diverse data types, away from the simple SQL relational tables in favor of more complex ones. Optimized for high performance, they are able to withstand high loads and service large numbers of concurrent clients reliably, thus being priceless for dependable, scalable systems.

As discussed earlier, both React and Spring encourage the transition to declarative programming. They remove the responsibility of low-level implementation details from developers, freeing them from the need to think as much about implementation and concentrating on architecture, design, and business logic instead. In React, a clear example of this is the JSX syntax, which makes it easier to render components. Instead of issuing rendering instructions by hand, the developers simply define what they want to be rendered in terms of HTML structure and allow React to handle the underlying rendering. The developer determines what gets rendered, not how it gets rendered to the DOM.  
Spring offers similar abstraction on the back end through its extensive use of annotations. One class may be annotated with a single annotation as a dependency to be injected, or as a REST endpoint. This cuts down on the developer having to code how dependencies are created, endpoints registered and accessed, or response bodies generated. Spring itself handles these low-level details with automation, freeing programmers to concentrate on issues at a higher level.  
These abstractions can be incredibly useful, especially for experienced practitioners who value productivity and would rather focus on delivering features rather than implementing technical basics all over again. They also have potential for harm, however, particularly to newcomers, students, or independent learners. Since so many underlying processes are hidden, new people may never get to know closely the underlying concepts such as networking, protocols, or security. This information deficiency can prove dangerous if their initial exposure to such topics is through a real event, such as a breach of security or data leak.  
Another area to be worried about is that these guidelines require only a basic level of understanding of their underpinning programming languages to adopt. For instance, JSX in React introduces a new syntax, and thus one can start writing components with shallow knowledge of JavaScript. It can also be done to start accepting connections in Spring with mere knowledge of Java classes and printing in the console. Though this ease is soothing, it creates a false impression of knowledge.  
For all these reasons, while libraries like React and Spring undeniably facilitate the process, it's well advised to learn them ideally once a programmer has acquired an understanding of the underlying language as well as a functional understanding of networking, protocols, and application security. It guarantees ease of abstraction with robust grasp behind the principles.