

Web Applications 101

MAY 16, 2022 BY ROBIN WIERUCH - [EDIT THIS POST](#)

 [Follow on Twitter](#)

@rwieruch

 [Follow on Facebook](#)

Let's start with a story: when people outside of my professional bubble ask me about what I am doing, I say "*I build websites*". If they are curious, I add that these websites are rather complex. If they keep on asking, I try to elaborate with some examples: Facebook, Spotify, Twitter. It's not that I work for these companies, but I hope that it gives them a good impression of "what kind of websites I build". However, most often the conversation doesn't go beyond "*I build websites*" and I am fine with that.

One website isn't equal to another website these days. Websites range from a marketing website for a product to a full-blown social media platform. As someone new to web development, it's not easy for you to grasp the whole landscape: what starts out as a traditional simple website with HTML and CSS, returned from a web server, turns into a far more complex full-stack application with sophisticated client-server communication and state management.



If you are already learning HTML, CSS, and JavaScript, and you don't know about the fundamentals of websites and web applications, then this comprehensive guide is for you.

In this walkthrough, I want to show you the evolution of web development from a simple website to a complex web application where we clarify terminology such as:

- client/server
- frontend/backend
- website/web application
- client application/server application
- REST/GraphQL
- web server/application server
- server-side rendering vs client-side rendering
- server-side routing vs client-side routing
- single-page application vs multi-page application
- code splitting, lazy loading, tree shaking, ...
- full-stack application
- static site generation
- BaaS, PaaS, IaaS, ...

overwhelming for beginners, especially towards the end. Let's get started ...

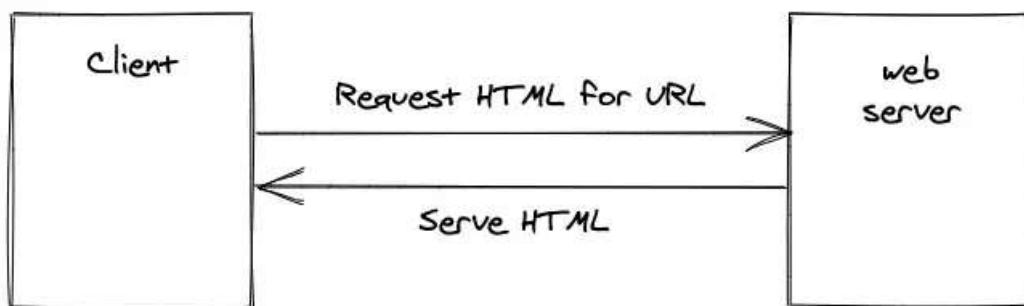
TABLE OF CONTENTS

- A traditional Website
 - What's the difference between Client and a Server?
 - What's the difference between a Web Server and an Application Server?
 - What happens when I change the path of an URL?
 - What happens when my website is more than HTML?
- Web 2.0: From Website to Web Application
- Single-Page Applications
- Code Splitting
- Full-Stack Applications
- Client-Server Communication
- Frontend and Backend
 - Microservices (Optional)
 - Backend-As-A-Service (Optional)
- Beyond Full-Stack Applications
 - Server-Side Rendering 2.0 (SSR)
 - Static Site Generation (SSG)



A TRADITIONAL WEBSITE

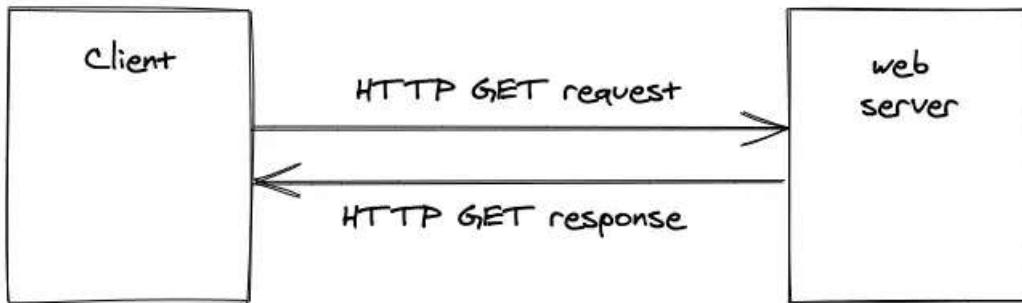
If you are learning about web development, you will most likely start with a traditional website built with HTML and CSS. A website which isn't styled (CSS) and doesn't have any logic (JavaScript) is just HTML.



If you navigate to a specific **URL** in your browser (e.g. Firefox) on your laptop or smartphone, a request is made to the web server responsible for that URL. If the web server is able to match the request to a website then it serves the HTML file for that website to your browser.

in front of every URL.

The communication between the client and the server is synchronous, which means that the client sends a request to the server and waits for a response from the server. Your website isn't displayed immediately because it takes time to send the request from a client to the web server and to receive a response back from the web server.



A HTTP request comes with four essential HTTP methods: GET, POST, PUT, and DELETE. The HTTP GET method is used to read a resource, whereas the remaining methods are used to write resources -- where a resource can be anything from HTML to JSON. All four methods can be abstracted to the infamous **CRUD** operations: Create, Read, Update, and Delete.



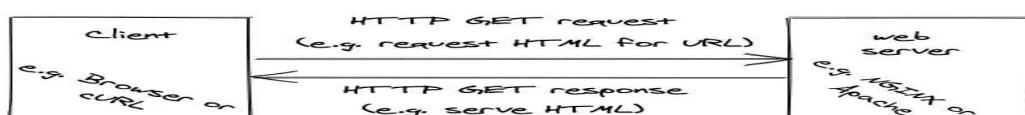
Create	->	HTTP POST
Read	->	HTTP GET
Update	->	HTTP PUT
Delete	->	HTTP DELETE

In our example of a website, which is served from a web server to a client by visiting a URL in a browser, the browser executes a HTTP GET method to read a HTML file from the web server.

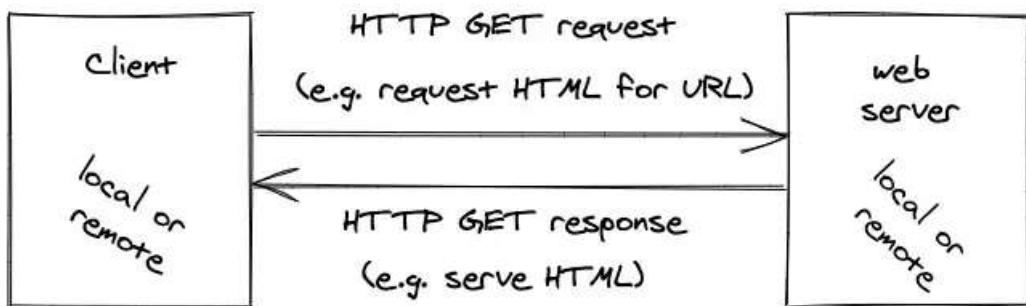
What's the difference between Client and a Server?

A Client is an entity that **consumes** a Server. It either reads resources from a server or writes resources to a server. For a traditional website, the client is your browser. If you navigate to a specific URL in your browser, your browser communicates with a server to request resources (e.g. HTML) to display a website. However, a client doesn't need to be a browser (e.g. `cURL`).

A Server is an entity that **serves** a Client. In the traditional sense of a website, a server reacts to a client's requests and either replies with resources (e.g. HTML, CSS, JavaScript) from HTTP GET requests or acknowledges manipulations from HTTP POST, PUT, DELETE requests. Popular web servers, which are a specific kind of server, are **NGINX** or **Apache**.



client. They work together, even though they don't need to be in the same place. For example, the browser on your machine is at your local location (e.g. Berlin in Germany) and the web server which serves a website to you is operating at a remote location (e.g. Frankfurt in Germany). A server -- which is just another computer -- is usually located somewhere else than your local machine. For the sake of developing a web application or website, you may have a server running on your local machine too (see [localhost](#)).



As a client doesn't necessarily need to be a browser on your local machine, it could be somewhere remote as well. But more about this later.



What's the difference between a Web Server and an Application Server?



A **Web Server** serves resources (e.g. HTML, CSS, and JavaScript) which are formats that can be transferred via HTTP. When a client requests resources from a web server, the web server fulfills the request by sending the resources back to the client. Normally, these resources are just files on the server. If HTML is sent to a client, then the client (in this case, the browser) interprets the HTML to render it.

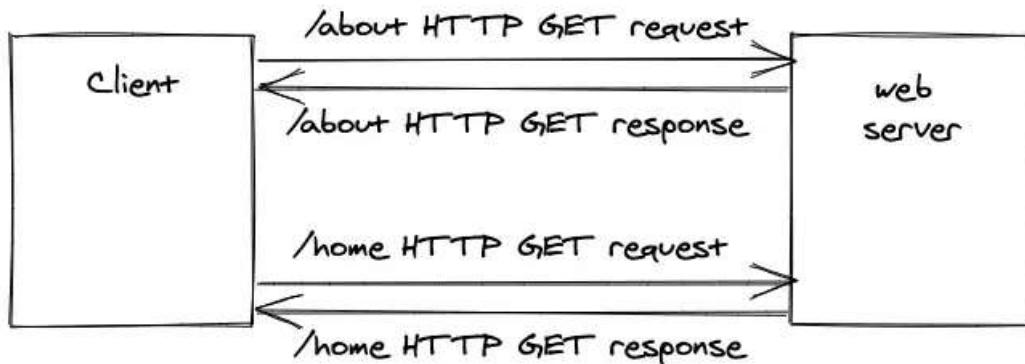


In contrast, an **Application Server** serves resources beyond HTML, CSS, and JavaScript. For example, [JSON](#) can be sent if a client requests data in a data friendly format. In addition, an application server isn't bound to a protocol. Whereas a web server is mainly used with the HTTP protocol, an application server can use other protocols (e.g. [WebSockets](#) for real-time communication). The most important fact is that an application server can have [implementation details](#) on its server-side in a specific programming language (e.g. [JavaScript](#) with [Node.js](#), [PHP](#), [Java](#), [Ruby](#), [C#](#), [Go](#), [Rust](#), and [Python](#)).

Both web servers and application servers can be classified as servers. So you will often hear people talk about servers when they mean one of these two. However, people often say server to mean a physical computer, which runs somewhere remote on which a web server or application server is running.

There are two more terms which you may meet: deploying and hosting. I will be brief about these terms: **deploying** describes the act of taking a website live on a server, **hosting** describes the continuous act of serving the website from this server. So, when developing a website on your own computer, you have to open it with a localhost URL, which just means that you are the local host of this website.

What happens when I visit a website at a URL and navigate around this domain (e.g. mywebsite.com) from a path (e.g. /about) to another path (/home)? For a traditional website, a new request is made by a client to a web server for every distinct URL.



For every URL, a distinct HTTP GET method is sent to the dedicated web server to fulfil the request. When a user accesses a website on its **/about** **path** (also called **page** or **route**) in a browser, for example `http://www.mywebsite.com/about`, the web server sends all the information about this URL back to the browser. This process is called **server-side routing**, because the server decides which resource is sent to the client on each URL. You will learn about client-side routing later.



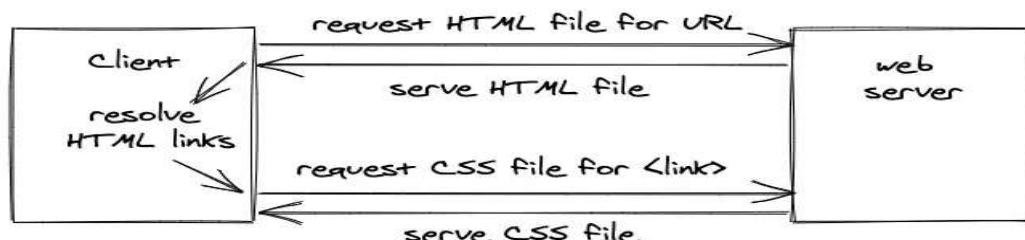
What happens when my website is more than HTML?

A modern website consists of HTML (structure), CSS (style), and JavaScript (logic). Without CSS a website wouldn't be shiny and without JavaScript a website wouldn't have dynamic interactions. Usually when CSS and JavaScript files are used, they are linked in a HTML file:

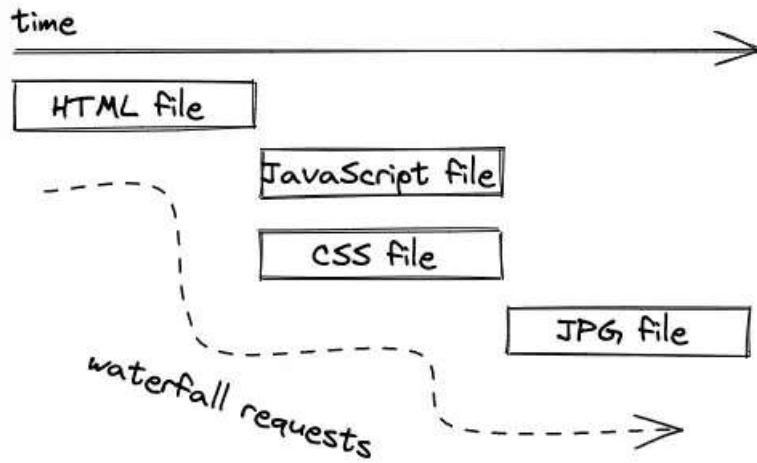
```

<link href="/media/examples/link-element-example.css" rel="stylesheet"
<h1>Home at /home route</p>
<p class="danger">Red text as defined in the external CSS file.</p>
  
```

If a browser asks a web server for the HTML of an URL, the web server sends back the HTML file which may include HTML tags that link to other resources like CSS or JavaScript files. For every link, another request is made to the web server to retrieve the files.

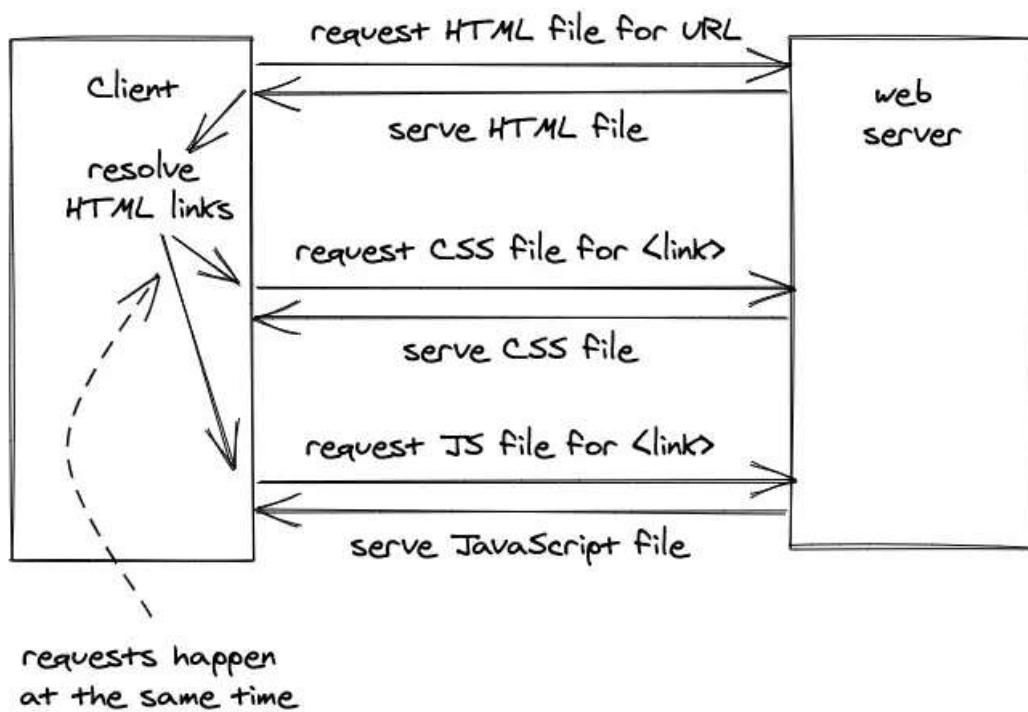


request to finish. In our example, the browser doesn't know that it needs to request the CSS file before the HTML file arrives with the HTML link tag. And in the next example the HTML file links to a JavaScript and CSS file, whereas the CSS file links to a JPG file (which may be used as CSS background for example).



However, at least if there are multiple references in one file, for example the initial HTML file that links to a CSS and a JavaScript file, these resources will be requested and resolved in parallel, as seen in the previous example, but also illustrated in the next one.

f
t
in

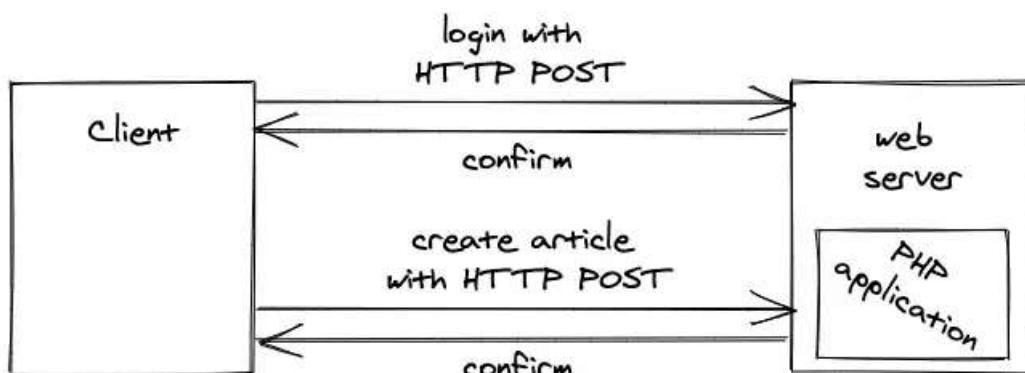


Eventually the browser will have all resources (e.g. HTML, CSS, JavaScript, PNGs, JPGs, SVGs) for a specific URL and interprets the HTML with all its included assets to display the desired result for you. It's ready for you to interact with it as a user.

WEB 2.0: FROM WEBSITE TO WEB APPLICATION

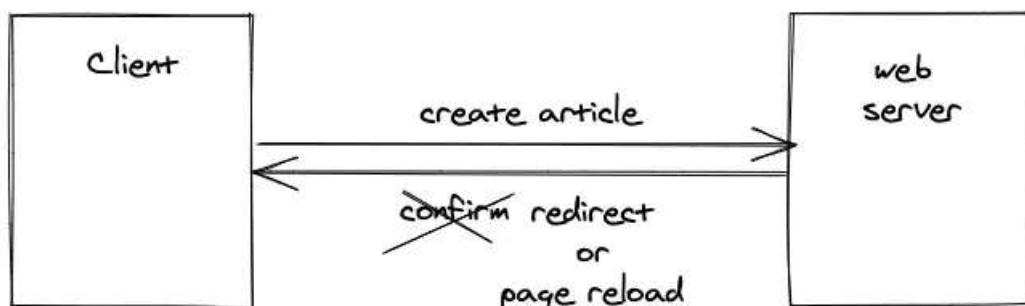
content; which led to **dynamic content**. Remember the HTTP methods from earlier? So far, we have only seen HTTP GET methods in action for reading resources, but what about the other HTTP methods?

With the rise of **content management systems** like Wordpress, a web server had to enable users to not only see resources, but also to manipulate them. For example, a user using a content management system must be able to log in, to create a blog post, to update a blog post, to delete a blog post, and to log out. At this time, the programming language **PHP** (which could be interpreted by a web server on the server-side) was the best fit for these kinds of dynamic websites.

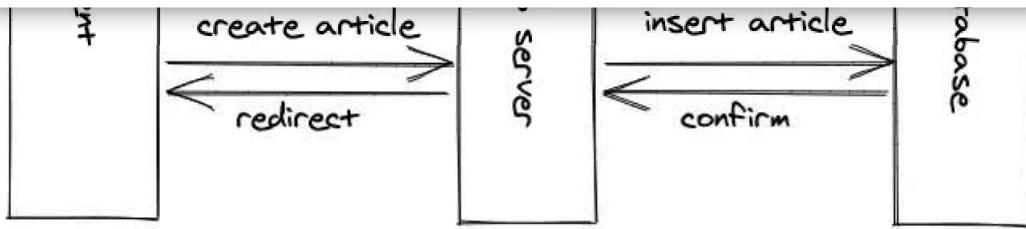


Having the logic on the server-side, developers are enabled to process reading and writing requests from their users. If a user wants to create a blog post (write operation), the user has to write the blog post in a browser and click a "Save" button to send the content to the server-side logic running on the web server. This logic verifies the user is authorized, validates the blog content, and writes the content in a database. All these permissions were not allowed to take place on a client, otherwise everyone would be able to manipulate the database unauthorized.

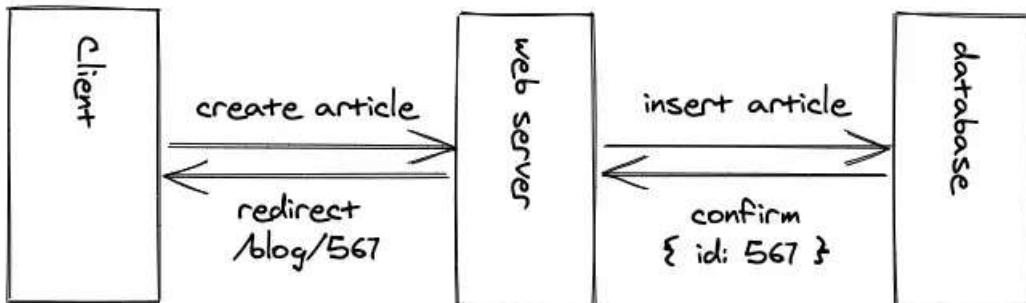
Since we still have **server-side routing**, the web server is able to redirect the user to a new page after the blog post has been created successfully. For example, the redirect could be to the newly published blog post. If there is no redirect, a HTTP POST/PUT/DELETE request usually leads to a page refresh/reload anyway.



Since users are able to create dynamic content now, we need to have a database to store this data. The database can be on the same physical server (computer) like the web server (most likely in the early days of Web 2.0) or on another remote computer (most likely in the modern era of web development).



Once the blog post is inserted in the database, an unique identifier may be generated for this blog post which can be used to redirect the user to the newly published blog post's URL. All of this still happens asynchronously.



Now, after a blog post has been created, how does a server send a HTML file for a blog post if the data for it isn't static, but instead stored in a database? That's where the principle of **server-side rendering** (not to mistake with server-side routing) comes into play.

Both Web 1.0 with consumer oriented websites (static content) and Web 2.0 with producer oriented websites (dynamic content) return HTML from the server. A user navigates to an URL in the browser and requests the HTML for it. However, for the dynamic content in Web 2.0, the HTML which is sent to the client isn't a static HTML file with static content anymore. Instead it gets interpolated with dynamic content from the database on the server:

```

<?php if ($expression == true): ?>
    This will show if the expression is true.
<?php else: ?>
    Otherwise this will show.
<?php endif; ?>
    
```

Templating engines for different programming languages (e.g. Pug for JavaScript on Node.js, Twig for PHP, JSP for Java, Django for Python) enable the interpolation of HTML and dynamic data before it's sent to the client. With the help of server-side rendering, user generated content can be served from a server to a client within HTML by creating the HTML on the fly when a client requests it.

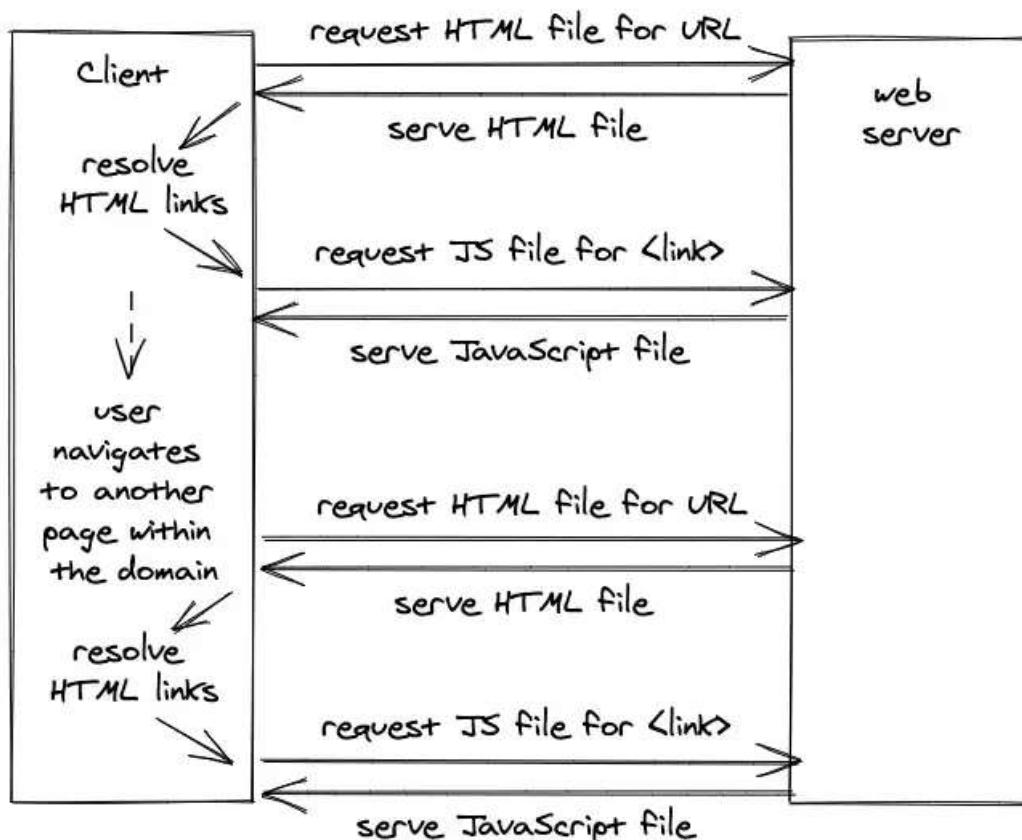
Are we still dealing with a Website here? Technically yes, but websites which go beyond static content by serving dynamic content from a web server (or application server) with a database may be called web applications as well. The line between both types is blurry though.

SINGLE-PAGE APPLICATIONS

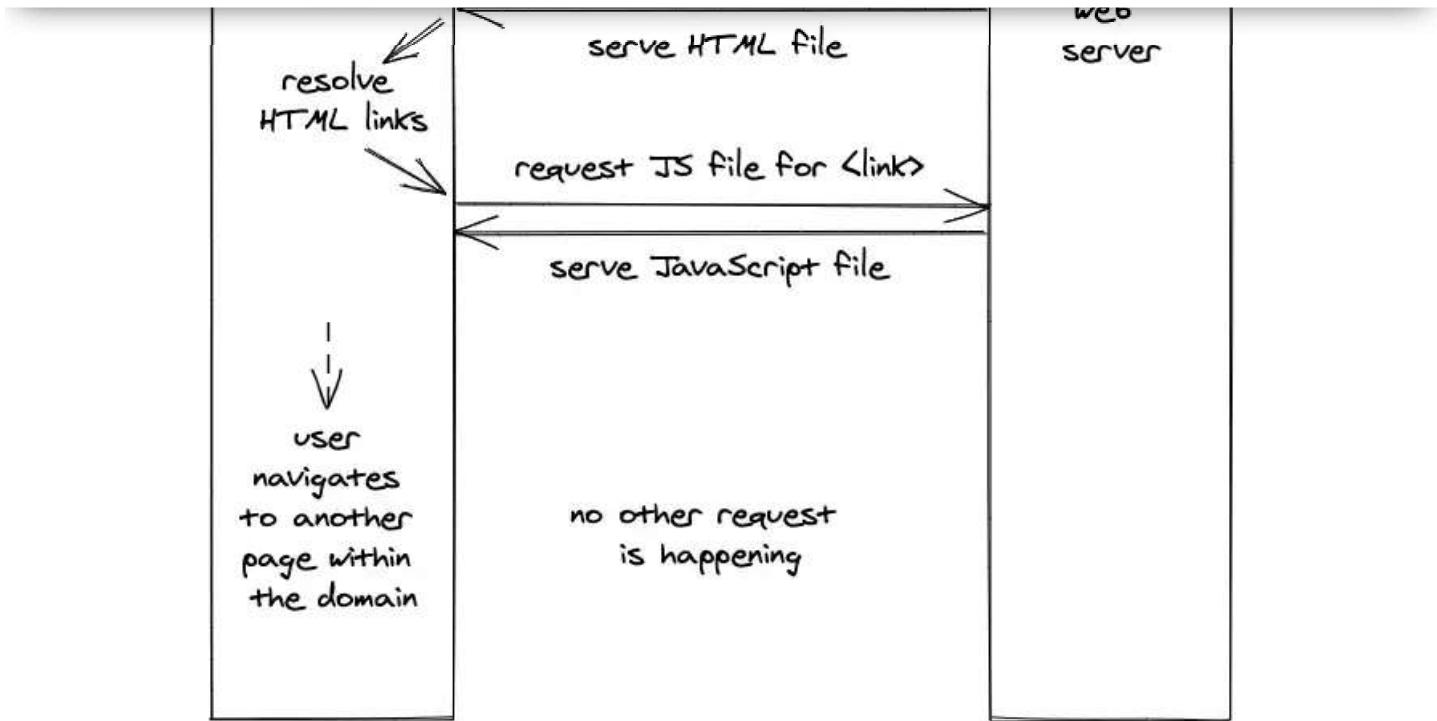
The rise of **single-page applications** (SPAs) after 2010 made JavaScript popular. But I am getting ahead of myself. Prior to this era, websites were mainly made with HTML plus CSS and only a sprinkle of JavaScript. The little JavaScript was used for animations or DOM manipulations (e.g. removing, adding, modifying of HTML elements), but not much beyond this. And `jQuery` was one of the most popular libraries to perform such tasks.

But who would have thought that entire applications could be built with JavaScript? A few of the earlier libraries/frameworks to write single-page applications in JavaScript were Knockout.js, Ember.js, and Angular.js; while React.js and Vue.js were released later. Most of them are still very active to this day in modern web applications.

Prior to single-page applications, a browser would request the HTML file and all linked files from a web server for a website. If a user happens to navigate from page (e.g. /home) to page (e.g. /about) within the same domain (e.g. mywebsite.com), there would be *a new request to the web server for every navigation*.



In contrast, a **single-page application** encapsulates the entire application in mostly JavaScript which has all the knowledge about how and what to render with HTML (and CSS) inside it. For the most basic usage of a single-page application, *the browser would request only once a HTML file with one linked JavaScript file for a domain*.



The requested HTML for a single-page application (here a React application) is just a middleman to request the JavaScript application (here `bundle.js`) which, after it has been requested and resolved on the client, will be rendered in the HTML (here `id="app"`):



```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello HTML File which executes a React Application</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="./bundle.js"></script>
  </body>
</html>
```

From there, React takes over with this little JavaScript from a `./bundle.js`:

```
import * as React from 'react';
import ReactDOM from 'react-dom';

const title = 'Hello React';

ReactDOM.render(
  <div>{title}</div>,
  document.getElementById('app')
);
```

In this little React application, only a variable called `title` is displayed in an HTML `div` element. However, everything between the HTML `div` element can be replaced with an entire HTML structure built with React components and its templating syntax JSX.

```

import ReactDOM from 'react-dom';

const App = () => {
  const [counter, setCounter] = React.useState(42);

  return (
    <div>
      <button onClick={() => setCounter(counter + 1)}>
        Increase
      </button>
      <button onClick={() => setCounter(counter - 1)}>
        Decrease
      </button>

      {counter}
    </div>
  );
};

ReactDOM.render(
  <App />,
  document.getElementById('app')
);

```

Increase Decrease 42



This is essentially a templating engine from earlier, but just executed on the client instead of the server and therefore this isn't server-side rendering anymore.

```

const App = () => {
  const [books, setBooks] = React.useState([
    'The Road to JavaScript',
    'The Road to React',
  ]);

  const [text, setText] = React.useState('');

  const handleAdd = () => {
    setBooks(books.concat(text));
    setText('');
  };

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={(event) => setText(event.target.value)}
      />
      <button
        type="button"
        onClick={handleAdd}
      >
        Add
      </button>
    </div>
  );
};

```

```
};

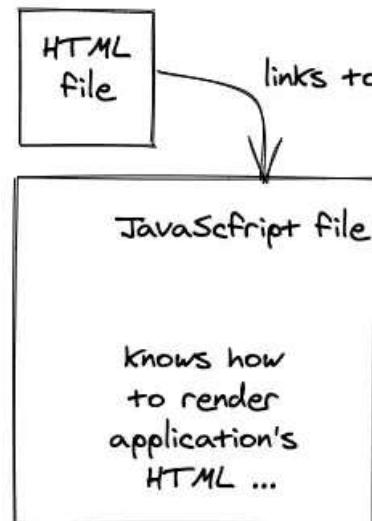
const List = ({ list }) => (
  <ul>
    {list.map((item, index) => (
      <li key={index}>{item}</li>
    )));
  </ul>
);
```

[Add](#)

The Road to JavaScript

The Road to React

Because of this change from server to client execution of the rendering, we call it **client-side rendering** now. In other words: Instead of serving the pre-rendered HTML directly from the web server, we serve mainly JavaScript from the web server which executes on the client and only then renders the HTML. Often the term SPA can be used synonymously with the term **client-side rendered application**.



If the SPA is requested only once from a web server, how does it work when a user navigates from one page to another page within the same domain (e.g. mywebsite.com/about to mywebsite.com/home) without requesting another HTML?

With the usage of a traditional SPAs, we also moved from server-side routing to **client-side routing**. The initially requested JavaScript file for the basic SPA has all the pages of a website encapsulated. Navigating from one page (e.g. /about) to another page (e.g. /home) wouldn't perform any request to the web server. Instead, a client-side router (e.g. React Router for React) takes over to render the appropriate page from the initially requested JavaScript file.



server-side rendering/routing while retrieving the entire application only once from a web server. It's a single page, because there is only one request made for the entire application, which is a single HTML page which links to one JavaScript file; which has all the actual UI pages encapsulated and executes on the client-side.

One could say that before we had single-page applications, we have been working with multi-page applications, because for every page (e.g. /about) a new request is made to the web server to retrieve all the necessary files for it. However, the term **multi-page application** isn't really a thing, because it was the default before single-page applications became popular.

Exercises:

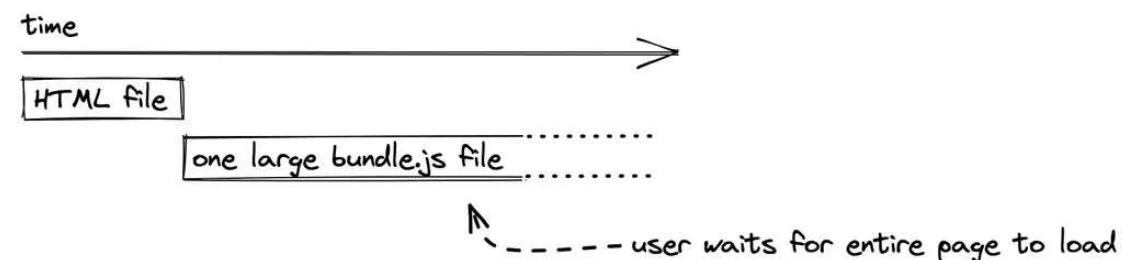
- Learn how to use React for single-page application development.
- Learn how to set up a React with Webpack application from scratch.



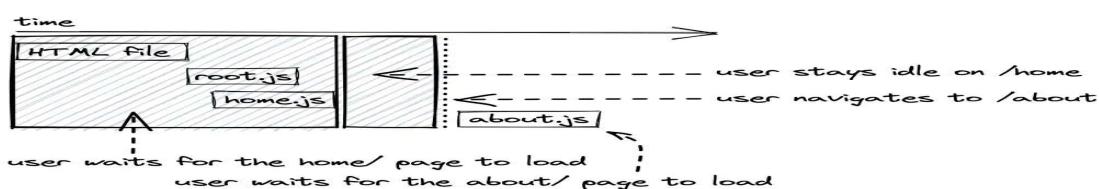
CODE SPLITTING



We have learned that SPAs get shipped in one small HTML file and one JS file as default. The JavaScript file starts small, but it grows in size when your application gets larger, because more JavaScript is packaged in one *bundle.js* file. This impacts the user experience of SPAs, because the initial load time to transfer the JavaScript file from web server to browser increases eventually. When all files are loaded, a user can navigate from page to page without interruptions (good). However, in contrast the initial load time decreases the user experience when the page is requested in the browser (bad).



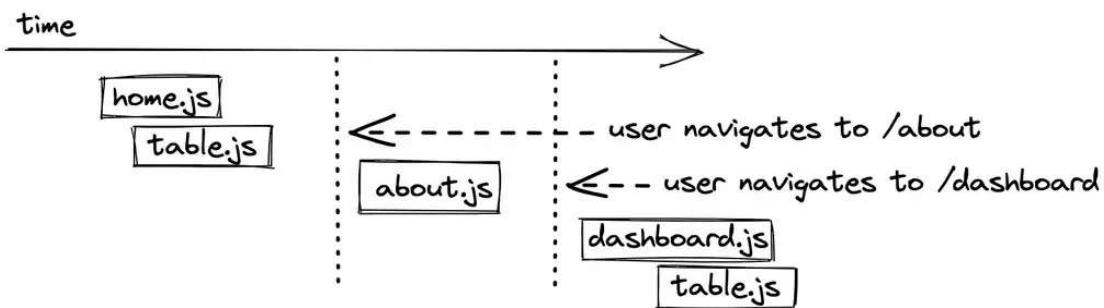
Requesting the entire application as JavaScript file becomes a disadvantage once the application grows in size. For a more sophisticated single-page application, techniques like **code splitting** (also called **lazy loading** in React + React Router) are used to serve only a fraction of the application that is needed for the current page (e.g. mywebsite.com/home). When navigating to the next page (e.g. mywebsite.com/about), another request is made to the web server to request the fraction for this page.



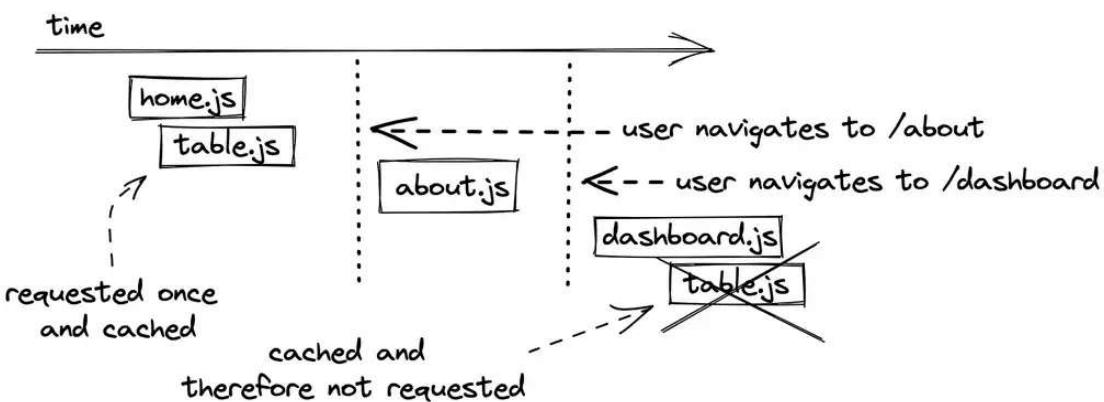
Code splitting enabled SPAs. For a traditional website, every time a user navigates to a new route a new HTML file (with optional CSS, JavaScript, and other asset files) is loaded. For the SPA with code splitting on a route level, every navigation leads to a newly requested JavaScript file.

Can we still call this single-page application or are we back at multi-page applications? You see how the terminology becomes fuzzy eventually ...

Code splitting does not need to happen on route level like in the previous scenario. For example, one can extract larger React components into their standalone JavaScript bundle too, so that it gets only loaded on pages where it is actually used.



However, as you can see this leads to redundant code that's requested from the web server. The same happens when a user navigates to a code split route twice, because it would get loaded from the web server twice as well. Hence we want the browser to cache (read: store in the browser's cache on the user's machine) the result.



Now what happens if the bundled `table.js` file changed, because we introduced a new feature to our table such as a [paginated view](#) or a [tree view](#)? If caching is enabled, we would still see the old version of the Table component in the browser.

As a solution to this problem, each new build of the application checks whether the bundled code has changed. If it has changed, it receives a new file name with a hash (e.g. `table.hash123.js` becomes `table.hash765.js`) based on a timestamp. When the browser requests a file with a cached file name, it uses the cached version. However, if the file has changed and therefore has a new hashed name, the browser requests the new file because the cached version is outdated.

Dropdown, code splitting can be applied too. Every component is a standalone JavaScript file then. When importing the Button component from the UI library, only the JavaScript from the Button gets imported but not the JavaScript from the Dropdown.

Continue Reading: [Learn how to create a React Button](#)

Continue Reading: [Learn how to create a React Dropdown](#)

For bundling of a React application (or library) into one or multiple (with code splitting) JavaScript files, another technique called **tree shaking** comes into play which eliminates dead code (read: unused code) so that it is not packaged in the final bundle. Historically the following bundlers were used in JavaScript (from past to recent):

- Grunt (2012)
- Gulp (2013)
- Webpack (2014+)
- Rollup (mainly libraries)
- esbuild (2020+)



Exercises:



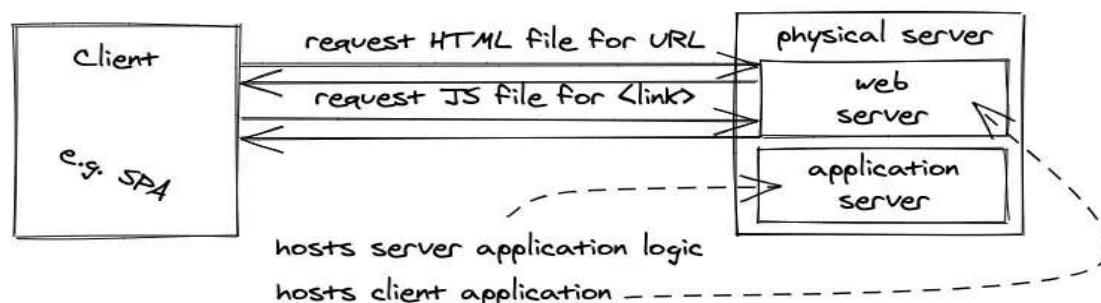
- Learn how to use React Router for client-side routing in React.
 - Learn how to use code splitting on a route level.



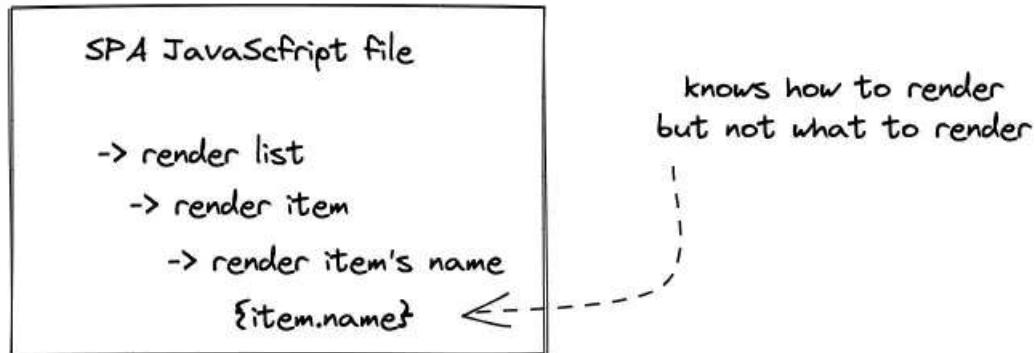
FULL-STACK APPLICATIONS

We are entering the paradigm of full-stack applications which became popular at the same time as SPAs. A full-stack application includes client (e.g. SPA) and server application. If companies are looking for full-stack developers, they often want to have someone who is able to create client-server applications on both ends. Sometimes the client and server share the same programming language (e.g. JavaScript with React on the client, JavaScript with Node.js on the server), but they don't have to.

Anyway, why do we need full-stack applications? The need for full-stack applications was born due to the rise of single-page applications on the client-side.



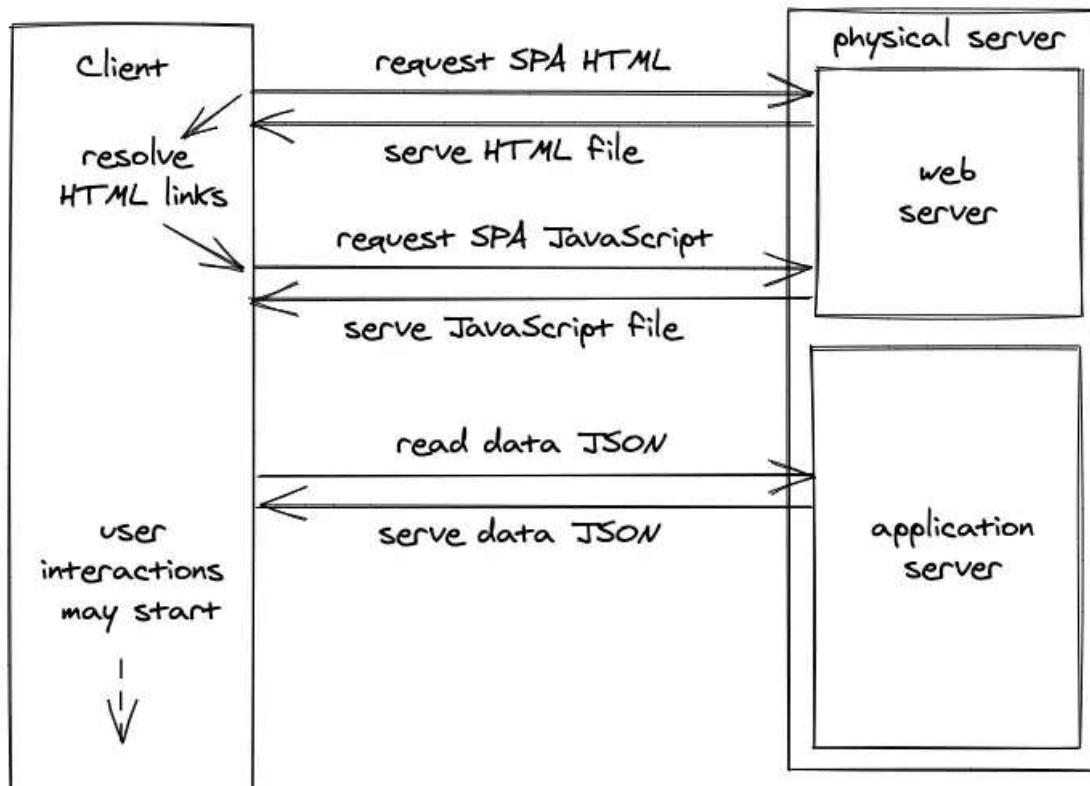
applications (e.g. React applications). Rendering static content is fine, but how do we render dynamic content, for example user specific content like a blog post (see Web 2.0, but this time with client-side rendering), if only JavaScript (and a little HTML) is served from a web server to a client when dealing with SPAs where client-side rendering took over?



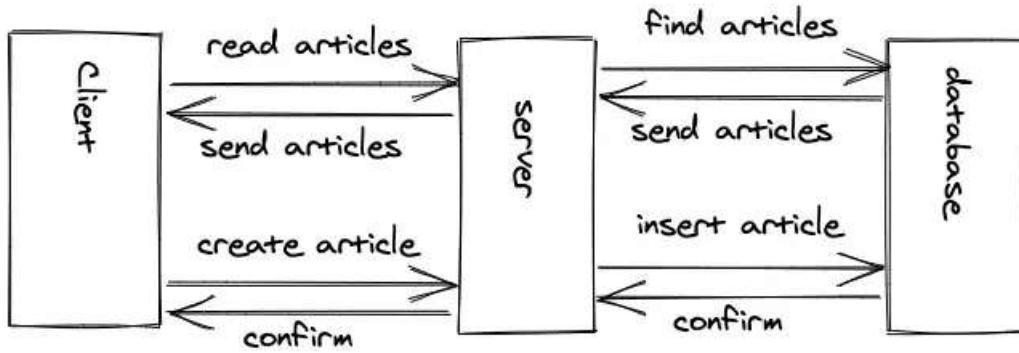
The SPA application -- which is encapsulated in a JavaScript file -- doesn't have any user specific data. It's just the logic for the pages; what they look like and how they behave for user interactions. The actual data isn't embedded in there, because it still sits somewhere in a database and doesn't get interpolated on the server anymore. That's the tradeoff you have to make when moving from server-side rendering to client-side rendering.



Therefore, another request from client to server (application server written with JavaScript/Node.js or another programming language) has to be made for requesting the missing data to fill in the blanks on the client. The client-side templating engine (e.g. JSX in React) takes care of interpolating content (data) with structure (HTML).



Once everything is rendered in the browser, a user starts to interact with the application -- for instance by creating a new blog post. JSON is a preferred format to send data from client to server and vice versa. The server deals with all the requests from the client by reading from or writing to the database; which can be on the same physical server, but doesn't need to be (e.g. just sitting on another physical server).



Client-side rendered applications (SPAs) come with the caveat of not having all the data at their disposal from the beginning. They have to request everything to fill in the blanks. As an end user surfing the web, you will notice client-side rendered applications in two ways:



- First, there are loading spinners (almost everywhere), sometimes from having a loading spinner for the whole page and in a split second afterward multiple loading spinners (waterfall requests) for smaller widgets, because requesting the data happens after rendering the initial page.
- And second, the navigation from route to route is instantaneous (code splitting not included, because then it feels a bit slower due to the extra bundle request to the server). That's the benefit we gain from an SPA.

Continue Reading: [Learn how to fetch data in React](#)

In addition to the extra data fetching roundtrip, client-side rendered applications have to deal with the challenge of state management, because user interactions and data need to be stored and managed somewhere on the client.

Consider the following challenge when working with an SPA: a user visits an author website where they can publish blog posts. At the current page, the user sees all of their blog posts and therefore all these blog posts need to get fetched when loading this page. These fetched blog posts are saved as state in memory on the client-side in code. Now, when a user starts to interact with the page and its data, a button for each blog post allows the user to delete each entry individually. What happens when a user clicks a delete button? Let's walk through this scenario:

A user clicks the delete button which sends a request to the application server with the identifier of the blog post as payload and the instruction (usually a HTTP DELETE is sufficient) to delete it. After all permission checks (e.g. is user authorized, is blog post

confirms the success of the operation back to the server and the sever sends a response back to the client. Now the client either removes the blog post from its local state in memory or fetches all blog posts again from the server and replaces the in-memory blog post with the updated list of blog posts.

Continue Reading: [Learn about State in React](#)

When performing client-side routing, requests for data (e.g. blog posts) can be minimised through state management. This means that ideally a user navigating from one page to another and then back to the initial page should not trigger a second request for data that is needed for the initial page. Instead it should already be cached on the client by using state management.

Last, but not least, the interface between a client and a server is called an **API**. In this case, it's a specific kind of API between two remote entities, here a client and a server, however lots of things are called APIs in programming.

Exercises:

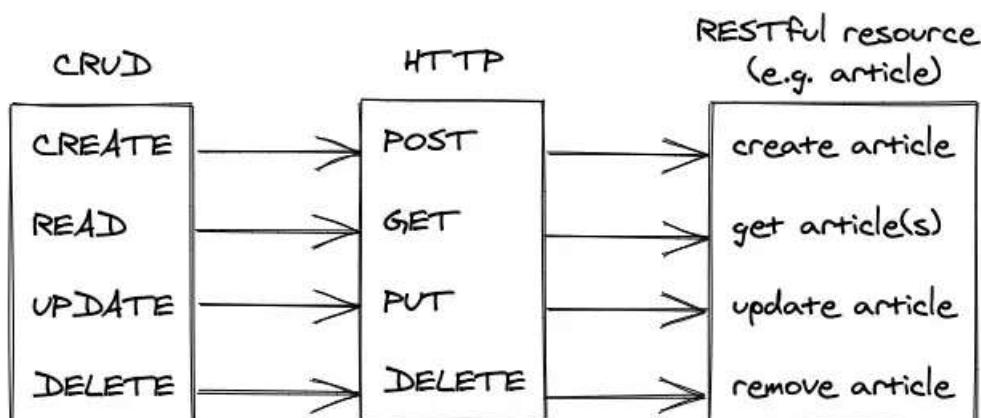
- Read more about different kinds of APIs.



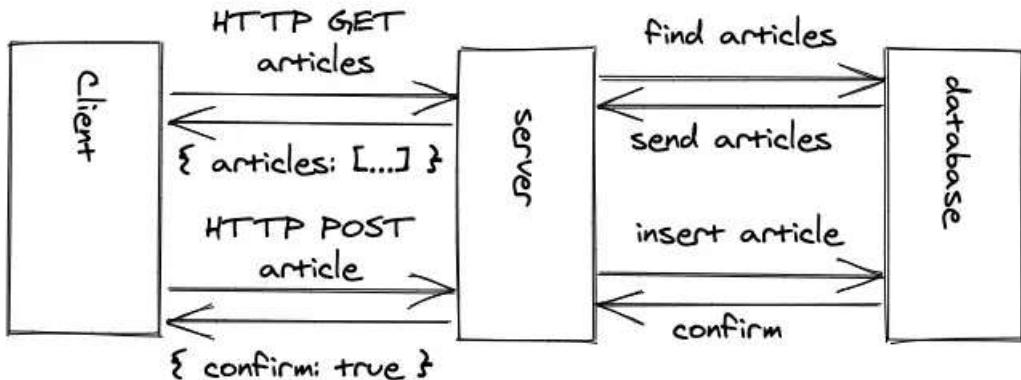
CLIENT-SERVER COMMUNICATION

Traditional full-stack applications use **REST** as their API paradigm; which employs HTTP methods for CRUD operations. Previously we have already been using HTTP methods for CRUD operations -- without following clear constraints -- among files and user interactions like creating a blog post with server-side languages like PHP.

However, when using a REST API we are using these HTTP methods on **RESTful resources**. For example, a RESTful resource could be a blog post. A user can read blog posts with a HTTP GET from the application server or create a new blog post with a HTTP POST on the application server.



sending and receiving HTTP requests and responses. REST is a communication paradigm which is free of a data format (it has been XML in the past, but these days it is JSON) and programming language.



A modern alternative to REST is [GraphQL](#) for APIs between clients and servers. GraphQL isn't bound to a data format either, and in contrast to REST not bound to HTTP, but most often you will see HTTP and JSON used here as well.



With the technology discussed up to this point, full-stack applications decouple client and server applications. Both communicate via a well-chosen API (e.g. REST or GraphQL). While the client application renders everything necessary for the web application in the browser, the server application handles requests from the client for read and write data.

Exercises:

- Learn how to create a REST API with Node.js.
- Read more about why you would want to use GraphQL instead of REST.
- Learn how to use GraphQL for full-stack JavaScript applications.

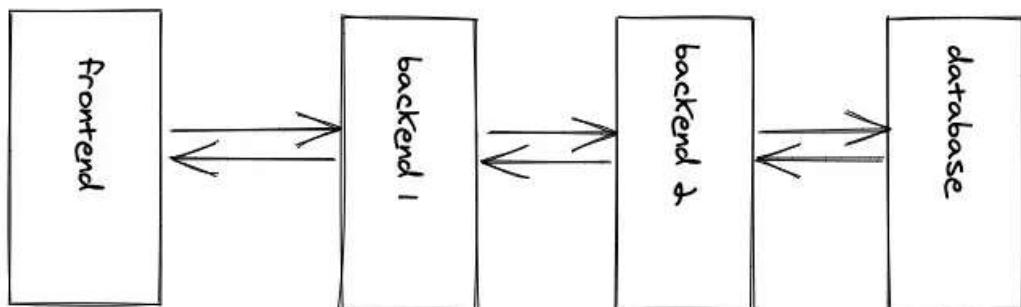
FRONTEND AND BACKEND

We haven't discussed the terms frontend and backend yet, because I didn't want to add too much information upfront. A frontend application may be everything a user sees in the browser (e.g. website, web application, SPA). Hence you will see frontend developers most often work with HTML/CSS or a library like React.js. In contrast, a backend is often the logic behind the scenes: It's the logic which reads from and writes to a database, the logic that speaks to other applications, and often the logic that offers an API.

Both entities result in a client-server architecture (frontend and backend relationship) whereas the backend would be needed for (A) business logic that shouldn't be exposed as source code to the frontend application (e.g. authorization) -- otherwise it would be accessible in the browser -- or for (B) establishing sensitive connections to third-party data sources (e.g. database(s)).

~~always talk backend here. These terms cannot be exchanged that easily. Whereas a frontend application is usually something seen in the browser, a backend usually performs business logic that shouldn't be exposed in a browser and often connects to a database as well.~~

But, in contrast, the terms client and server are a matter of perspective. A backend application (Backend 1) which consumes another backend application (Backend 2) becomes a client application (Backend 1) for the server application (Backend 2). However, the same backend application (Backend 1) is still the server for another client application which is the frontend application (Frontend).



```

// Frontend: Client of Backend 1
// Backend 1: Server for Frontend, also Client of Backend 2
// Backend 2: Server for Backend 1
  
```

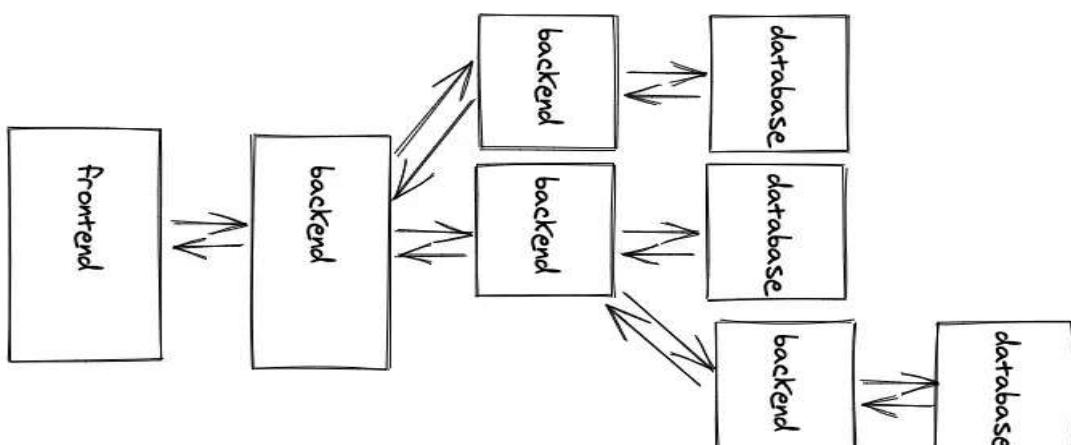


If you want to answer the client-server question if someone asks you what role an entity plays in a client-server architecture, always ask yourself who (server) is serving whom (client) and who (client) consumes whom's (backend) functionalities?



Microservices (Optional)

For example, microservices is an architecture which splits up one big backend (also called **monolith**) into smaller backends (**microservices**). Each smaller backend may have one domain specific functionality, but they all serve one frontend (or multiple frontends) after all. However, a backend can also consume another backend, whereas the former backend becomes a client and the latter backend the server.



programming language, while all backends are able to communicate with each other via APIs. It doesn't matter which API paradigm they choose, whether REST API or GraphQL API, as long as a client who speaks to their servers understands the API specification. It can also happen that a frontend doesn't speak to only one backend, but to multiple backends side-by-side.

Exercises:

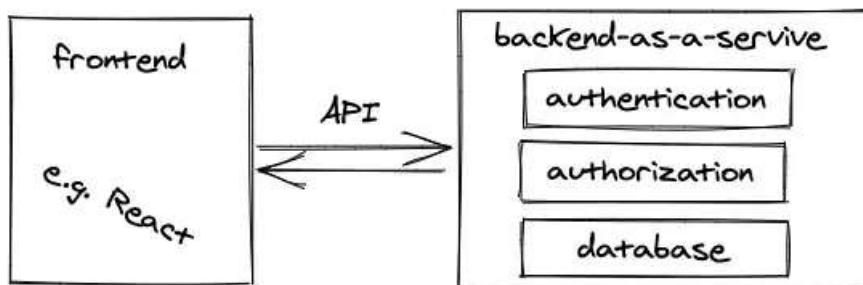
- Read why I stopped using Microservices as a solo developer.

Backend-As-A-Service (Optional)

In a traditional sense, a single backend application, which is only there to serve one frontend application, often connects to a database. That's a typical full-stack application. However, most often the backend application doesn't do much except for reading from and writing to a database, permitting certain users to perform certain actions (authorization), or authenticating (e.g. log in, log out, register) the user in the first place. If this is the case, it's often not needed to implement a backend application yourself.



Firebase (by Google), one solution for a backend-as-a-service, offers a database, authentication and authorization as a backend out of the box. A developer is only left with implementing the frontend application (e.g. React application) which needs to connect to this backend-as-a-service.



A backend-as-a-service (BaaS) such as Firebase allows a developer to get up and running with their frontend application very quickly. Everything from authentication, authorization and database is done for you. In addition, most BaaS offer hosting as well, for example your React application can be hosted with Firebase too. So Firebase serves your React application to your client (browser) and enables your application to speak to it for all the other features (e.g. authentication, database). A popular open source alternative to Firebase is Supabase.

Exercises:

- Learn how to use Firebase with React.

If all of this wasn't too confusing for you yet, try to keep up with me with the latest developments of full-stack applications. With all the development from traditional website to full-stack applications, you may have noticed the shifts from X to Y often make things more complicated ...

- Server-Side Routing (X) to Client-Side Routing (Y)
 - large bundle size which can be solved by code splitting though
- Server-Side Rendering (X) to Client-Side Rendering (Y)
 - extra (waterfall) requests for data
 - extra data fetching and state management work for developers
 - lots of loading spinners for end user

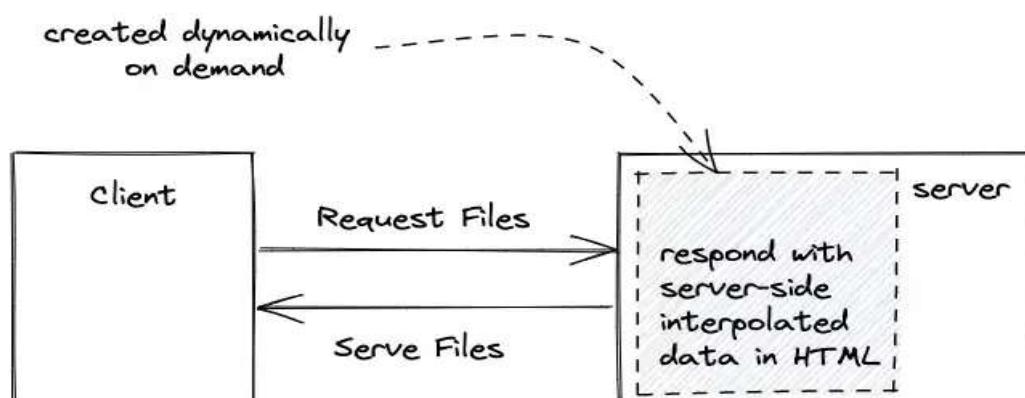
In the following, I want to present you two approaches, whose philosophies (SSR, SSG) aren't new but which are super powerful when used with modern libraries (e.g. React) and meta frameworks on top (e.g. Next.js, Gatsby.js) which make these approaches possible. I am a React developer, that's why the suggestions for these technologies are biased, however, I am sure you may find similar technologies for your preferences as well.



Server-Side Rendering 2.0 (SSR)

We have learned about server-side rendering for Web 2.0 before. At a later point in time, full-stack applications decoupled client and server and introduced client-side rendering with libraries like React. So what about taking one step back again and use React for server-side rendering?

When using the popular Next.js framework, which sits on top of React, you are still developing React applications. However, everything you implement in Next.js will be server-side rendered React. In Next.js, you implement each page (e.g. /about, /home) with React. When a user navigates from page to page, only a fraction of the server-side rendered React is sent to the browser. The great thing about it: You can already request the data to fill in the blanks on the server, interpolate the data with React, and send it to the client without any gaps.



client-side, and only starts to request the data to fill in the gaps once it got initially rendered with no data on the client. With SSR React, you can interpolate the data in React already on the server, but also can choose to fetch the data on the client when the application got rendered. Both options, client-side rendering and server-side rendering can be mixed up.

- Advantage: Client receives HTML already populated with data (improvements for UX and SEO).
- Disadvantage: Client may have to wait longer, because the populated HTML gets created on the fly on the server (HTTP caching minimizes the problem).

Exercises:

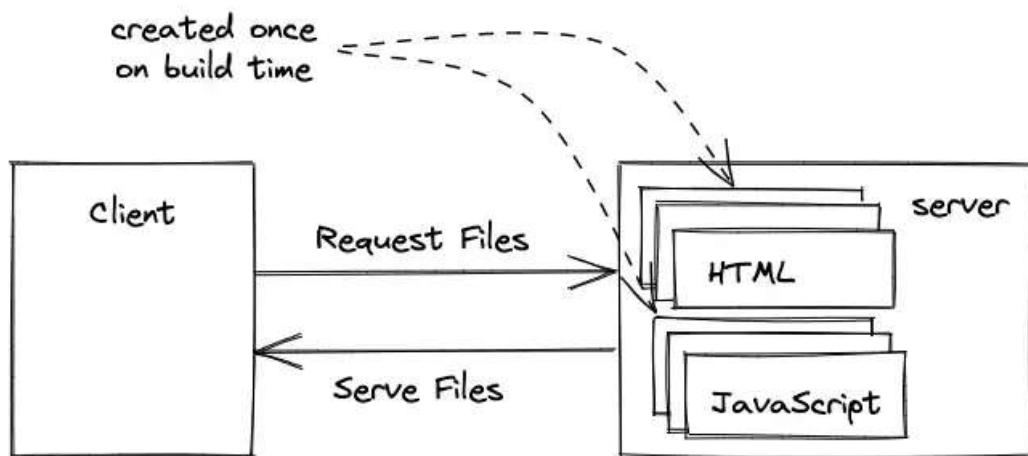
- Learn how I use Next.js for my course website.
- Learn Next.js for React.

Static Site Generation (SSG)

f Traditional websites use static files from a web server to be rendered on a browser. As we have learned, there is no involvement of an application server and no involvement of server-side rendering. The approach of a traditional website is pretty straightforward, because a web server just hosts your files and on every URL a user visits your browser makes a request to get the necessary files. So what if we could use React for static files?



React per se isn't made for static files. Instead, React is just JavaScript files which create the application on the fly on the client-side. However, Gatsby.js, a framework which sits on top of React, is used for static site generation for React applications. Gatsby takes a React application and compiles it to static HTML and JavaScript files. Then all these files can be hosted on a web server. If a user visits a URL, the static files are served to the browser.



In contrast to server-side rendering React, the static files are not created on the fly when a user requests them but are only created once on build time. This can be a drawback for dynamic content where the data changes often (e.g. eCommerce), however, for

Exercises:

- Learn how I use Gatsby.js for my website.
- Learn Gatsby.js for React.

. . .

If anything is missing in this marathon from website to web application, please let me know. I hope you enjoyed reading it! If you think it would be a good idea to add more content to this blog post and release it as 101 web development, let me know and sign up for my Newsletter to hear more about it :-)



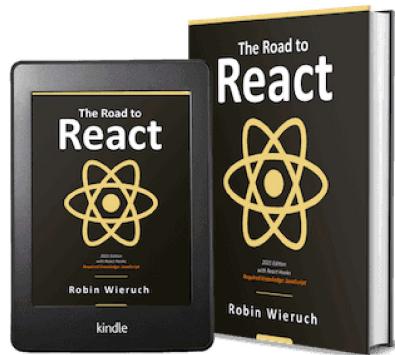
KEEP READING ABOUT REACT >

HOW TO USE GRAPHQL IN GATSBY

Gatsby is an open-source framework based on React that helps build websites and apps. It allows you to build your website and apps using React and then generates HTML, CSS, and JS when you build for...

REACT SEMANTIC UI TUTORIAL FOR BEGINNERS

Semantic UI is a UI component framework for theming websites. Semantic UI enables developers to build websites with fast and concise HTML, along with a complete mobile responsive experience. Semantic...



THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like

50.000+ readers.

GET THE BOOK

Get it on Amazon.

- ✓ Join 50.000+ Developers
- ✓ Learn Web Development
- ✓ Learn JavaScript
- ✓ Access Tutorials, eBooks and Courses
- ✓ Personal Development as a Software Engineer

[SUBSCRIBE](#)[View our Privacy Policy.](#)**PORTFOLIO**[Online Courses](#)[About me](#)[Open Source](#)[How to work with me](#)[Tutorials](#)[What I use](#)[How to support me](#)**ABOUT**

© Robin Wieruch

[Contact Me](#) [Privacy & Terms](#)